WebSphere Message Broker

IBM

# User-defined Extensions

*Version 6  Release 1*

WebSphere Message Broker

# User-defined Extensions

*Version 6  Release 1*

# Contents

# About this topic collection

This PDF file has been created from the WebSphere Message Broker Version 6.1 (fix pack 8 update, July 2010) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the "Related Links" sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but result in a "file not found "error message. Use the online information to navigate freely between topics.

**Feedback**: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

# Part 1. Developing user-defined extensions

# Developing user-defined extensions

A user-defined extension is a component that you design and implement to extend the function of WebSphere® Message Broker.

You can create and implement the following types of user-defined extension:
- User-defined nodes
- User-defined parsers
- User-defined exits

The user-defined nodes, parsers, and exits that you create can be used with the nodes and parsers that are supplied with the product, and with nodes and parsers that are supplied by independent software vendors.

For information about each type of user-defined extension that you can create, see the relevant topics listed under the related links. These topics help you to understand how your user-defined extension interacts with other components of WebSphere Message Broker, such as message flows and their associated execution groups. A good understanding of the broker architecture helps you to plan and construct user-defined extensions more effectively.

## User-defined extensions overview

A user-defined extension is an optional component that you design and create to extend the functionality of WebSphere Message Broker.

You can create the following types of user-defined extension:
- Input nodes
- Message processing nodes
- Output nodes
- Parsers
- User exits

The user-defined nodes and parsers that you create can be used with the nodes and parsers supplied with the product, and with nodes and parsers that are supplied by other vendors. You can configure a user-defined node to use a user-defined parser.

You can write user-defined exits and parsers only in the C programming language. You can write user-defined nodes in the C or the Java™ programming languages. You must compile user-defined nodes and parsers that are written in C into a loadable implementation library (LIL), and user exits that are written in C into a loadable exit library (LEL): that is, a shared library on Linux® and UNIX® systems, or a dynamic link library (DLL) on Windows® systems. You must package user-defined nodes that are written in Java as a JAR file. You must import any user-defined nodes that you create into the workbench before you can use them.

The samples gallery on the start screen of the workbench has examples of user-defined nodes and parsers. Look at the following sample for an example of how a node is created and used.

- User-defined Extension

You can view samples information only when you use the information center that is integrated with the Message Broker Toolkit or the online information center.

To achieve platform independence, use the ANSI standard C or Java programming languages, and avoid platform-specific code in your user-defined extension.

The related links help you to understand how your user-defined extensions interact with other components of WebSphere Message Broker, such as message flows and execution groups. A good understanding of the broker architecture helps you to plan and construct user-defined extensions more effectively.

## Why use a user-defined extension?

Use a user-defined node or parser when the built-in resources do not provide the required functions.

Before you start to create your user-defined extension, be clear about its purpose. Most tasks can be performed by using the functions already provided with WebSphere Message Broker, but you might need a user-defined extension for your particular task.

To write user-defined extensions you need to be a skilled programmer, with some knowledge of WebSphere Message Broker and its architecture, therefore make sure that you have the skills and knowledge required. You also need the time to test and debug your user-defined node or parser, and a safe environment in which to do this.

Also note that the maintenance and servicing of your own user-defined extensions is your responsibility. You should ensure that there will be someone available who can perform future updates or fixes.

A user-defined extension might be appropriate in the following situations:
- When you cannot manipulate the supplied nodes or parsers to perform the function you require. For example, you might want to connect to another software component in your message flow outside of WebSphere MQ. If there is no supplied node for doing this, you must create your own.
- When you can improve performance, ease of use, or reliability by using your own user-defined extensions in place of the supplied nodes or parsers.
- If the available choices are not appropriate for your requirement. You can create user-defined extensions to handle internal, customer-specific, or generic commercial messages formats.

Consider the following design factors when you are planning or writing a user-defined node or parser. You should be familiar with the concepts covered in the following topics before designing a user-defined extension.
- "Errors and exception handling" on page 5
- "Storage management in user-defined nodes" on page 7
- "String handling in user-defined nodes" on page 7
- "Threading considerations for user-defined extensions" on page 8
- "ODBC restrictions for user-defined nodes" on page 8
- "User-defined extensions in the runtime environment" on page 9
- "Node and parser factory behavior" on page 10

## Errors and exception handling

Correct handling of errors and exceptions is important for correct broker operation. You must consider how and when your user-defined extension must handle errors and exceptions.

The errors and exception handling here describes factors that you must consider when you develop user-defined extensions for WebSphere Message Broker in the C programming language. If you are developing user-defined extensions using the Java programming language, you can use standard Java error and exception handling methods. If, for example, WebSphere Message Broker throws an exception internally, a Java exception of class MbException is made available.

The broker generates C++ exceptions to handle error conditions. These exceptions are caught in the relevant software layers in the broker, and are handled accordingly. However, programs written in C cannot catch C++ exceptions, and all exceptions thrown, by default, bypass all C user-defined extension code and are caught in a higher layer of the broker.

Utility functions, by convention, typically use the return value to pass back requested data; for example, the address or handle of a broker object. The return value sometimes indicates that a failure has occurred. For example, if the address or handle of a broker object could not be retrieved, zero (CCI_NULL_ADDR) is returned. Additionally, the reason for an error condition is stored in the return code output parameter, which is, by convention, part of the function prototype of all utility functions. If the utility function completed successfully and `returnCode` was not null, `returnCode` contains CCI_SUCCESS. Otherwise, it contains one of the return codes described here. You can test the value of `returnCode` to determine whether a utility function was successful.

If the call to a utility function causes the broker to generate an exception, the error is visible to the user-defined extension only if it specified a value for the `returnCode` parameter to that utility function. If a null value was specified for `returnCode`, and an exception occurs:

- The user-defined extension is not be aware of that exception
- The utility function does not return to the user-defined extension
- Execution control passes to higher layers in the broker stack to process the exception

Therefore, a user-defined extension cannot perform its own error recovery. If, however, the `returnCode` parameter is specified, and an exception occurs, a return code of CCI_EXCEPTION is returned. In this case, cciGetLastExceptionData or cciGetLastExceptionDataW (the difference being that cciGetLastExceptionDataW returns a CCI_EXCEPTION_WIDE_ST which can contain Unicode trace text) can be used to obtain diagnostic information on the type of exception that occurred. The data is returned in the CCI_EXCEPTION_ST or CCI_EXCEPTION_WIDE_ST structure.

If there are no resources to be released, do not set the returnCode argument in your user-defined extension. Not setting this argument allows exceptions to bypass your user-defined extensions. These exceptions can then be handled higher up the WebSphere Message Broker stack, by the broker.

Message inserts can be returned in the CCI_STRING_ST members of the CCI_EXCEPTION_ST structure. The CCI_STRING_ST allows the user-defined extension to provide a buffer to receive all required inserts. The broker copies the

data into this buffer, and returns the number of bytes output and the actual length of the data. If the buffer is not large enough, no data is copied and the "dataLength" member can be used to increase the size of the buffer, if required.

The user-defined extension can set a non-null value for `returnCode` and provide its own error recovery, if required. The utility function calls return to the user-defined extension and pass their status through `returnCode`. All exceptions that occur in a utility function must be passed back to the broker for additional error recovery to be performed; that is, when CCI_EXCEPTION is returned in `returnCode`. You do this by calling cciRethrowLastException, after the user-defined extension has completed its own error processing. Calling cciRethrowLastException causes the C interface to re-throw the last exception so that it can be handled by other layers in the broker. In the same way as the C exit call, cciRethrowLastException does not return in this case.

If an exception occurs and is caught by a user-defined extension, the extension must not call utility functions except cciGetLastExceptionData, cciGetLastExceptionDataW, or cciRethrowLastException. An attempt to call other utility functions results in unpredictable behavior that can compromise the integrity of the broker.

If a user-defined extension encounters a serious error, cciThrowException or cciThrowExceptionW can be used to generate an exception that is processed by the broker in the correct manner. The generation of such an exception causes the supplied information to be written to the system log (syslog or Eventviewer) if the exception is not handled. The information is also written to trace (if trace is active).

**Types of exception and broker behavior:** The broker generates a set of exceptions that can be passed to a user-defined extension. These exceptions can also be generated by a user-defined extension when an error condition is encountered. The exception classes are:

**Fatal** Fatal exceptions are generated when a condition occurs that prevents the broker process from continuing execution safely, or where it is broker policy to terminate the process. Examples of fatal exceptions are a failure to acquire a critical system resource, or an internally-caught severe software error. The broker process terminates following the throwing of a fatal exception.

**Recoverable**
These exceptions are generated for errors which, although not terminal in nature, mean that the processing of the current message flow has to be ended. Examples of recoverable exceptions are invalid data in the content of a message, or a failure to write a message to an output node. When a recoverable exception is thrown, the processing of the current message is aborted on that thread, but the thread recommences execution at its input node.

**Configuration**
Configuration exceptions are generated when a configuration request fails. This can be because of an error in the format of the configuration request, or an error in the data. When a configuration exception is thrown, the request is rejected and an error response message is returned.

**Parser** These exceptions are generated by message parsers for errors that prevent the parsing of the message content or creating a bit stream. A parser exception is treated as a recoverable exception by the broker.

**Conversion**

These exceptions are generated by the broker character conversion functions if invalid data is found when trying to convert to another data type. A conversion exception is treated as a recoverable exception by the broker.

**User** These exceptions are generated when a Throw node throws a user-defined exception.

**Database**

These exceptions are generated when a database management system reports an error during broker operation. A database exception is treated as a recoverable exception by the broker.

## Storage management in user-defined nodes

Consider issues that relate to storage management when you develop user-defined extensions in the C programming language.

If you are developing user-defined extensions using the Java programming language, you can use standard Java string handling methods.

All memory that is allocated by a user-defined extension must be released by the user-defined extension. The construction of a node at run time causes the cniCreateNodeContext function to be invoked, which allows the user-defined extension to allocate node instance specific data areas to store a context. The address of the context is returned to the broker, and is passed back from the broker when an internal method causes a user-defined extension function to be invoked; thus, the C user-defined extension can locate and use the correct context for the function processing.

The broker passes addresses of C++ objects to the user-defined extension, which are used as handles to be passed back on subsequent function calls. Your C user-defined extension must not manipulate or use these pointers in any way, for example, by trying to release storage using the free function. Such actions cause unpredictable behavior in the broker.

The cniCreateNodeContext implementation function is invoked whenever the underlying node object has been constructed internally. It is called when a broker is defined with a message flow that uses a user-defined node. This activity is not necessarily the same as creating (or reusing) a thread to execute a message flow instance that contains the node. The cniCreateNodeContext function is called only once, during the configuration of the message flow, regardless of how many threads are executing the message flow.

Similar considerations apply to user-defined parsers, and the corresponding implementation function cpiCreateContext.

## String handling in user-defined nodes

Consider issues that relate to string handling when you develop user-defined extensions in the C programming language.

If you are developing user-defined extensions using the Java programming language, you can use standard Java string handling methods.

To enable a broker to handle messages in all languages at the same time, text processing within the broker is done in UCS-2 Unicode. UCS-2 Unicode character strings are also used across the Java and C language user-defined extension APIs

to pass and return character data. Attributes are received in XML configuration messages as character strings, regardless of data type. If the true data type of an attribute is not a string, the cniSetAttribute function must perform the necessary verification and conversion before storing the attribute value. Similarly, when an attribute value is retrieved using cniGetAttribute2, conversion must be performed to a UCS-2 Unicode character string before returning the result.

`CciChar` defines a 16-bit character with UCS-2 Unicode representation. A `CciChar*` is a string of such characters terminated with a `CciChar` of 0. By default, a `CciChar` is represented by type `wchar_t`. However, some platforms do not have a convenient way of representing UCS-2 constants in source code, typically because of 4-byte `wchar_t` or EBCDIC representation. For example, a source-code constant such as L"ABC" expands to 12 bytes on Solaris.

For this reason, WebSphere Message Broker provides the utility functions cciMbsToUcs and cciUcsToMbs. Use these functions, where appropriate, to ensure portability of your user-defined nodes.

## Threading considerations for user-defined extensions

Message processing nodes and parsers must work in a multi-instance, multithreaded environment. Many node objects or parser objects are available, each with several syntax elements, and many threads can be executing methods on these objects.

An instance of a message flow processing node is shared and used by all the threads that service the message flow in which the node is defined. Parsers are invoked on the same thread as the nodes, therefore, if the flow is using multiple threads, the parsers are as well.

A user-defined extension must use this model. If a user-defined node requires global data or resources, you must protect the global data or resources by using semaphores to serialize access across threads. However, such serialization can result in performance bottlenecks. Avoid using global data and resources to create a more scalable solution.

The functions implemented by user-defined extensions must be reentrant, and any functions that they invoke must also be reentrant. All user-defined extension utility functions are fully reentrant.

Although a user-defined extension can create additional threads if required, all C utility functions and Java methods must be invoked on the same thread that called the cniEvaluate function in C or the evaluate method in Java, as appropriate for the language in which the node is written. If the same thread is not used, your code might compromise the integrity of the broker and cause unpredictable behavior. Any additional threads must not call the user-defined extension API. The API must only be used from the main thread that is invoked by the Broker.

For information about the cniEvaluate function see "cniEvaluate" on page 159.

## ODBC restrictions for user-defined nodes

The ODBC environment cannot be accessed using the Java or C language user-defined extension API.

Database access must be performed using the supplied processing nodes, or by using the following implementation functions supplied for that purpose:
- "cniSqlCreateStatement" on page 190

- "cniSqlExecute" on page 193
- "cniSqlSelect" on page 195
- "cniSqlDeleteStatement" on page 192

**Java Database Connectivity**

Types 2 and 4 JDBC drivers are supported, but are not provided with the broker.

## User-defined extensions in the runtime environment

Before you design and implement user-defined extensions, familiarize yourself with the core components. Ensure that you also understand the basic WebSphere Message Broker runtime architecture.

Ensure that you are familiar with the following runtime components and concepts:
- Runtime environment
- Broker domains
- Configuration Manager
- Brokers
- Execution groups
- "User-defined extensions execution model"

Also make sure that you understand the following concepts:
- Message flows overview

When you have gained an understanding of the runtime environment, read the following topics to help you understand how your user-defined extension interacts with the runtime components.
- "C user-defined input node life cycle" on page 13
- "Java user-defined input node life cycle" on page 14
- "C user-defined message processing nodes life cycle" on page 18
- "Java user-defined message processing nodes life cycle" on page 20
- "User-defined output node life cycle" on page 25
- "User-defined parser life cycle" on page 26

**User-defined extensions execution model:**

The execution model is the system used to start message flows through a series of nodes.

When an execution group is initialized, the appropriate loadable implementation library (LIL) files and Plug-in Archive (PAR) files are made available to the runtime environment. The execution group runtime process starts, and creates a dedicated configuration thread. You are responsible for ensuring that a user-defined node is thread-safe. If a node updates a variable across multiple threads, appropriate locking must be in place. Do not compromise this threading model in your implementation of user-defined nodes. Consider the following points:
- An input message sent to a message flow is processed only by the thread that received it.
- A single instance of a user-defined extension might be invoked on several threads concurrently.

- The message flow execution environment is conceptually like procedural programming. Nodes that you insert into a message flow are like subroutines called using a function call interface. However, rather than a call-return interface, in which parameters are passed in the form of input message data, the execution model is referred to as a propagation-and-return model.

As an example, consider a message flow in which you use both user-defined nodes and parsers. You use a user-defined node to process messages, and a user-defined parser to parse messages; both the node and parser contain implementation functions. The broker calls the implementation functions, or callback functions, when certain events occur:

- When an input message is received by the message flow and is propagated to the user-defined node:
  - For C nodes, the broker calls the cniEvaluate function for the user-defined node. See "cniEvaluate" on page 159.
  - For Java nodes, the broker calls the evaluate method that is implemented by the user-defined node.
- If the user-defined node wants to query the message to decide what to do with it, the node calls a C utility function or a Java method, as appropriate for the language in which the node is written.

The broker invokes the user-defined parser on one of its implementation functions, for example cpiParseFirstChild. This function instructs the parser to build the parse tree. The parser builds the tree by invoking utility functions that create elements in the parse tree, for example cpiCreateElement. The parser can be called many times by the broker.

## Node and parser factory behavior

The node factory and the parser factory assume roles in declaring a node to the broker or defining a parser.

Each loadable implementation library (LIL) has one node factory, or one parser factory, or has both. A node factory can identify many nodes, and a parser factory can identify many parsers.

When the broker loads the LIL, it calls the following functions:

- **`bipGetMessageflowNodeFactory`**

  After the operating system has loaded and initialized the LIL, the broker calls initialization function `bipGetMessageflowNodeFactory`. The `bipGetMessageflowNodeFactory` function calls the utility function `cniCreateNodeFactory`, which passes back a factory name (or group name) for all the nodes that your LIL supports.

- **`bipgetparserfactory`**

  After the operating system has loaded and initialized the LIL, the broker calls initialization function `bipgetparserfactory`. The `bipgetparserfactory` function defines the name of the factory that the user-defined parser supports, and the classes of objects, or shared object, that the factory supports. The initialization function `bipgetparserfactory` calls the utility function `cpiCreateParserFactory`, which passes back a factory name (or group name) for all the parsers that your LIL supports.

Before the node factory is returned, the broker calls the following functions:

1. **cniCreateNodeFactory**

   This function creates a single instance of the node factory in the broker.

2. **cndDefineNodeClass**

   This function defines the name of a node class that a node factory supports, and identifies the nodes that the node factory can create.

Before the parser factory is returned, the broker calls the following functions:

1. **cpiCreateParserFactory**

   This function creates a single instance of the named parser factory in the message broker.

2. **cpiDefineParserClass**

   This function defines the name of a parser class that a parser factory supports, and identifies the parsers that the factory can create.

See the following topics for information on these functions:

- "cniCreateNodeFactory" on page 143
- "cpiCreateParserFactory" on page 209
- "cniDefineNodeClass" on page 144
- "cpiDefineParserClass" on page 210

# Why use a user exit?

Use a user exit to intercept the progress of messages through message flows without having to redesign the message flow.

User exits provide a mechanism to apply actions (such as monitoring, message tracking, and auditing) operationally to deployed message flows at run time.

You can use user exits to call (by using callbacks) your custom C code, which is provided in a loadable exit library (LEL), at key points in message transactions in deployed message flows. These user exits can use utility functions from the user-defined extensions APIs to extract details of the broker, execution group, message flow, node, and message assembly. In addition, the user exits can use utility functions from the user-defined extensions APIs to modify parts of the message assembly.

To write user exits, you must be a skilled programmer with an understanding of WebSphere Message Broker and its architecture. Testing and debugging user exits can be time-consuming, and must be done in a safe environment. You must also maintain and service your own user exit.

Consider the following design factors when you plan and write a user exit:

- The effect on performance

  User exit callbacks are run inline with the current message transaction; that is, progress of the transaction is blocked until the return from the callback is received. Updating the message in a user exit callback can affect performance, particularly if the input message would not otherwise be changed in the message flow.

- Message parse timing

  On-demand parsing, referred to as partial parsing, is used to parse a message bit stream only as far as is necessary to satisfy the current reference in the message assembly. A user exit can navigate the message at each of its callback points, which can mean that the parse timing of the message flow is changed when you enable the user exit.

- Error handling

To ensure that the error handling that is provided by the designer of a message flow that is being intercepted by a user exit continues to operate as designed, you must program the user exit in the following way:

– All internal errors must be handled within the user exit, and the normal return from the callback must enable the message flow transaction to complete as normal.

– All exception condition that is encountered when the user exit calls utility functions in the user-defined extensions APIs must be returned to the flow for normal error processing. This behavior is achieved by calling cciRethrowLastException() to cut short the callback processing.

# Which type of user-defined extension to use

The types of user-defined extension that you can write include user-defined nodes, user-defined parsers, user-defined exits, and using a subflow as a user-defined node.

The following topics describe the different types of user-defined extension in more detail:

- "User-defined nodes"
- "User-defined parsers" on page 26
- "User exits" on page 30

## User-defined nodes

User-defined nodes are the main mechanism for extending the functions of WebSphere Message Broker.

The most common uses for a user-defined node are:

- Calling an external system for which WebSphere Message Broker does not provide nodes
- Calling already defined program libraries that perform a transformation or calculation that is required in the design of a message flow

Before you consider constructing a user-defined node, make sure that no built-in node is available to perform the required actions. For example, you might have considered creating a user-defined node to perform the following tasks, but you can use a JavaCompute node instead:

- Allowing programming languages other than ESQL to be used for coding message flow functions
- Performance advantages in performing some actions in compiled code
- Complex functions that are not available in ESQL, such as the large number of classes provided in JS2E

The following topics describe the different types of user-defined node in more detail:

- "User-defined input nodes"
- "User-defined message processing nodes" on page 17
- "User-defined output nodes" on page 25

**User-defined input nodes:**

A user-defined input node is an extension to the broker that provides a new input node in addition to the nodes supplied with the product.

You create user-defined input nodes using either the C or Java programming language, to provide message input to a message flow from a message queue when you want your broker to accept messages from a transport protocol other than WebSphere MQ.

You can use a user-defined input node to receive data from an external data source and to allow that data to be processed in a broker. In this way, you can complement the primitive input node types provided by WebSphere Message Broker.

You cannot use a user-defined input node to provide the In terminal to a message subflow. If you want to provide the In terminal to a subflow, you must use the supplied Input node.

Before writing a user-defined node, make sure that you are familiar with the concepts that are introduced in "Why use a user-defined extension?" on page 4 and "User-defined extensions in the runtime environment" on page 9.

*C user-defined input node life cycle:*

A user-defined input node that is written in the C programming language progresses through several stages during its lifetime.

The stages of the life cycle are:
- Registration
- Instantiation
- Processing
- Destruction

*Registration:*  During the registration phase, the broker discovers which resources are available and which LILs can provide them. In this instance, the resources available are nodes. The phase starts when an execution group starts. The LILs are loaded on the startup of an execution group, and the broker queries them to find out what resources they can provide.

A CciFactory structure is created during the registration phase, when the user-defined node calls cniCreateNodeFactory.

The following APIs are called by the broker during this stage:
- biGetMessageflowNodeFactory
- bipGetParserFactory

The following API is called by the user-defined node during this stage:
- cniCreateNodeFactory

*Instantiation:*  An instance of a user-defined input node is created when the mqsistart command starts or restarts the execution group process, or when a message flow that is associated with the node is deployed.

The following APIs are called during this phase:
- **cniCreateNodeContext**. This API allocates memory for the instantiation of the user-defined node to hold the values for configured attributes. This API is called once for each message flow that is using the user-defined Input node.

- **cniCreateInputTerminal**. This API is invoked within the cniCreateNodeContext API, and is used to tell the broker what input terminals, if any, your user-defined input node has.

  Your user-defined input node only has input terminals if it is also acting as a message processing node. If this is the case, it is typically better to use a separate user-defined message processing node to perform the message processing, rather than combine both operations in one, more complex, node.

- **cniCreateOutputTerminal**. This API is invoked within the cniCreateNodeContext API, and is used to tell the broker what output terminals your user-defined input node has.

- **cniSetAttribute**. This API is called by the broker to establish the values for the configured attributes of the user-defined node.

During this phase, a CciTerminal structure is created when cniCreateTerminal is called.

*Processing:* The processing phase begins when the cniRun function is called by the broker. The broker uses the cniRun function to determine how to process a message, including determining the domain in which a message is defined, and invoking the relevant parser for that domain.

A thread is demanded from the message flow's thread pool, and is started in the run method of the input node. The thread connects to the broker's queue manager, and retains this connection for its lifetime. When a thread has been allocated, the node enters a message processing loop while it waits to receive a message. It remains in the loop until a message is received. If the message flow is configured to use multiple threads, thread dispatching is activated.

The message data can now be propagated downstream.

The following APIs are called by the broker during this phase:
- cniRun. This function is called by the broker to determine how to process the input message.
- cniSetInputBuffer. This function provides an input buffer, or tells the broker where the input buffer is, and associates it with a message object.

*Destruction:* A user-defined input node is destroyed when the message flow is redeployed, or when the mqsistop command is used to stop the execution group process. You can destroy the node by implementing the cniDeleteNodeContext function.

When a user-defined input node is destroyed in one of these ways, you should free any memory used by the node, and release any held resources, such as sockets.

The following APIs are called by the broker during this phase:
- cniDeleteNodeContext. This function is called by the broker to destroy the instance of the input node.

*Java user-defined input node life cycle:*

A user-defined input node that is written in the Java programming language progresses through several stages during its lifetime.

The stages of the life cycle are:
- Registration

- Instantiation
- Processing
- Destruction

*Registration:* During the registration phase a user-defined input node written in Java makes itself known to the broker. The node is registered with the broker through the static getNodeName method. Whenever a broker starts, it loads all the relevant Java classes. The static method getNodeName is called at this point, and the broker registers the input node with the node name specified in the getNodeName method. If you do not specify a node name, the broker automatically creates a name for the node based on the package in which it is contained.

Using a static method here means that the method can be called by the broker before the node itself is instantiated.

*Instantiation:* A Java user-defined input node is instantiated when a broker deploys a message flow containing the user-defined input node. When the node is instantiated, the broker calls the constructor of the input node's class.

When a node is instantiated, any terminals that you have specified are created. A message processing node can have any number of input and output terminals associated with it. You must include the createInputTerminal and createOutputTerminal methods in your node constructor to declare these terminals.

To handle exceptions that are passed back to your input node, use createOutputTerminal to create a catch terminal for your input node. When the input node catches an error, the catch terminal processes it in the same way as an MQInput node would. You can allow most exceptions, such as exceptions that are caused by deployment problems, to pass back to the broker, and the broker will warn the user of any possible configuration errors.

As a minimum, your constructor class needs only to create these output terminals on your input node. However, if you need to initialize attribute values, such as defining the parser that will initially parse a message passed from the input node, you should also include that code at this point in your input node.

*Processing:* Message processing for an input node begins when the broker calls the run method. The run method creates the input message, and should contain the processing function for the input node.

The run method is defined in MbInputNodeInterface, which is the interface used in a user-defined node that defines it as an input node. You must include a run method in your node. If you do not include a run method in your user-defined input node, the node source code will not compile.

When a message flow containing a user-defined input node is deployed successfully, the broker calls the node's run implementation method, and continues to call this method while it waits for messages to process.

When a message flow starts, a single thread is dispatched by the broker, and is called into the input node's run method. If the dispatchThread() method is called, further threads can also be created in the same run method. These new threads immediately call into the input node's run method, and can be treated the same as the original thread. The number of new threads that can be created is defined by the additionalInstances property. Make sure that threads are dispatched after a

Developing user-defined extensions **15**

message has been created, and before it is propagated, to ensure that only one thread at a time is waiting for a new message.

The user-defined input node can choose a different threading model and is responsible for implementing the chosen model. If the input node supports the additionalInstances property, and dispatchThread() is called, the code must be fully re-entrant, and any functions that are invoked by the node should also be re-entrant. If the input node forces single threading, that is, it does not call dispatchThread(), the node documentation must state that setting the additionalInstances property has no effect on the input node.

For more information on the threading model for user-defined input nodes, see "Threading considerations for user-defined extensions" on page 8.

*Destruction:* A Java user-defined input node is destroyed when the node is deleted or the broker is shut down. You do not need to include anything in your code that specifies the node should be physically deleted, because this can be handled by the garbage collector.

However, if you want notification that a node is about to be deleted, you can use the onDelete method. You might want to do this if there are resources that you want to delete, other than those that will be garbage collected. For example, if you have opened a socket, this will not be properly closed when the node is automatically deleted. You can include this instruction in your onDelete method to ensure that the socket is closed properly.

*Planning user-defined input nodes:*

Before you develop a user-defined input node, plan and design its content and purpose.

*Analysis:* Before you develop a user-defined input node, ask yourself the following questions:
- Do you need to create a custom input node?

  You must include at least one input node in a message flow. Which one you choose depends on the source of the input messages:
  - If the messages arrive at the broker on a WebSphere MQ queue, use the MQInput or MQOptimizedFlow node.
  - If SOAP messages are received over HTTP, use the SOAPInput node.
  - If other messages are received over HTTP, use the HTTPInput node.
  - If the messages are received from a multicast application, use the Real-timeInput node.
  - If the messages are received from a JMS source, use either the Real-timeInput node or the JMSInput node.
  - If the messages are sent by telemetry (SCADA) devices, use the SCADAInput node.
  - If the messages are received from an EIS, use the PeopleSoftInput, SAPInput, or SiebelInput node.
  - If the messages are retrieved from files, use the FileInput node.
  - If the message source is any other, you must use a user-defined input node.

  For information about using more than one input node in a message flow, see Using more than one input node.

- To successfully input the data concerned, does the input node have to interface with vendor software? If so, does the API that enables access to this software break your threading model?
- Do you need a new user-defined parser to interpret the body (payload) of the message generated by this input node, or can it be parsed by a standard built-in parser?
- Do you need the user-defined input node to operate the message flow instance in which it resides under transactional control as a globally-coordinated transaction?
- Do you need the new user-defined input node to offer configuration options?
- Do you need messages propagated by this input node to be processed by the following primitives?
  - All primitive output nodes
  - ResetContentDescriptor nodes

*Design considerations:* Before developing and implementing your input node, decide on the following factors:
- Which message parser will initially parse the input message.
- Whether to override the default message parser attribute values for this input node.
- Which threading model is appropriate for the input node.
- What kind of message processing and transaction support will the node support.
- Which configuration attributes required by the input node should be externalized for alteration by the message flow designer.
- What optional node APIs will the user-defined node provide.
- How you will handle general development issues:
  - "Threading considerations for user-defined extensions" on page 8
  - "Storage management in user-defined nodes" on page 7
  - "String handling in user-defined nodes" on page 7
  - "Errors and exception handling" on page 5
  - Expected message formats for primitive nodes that expect specific header folders.

When you design nodes to be used as extensions to WebSphere Event Broker, the following restrictions apply:
- User-defined input nodes can support only XML, BLOB, and the WebSphere MQ parsers, because the MRM parser is not shipped with WebSphere Event Broker and user-defined parsers are not supported.
- User-defined nodes must not allow users to evaluate user ESQL code, because the use of ESQL in WebSphere Event Broker is not supported. For example, nodes that expose the input to MbSQLStatement as a node attribute are effectively emulating a Compute node.

**User-defined message processing nodes:**

A user-defined message processing node is a node that you can create to complement the supplied built-in node types.

You might consider the use of a user-defined message processing node in the following situations:

- Your messages need transformations that the built-in nodes do not provide. For example, you might need a currency converter node.
- You want to write messages into a flat file on the local system for later processing by another application or utility program.

Combine your user-defined nodes with the built-in nodes to create message flows that meet your exact business requirements.

*C user-defined message processing nodes life cycle:*

A user-defined message processing node for the C programming language goes through various stages.

This topic covers the objects that are created and destroyed, and the implementation functions and classes that are called in the following stages:
- Registration
- Instantiation
- Processing
- Destruction

The information in this topic applies to both output nodes and message processing nodes. Both of these node types can be considered together, because although a message processing node is typically used to process a message, and an output node is used to provide an output in the form of a bit stream, you can use either type of node to perform either of these functions.

*Registration:* A user-defined message processing node is registered with the broker when the LIL that contains the node has been loaded and initialized by the operating system.

The broker calls bipGetMessageflowNodeFactory to establish the function of the LIL, and how the LIL should be called.

The bipGetMessageflowNodeFactory function in turn calls the cniCreateNodeFactory function, which returns a factory or group name for all of the nodes that are supported by your LIL.

The LIL should then call the utility function cniDefineNodeClass to pass both the name of each node and a virtual function table of the function pointers of the implementation functions.

*Instantiation:* During the instantiation phase, an instance of a user-defined message processing node is created. The phase starts when the broker creates a message flow and calls the cniCreateNodeContext function for each instantiation of the user-defined node in that message flow. The cniCreateNodeContext function is that which is specified in the iFpCreateNodeContext field of the CNI_VFT struct passed to cniDefineNodeClass for that node type. This function should allocate the resources required for that node, including memory such that the instantiation of the user-defined node can hold the values for the configured attributes.

The broker will create a node instance and call cniCreateNodeContext on the following occasions:
- Message flow is created:

- Broker is being started (user has run mqsistart). Any message flows previously deployed are re-created when the broker starts.
- Execution group is being reloaded (user has run mqsireload). Any message flows that have been deployed previously are re-created when the execution group reloads.
- A severe error has occurred within the execution group which results in the execution group being restarted.
- Message flow is redeployed. When a message flow is changed and redeployed, the broker processes redeploy by deleting all nodes in the flow and then re-creating them with the new configuration.

**Note:** A message flow is not created when starting an execution group. Stopping an execution group simply stops all flows and does not delete the flow or bring the process down. Restarting an execution group, starts the message flows but does not re-create the message flows.

Within cniCreateContext, the user-defined extension calls the two functions cniCreateInputTerminal and cniCreateOutputTerminal in order to establish what input and output terminals the message processing node has.

*Processing:* During the processing phase of the life cycle of a user-defined message processing node, the message is transformed in some way, when some processing operation takes place on the input message.

When the broker retrieves a message from the queue and that message arrives at the input terminal of your user-defined node, the broker calls the implementation function cniEvaluate. This function is used to decide what to do with the message.

You can use a range of node utility functions in your user-defined message processing node to perform a range of message processing functions, such as accessing the message data, accessing ESQL, transforming a message object, and propagating a message. You should include the node utility functions you are going to use to process the message within the cniEvaluate function.

This interface does not automatically generate a properties subtree for a message. It is not a requirement for a message to have a properties subtree, although you might find it useful to create one to provide a consistent message tree structure regardless of input node. If you want a properties subtree to be created in a message, and you are also using a user-defined input node, you must do this yourself

*Destruction:* When a user-defined message processing node has processed a message, you should ensure that it is destroyed, to release any system resources that it used, and to release any data areas specific to the node instance, such as context, that were acquired when the message was constructed or processed.

An instance of a user-defined message processing node is destroyed when the broker calls the cniDeleteNodeContext function.

The broker calls cniDeleteNodeContext when the instance of the node is deleted. The following events can cause a node to be deleted:
- Controlled termination of the execution group process:
  - Broker is being stopped (user has run mqsistop)
  - Execution group is being reloaded (user has run mqsireload)

– A severe error has occurred within the execution group, which results in the execution group being restarted.

**Note:** This does NOT include stopping an execution group. Stopping an execution group simply stops all flows, and does not delete the flow or bring the process down.

- Message flow is deleted. For example, a message flow is deleted from the tooling's Broker Administration perspective.
- Message flow is redeployed. When a message flow is changed and redeployed, the broker processes redeploy by deleting all nodes in the flow and then re-creating them with the new configuration.

*Java user-defined message processing nodes life cycle:*

During the lifecycle of the user-defined nodes that you create in Java, objects are created and destroyed, and different methods and classes called.

Every node goes through the following stages:
- Registration
- Instantiation
- Processing
- Destruction

The information here applies to both output nodes and message processing nodes. Both of these node types can be considered together, because although a message processing node is typically used to process a message, and an output node is used to provide an output, in the form of a bit stream, from a message, you can use both types of node to perform either of these functions.

*Registration:* The registration phase occurs when a user-defined message processing node that is written in Java contacts the broker, or registers with the broker.

Whenever a broker starts, it loads all relevant LIL files and Java classes. To ensure that a message processing node is registered with the broker, you must provide the broker with a class that implements the MbNodeInterface interface and is contained in the classpath used by the broker.

*Instantiation:* A Java user-defined message processing node is instantiated when a broker deploys a message flow that contains the user-defined message processing node. When the node is instantiated, the constructor of the message processing node class is called.

When a node is instantiated, all terminals that you have specified are created. A message processing node can have an unlimited number of input and output terminals associated with it. You must include the createInputTerminal and createOutputTerminal methods in your node constructor to declare these terminals.

Output terminals include out, failure, and catch terminals. Use the createOutputTerminal class within the node class constructor in order to create as many output terminals as you require.

As a minimum, you must create only these output terminals by using your constructor class. However, if you need to initialize attribute values, you must also include that code at this point in your message processing node.

If you want to handle exceptions that are passed back to your message processing node, it is good practice to do this by creating a failure terminal for your user-defined message processing node, by using the createOutputTerminal method. It is sensible to use the failure terminal for this process because that is the terminal to which errors are propagated.

Make sure that all exceptions that are caught by the message processing node are dealt with properly. If you do not include a failure terminal, the message processing node does not attempt to handle the exception. If your message flow does not contain a method of exception handling, all exceptions thrown are passed back to the input node, where the input node deals with the exceptions.

If you do catch exceptions, make sure that you rethrow all exceptions that the message processing node cannot deal with. This action causes the exception to be passed back to the input node for handling; for example, when you want to rollback a transaction.

*Processing:* During the processing phase of the life cycle of a user-defined message processing node, the message processing node takes the logical hierarchy of the message and processes it in some way.

*Destruction:* A Java user-defined message processing node is destroyed when the node is deleted, or the broker is shut down. You do not have to include anything in your code to specify that the node is physically deleted, because this process can be handled by the garbage collector.

However, if you want notification that a node is about to be deleted, you can use the onDelete method. You might want to receive notification if the node has resources that you want to delete, other than those that will be garbage collected. For example, if you have opened a socket, it is not properly closed when the node is automatically deleted. You can include this instruction in your onDelete method to ensure that the socket is closed properly.

*Planning user-defined message processing nodes:*

Plan how to write your message processing node or output node, and how to navigate the message within the node.

*Design factors:* Before developing and implementing your message processing node, consider the following points:
- Which parser will parse messages.
- Whether to override the default message parser attribute values for this message processing node.
- What is the appropriate threading model for the message processing node.
- How to implement the end of message processing and transaction support that the node must support.
- What configuration properties required by the message processing node must be externalized for alteration by the message flow designer.
- What optional node APIs will the user-defined node provide.
- General development issues:
  – "Threading considerations for user-defined extensions" on page 8
  – "Storage management in user-defined nodes" on page 7
  – "String handling in user-defined nodes" on page 7
  – "Errors and exception handling" on page 5

– Expected message formats for built-in nodes that expect specific header folders, see Element definitions for message parsers

*Syntax element navigation:*   The broker provides functions that your node can call to traverse the tree representation of the message, as well as functions and methods that support navigation from the current element to other elements:
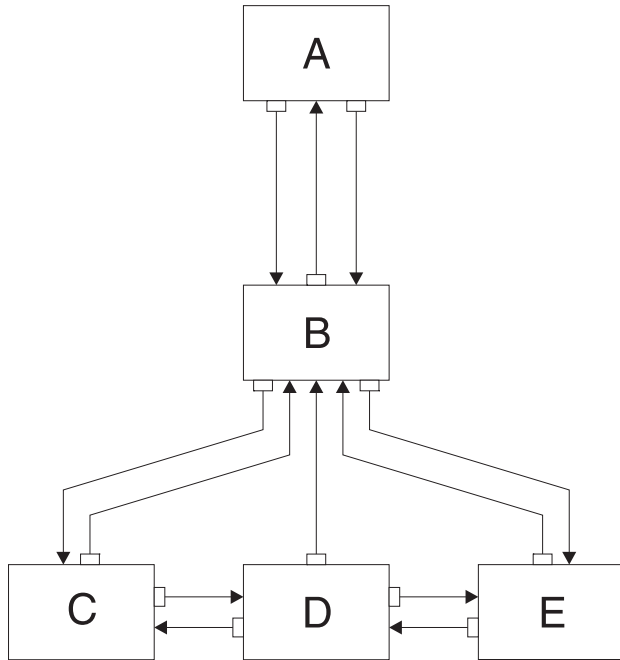
- Parent
- First child
- Last child
- Previous (or left) sibling
- Next (or right) sibling

These relationships are shown in the following diagram.
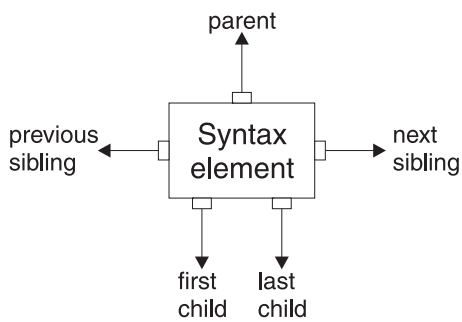


Other functions and methods support the manipulation of the elements themselves, with functions and methods to create elements, to set or query their values, to insert new elements into the tree, and to remove elements from the tree. See "C node utility functions" on page 125 and "C parser utility functions" on page 198, or the Javadoc information for more details.

The following diagram describes a simple syntax element tree that shows a full range of interconnections between the elements.

The element **A** is the root element of the tree. It has no parent because it is the root. It has a first child of element **B**. Because **A** has no other children, element **B** is also the last child of **A**.

Element **B** has three children: elements **C**, **D**, and **E**. Element **C** is the first child of **B**; element **E** is the last child of **B**.

Element **C** has two siblings: elements **D** and **E**. The next sibling of element **C** is element **D**. The next sibling of element **D** is element **E**. The previous sibling of element **E** is element **D**. The previous sibling of element **D** is element **C**.

The following diagram shows the first generation of syntax elements of a typical WebSphere MQ message received by a broker. (Not all messages have an MQRFH2 header.)
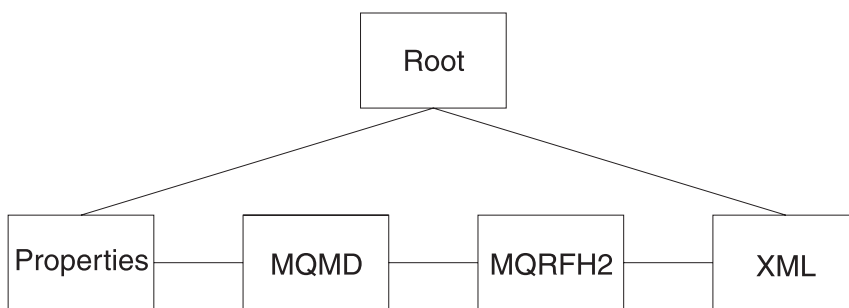


These elements at the first generation are often referred to as folders, in which syntax elements that represent message headers and message content data are stored. In this example, the first child of root is the Properties folder. The next sibling of Properties is the folder for the MQMD header. The next sibling is the folder for the MQRFH2 header. The last folder represents the message content, which (in this example) is an XML message.

The previous figure includes an MQMD and an MQRFH2 header. All messages that are received by a processing node that handles WebSphere MQ include an MQMD header; a number of other headers can also be included.

*Navigating an XML message:*   Consider the following XML message:

```
<Business>
  <Product type='messaging'></Product>
  <Company>
    <Title>IBM</Title>
    <Location>Hursley</Location>
    <Department>WebSphere MQ</Department>
  </Company>
</Business>
```

In this example, the elements are of the following types:

**Name element**
> Business, Product, Company, Title, Location, Department

**Value element**
> IBM, Hursley, WebSphere MQ

**Name-value element**
> type='messaging'

Use supplied node utility functions and methods (or the similar parser utility functions) to navigate through a message. Using the XML message shown, you must call cniRootElement first, with the message received by the node as input to this function. In Java, you must call getRootElement on the incoming MbMessage object. This call returns an MbElement that represents the root of the element. Do not modify this root element in the user-defined node.

The figure of the first generation of the syntax elements of a typical message that is received by the broker, shows that the last child of the root element is the folder containing the XML parse tree. Navigate to this folder by calling cniLastChild (with the output of the previous call as input to this function) in a C node, or by calling the method getLastChild on the root element, in a Java node.

Only one element (`<Business>`) is at the top level of the message, therefore call cniFirstChild (in C) or getFirstChild (in Java) to move to this point in the tree. Use cniElementType or getType to get its type (which is `name`), followed by cniElementName or getName to return the name itself (`Business`).

The element `<Business>` has two children, `<Product>` and `<Company>`. Use cniFirstChild or getFirstChild followed by cniNextSibling or getNextSibling to navigate to each child in turn.

The element `<Product>` has an attribute (`type='messaging'`), which is a child element. Use cniFirstChild or getFirstChild to navigate to this element, and cniElementType or getType to return its type (which is `name-value`). Use cniElementName or getName to get the name. To get the value, call cniElementValueType to return the type, followed by the appropriate function in the cniElementValue group, in this example it is cniElementCharacterValue. In Java use the method getValue, which returns a Java object representing the element value.

The element `<Company>` has three children, each one having a child that is a value element (`IBM`, `Hursley`, and `WebSphere MQ`). Use the functions already described to navigate to them and access their values.

Other functions are available to copy the element tree (or part of it). The copy can then be modified by adding or removing elements, and changing their names and values, to create an output message. See "C node utility functions" on page 125 and "C parser utility functions" on page 198, or the Java user-defined node API, for more information.

**User-defined output nodes:**

A user-defined output node is an extension to the broker that provides a new message flow output node in addition to the nodes supplied with the product.

If you want your message flow to send messages by using a protocol that is not supported by WebSphere Message Broker, you can create your own output node.

WebSphere Message Broker provides the following output nodes:
- MQOutput - delivers an output message from a message flow to a WebSphere MQ queue
- MQReply - sends a response to the originator of the input message.
- SCADAOutput - sends a message to a client connecting using the MQIsdp protocol
- Publication - filters output messages from a message flow and transmit them to subscribers who have registered an interest in a particular set of topics.
- JMSOutput - sends a message to a JMS destination
- EmailOutput - sends an e-mail message to one or more recipients
- FileOutput - writes a message to a file

If the target application expects to receive message in any other way, you must use a user-defined output node.

User-defined output nodes can be considered together with user-defined message processing nodes. Conceptually, these two kinds of user-defined nodes are the same. Although a message processing node is typically used to process a message, and an output node is used to provide an output, in the form of a bit stream, from a message, you construct output nodes and message processing nodes in a similar way, and you can use either type of node to perform either function.

For more information about user-defined output nodes, read the topics that cover user-defined message processing nodes, see "User-defined message processing nodes" on page 17.

*User-defined output node life cycle:*

The life cycle of a user-defined output node follows the same pattern as the life cycle of user-defined message processing nodes.

The following topics describe the life cycle of user-defined message processing nodes; read the topic that corresponds to your type of output node:
- "C user-defined message processing nodes life cycle" on page 18
- "Java user-defined message processing nodes life cycle" on page 20

Although a message processing node is typically used to process a message, and an output node is used to provide an output in the form of a bit stream, you can use both types of node to perform either of these functions.

*Planning user-defined output nodes:*

A user-defined output node generates an output bit stream from a message tree.

Optionally, you can connect the node to another node and propagate the message tree for further processing. User-defined output nodes and message processing nodes are, therefore, structured in the same way. All relevant information for output nodes is included in "Planning user-defined message processing nodes" on page 21.

## User-defined parsers

A user-defined parser is a program that interprets the bit stream of an incoming message and creates an internal representation of the message in a tree structure. A user-defined parser can also regenerate a bit stream for an outgoing message from the internal message tree representation

Create a user-defined parser when the WebSphere Message Broker parsers are not sufficient to parse user-defined messages.

Do not use user-defined parsers to provide connectivity or transformation functions. In most cases, the MRM or other IBM® supplied parsers are capable of passing most standard type of format. You can also parse a message and construct a message tree in a user-defined node without the need to write a parser. For example, a user-defined node that reads e-mails from a POP3 server can parse the e-mail and construct a message tree without the need to write a user-defined parser.

If the parser is going to be used only in a user-defined node, you do not need to use a user-defined parser. However, consider a user-defined parser if the parser will be called from other message flow nodes.

**User-defined parser life cycle:**

Various stages exist in the life of a user-defined message flow parser.

These stages are involved:
• Registration
• Instantiation
• Processing
• Destruction

This topic describes the interactions that take place between WebSphere Message Broker components when you run a user-defined parser. It explains each stage in terms of the events that start each stage, and the events that occur during and after each stage, and the APIs that are called. Understanding the concepts here help you to design and develop your parser more effectively.

*Registration:* The first phase in the user-defined parser's life cycle is the registration phase. The purpose of the registration phase is to register the user-defined parser with the broker. This phase starts when the execution group starts.

*Instantiation:* The parser is created during the instantiation phase of the parser life cycle. When an input message is received, or an output message is built in a Compute node, the relevant parser is identified, and parser requirements are taken from the message header, such as the MQMD. The broker starts and loads the

Loadable Implementation Library (LIL) and the parser factory. Before the cpiCreateContext function is called, the broker creates a name element as the effective root element for the parser. However, this element is not named. The parser should name this element in the cpiSetElementName function. The execution group process creates an instance of the parser, and the broker makes a call to cpiCreateContext to allow the parser object to acquire the appropriate section of the message.

The broker then makes a call to cpiParseBuffer. cpiParseBuffer performs any necessary initialization, and returns the length of the message content that the parser is taking ownership of. The parser assesses how much of the message data to parse, and claims the appropriate number of bytes.

Whenever an instance of a user-defined parser object is created, the context creation implementation function cpiCreateContext is also invoked by the broker. This call allows the parser to allocate instance data associated with the parser. A cpiDeleteContext function to delete the context of the parser object is also required.

*Processing:*  During the processing phase, the parser manipulates, alters, and references elements within the message object. The message flow processing phase begins when any message processing activity occurs, such as navigation, that requires access to an element within a message that does not exist in the broker's internal model representation of the message concerned.

During the message flow processing phase, the parser is invoked in response to attempts to navigate into the message tree. The parser examines the buffer that was allocated when cpiParseBuffer was called, and creates any necessary message elements.

The parser can then navigate through the message elements, using any or all of the following parser implementation functions:
- cpiParseFirstChild
- cpiParseLastChild
- cpiParsePreviousSibling
- cpiParseNextSibling

These functions are invoked when any form of navigation is made (such as a filter expression that specifies a message field) into the part of the syntax element tree that logically represents the data for a message format supported by a user-defined parser. This navigation occurs when an operation within the broker requires a syntax element tree to be built or extended.

Consider the following points when deciding how best to navigate the syntax element tree:
- A Syntax element has five pointers to its parents, siblings, and first and last children, so that a finite set of navigations is available.
- The same internal classes are used to perform all of these navigations.
- The parser does not control the navigation. The ESQL or a user-defined node makes the decision about the direction in which to navigate, and the order in which the navigational parser implementation functions are invoked. The user-defined parser has no control over the direction and order, and needs to respond correctly to the chosen navigation scheme; for example, parsing right to left, as well as left to right.

- When writing a user-defined parser, place the parser code in a parseNextItem function. This function should build the syntax element tree one element at a time, setting names, values and complete flags appropriately. How you implement this function depends on the nature of the bit stream to be parsed. The supplied sample parser demonstrates this behavior.

When the parser has finished parsing the relevant parts of the syntax element tree, it calls cpiWriteBuffer. This function appends its portion of the syntax element tree to the bit stream in the message buffer that is associated with the parser object, and creates the output message.

*Destruction:*   The Destruction phase is the final phase in the user-defined parser life cycle. When the parser has written its portion of the syntax element tree to the bit stream and created the output message, the system resources that were created by the broker for the parser to use need to be released.

The destruction phase begins when the mqsistop command is used to stop the execution process.

**Planning user-defined parsers:**

Read about the concepts that you should consider before you develop a user-defined parser.

When you have considered the information provided here, and are ready to develop your own parser, use the instructions in "Implementing a user-defined parser" on page 86 to construct your parser.

*Analysis:*   Before you start to create your own parser, be clear about its purpose. You can perform most tasks using the functions that are provided with WebSphere Message Broker, so you might not need to create a user-defined parser for your particular task.

Before you construct and implement a user-defined parser, consider the following questions:
- Do you need to create a user-defined parser?

  If the available parsers in WebSphere Message Broker are not appropriate for your needs, define your own parser to parse internal, customer-specific, or generic commercial message formats.
- Does WebSphere Message Broker already provide a parser for the domain or message header?

  See Parsers for details of message domains for which the supplied parsers can accept input messages, and message headers with which the supplied parsers can work.
- Does the syntax of the in-house or commercial message dictate a format that can be parsed?
- To parse the message successfully, does the parser need to interact with vendor software? If so, does the API that enables access to this software break your threading model?
- Do you need to process multi-part, multi-format messages?

  WebSphere Message Broker does not support multi-part, multi-format messages. A multi-part MRM message must consist of messages that are all in the same format.
- What type of parsing strategy will provide best performance?

WebSphere Message Broker supports partial parsing, which allows your parser to parse only relevant fields in a message. Using partial parsing can save system resources.

*Partial and full parsing:* WebSphere Message Broker supports *partial parsing*. If an individual message contains hundreds or even thousands of individual fields, the parsing operation requires considerable memory and processor resources to complete. An individual message flow might reference only a few of these fields, or none at all, so it is inefficient to parse every input message completely. For this reason, WebSphere Message Broker allows parsing of messages on an as-needed basis. (This ability does not prevent a parser from processing the entire message in one step, and some parsers are written to process the entire message in this way.)

Each syntax element in a logical message has two bits that indicate whether all the elements on either side of an element are complete, and whether its children are complete. Parsing is typically completed in a bottom-to-top, left-to-right manner. When a parser has parsed the siblings of a particular element that precede the given element and the first child, it sets the first completion bit to one. Similarly, when the pointer to the next sibling of an element is complete, as well as its last child pointer, the other completion bit is set to one.

In partial parsing, the broker waits until a part of the message is referenced, and invokes the parser to parse that part of the message. Message processing nodes refer to fields within a message using hierarchical names. The name begins at the root of the message and proceeds down the message tree until the particular element is located. If an element is encountered without its completion bits set, and further navigation from this element is required, the appropriate parser entry point is called to parse the necessary part of the message. The relevant part of the message is parsed, appropriate elements are added to the logical message tree, and the element in question is marked as complete.

If you do not need to parse the full bit stream, you can use partial parsing. During partial parsing, a parser is called recursively until the requested element is returned, or until the message tree has been marked as complete, and the requested element is known not to exist.

Whether you choose to perform a full or partial parse depends on how the message will be processed. If most field elements within the message are likely to be accessed during processing, performing a full parse of the message when an attempt is made to access it is typically more efficient, particularly for smaller messages.

However, if most field elements within the message are not likely to be accessed during processing, performing a partial parse of the message when an attempt is made to access a specific field is typically more efficient, particularly when the message size grows.

**Specific types used by parsers:**

Specific types are used when a parser needs additional information that is associated with some or all the elements in a tree in order to generate the bit stream.

For the XML parser, the specific type information is used to mark special elements such as components, processing instructions, and CDATA sections. The methods

getSpecificType and setSpecificType are used by user-defined nodes to query this information and to generate message trees that use these special types.

Developers of user-defined parsers can generate their own specific type values to control special handling characteristics in their parser code using the existing C user-defined parser interface. The getSpecificType and setSpecificType methods enable Java user-defined nodes to fully use this parser capability.

## User exits

A user exit is user-provided custom software, written in C, to track data passing through message flows.

User-provided functions can be invoked at specific points during the life cycle of a message while it passes through the message flow, and can invoke utility functions to query information about the point in the flow, and the contents of the message assembly. The utility function can also modify certain parts of the message assembly. For more information about using user exits, see "Why use a user exit?" on page 11.

The user exits can be invoked when one or more of the following events occur:
- The end of a unit-of-work (UOW) or transaction (COMMIT or ROLLBACK).
- A message passes between two nodes.
- A message is successfully enqueued or sent to a transport in an output, reply, or request node.
- A message is dequeued or received in an input, response, or TimeoutNotification node.



MQInput　　　　　　Compute　　　　　MQOutput
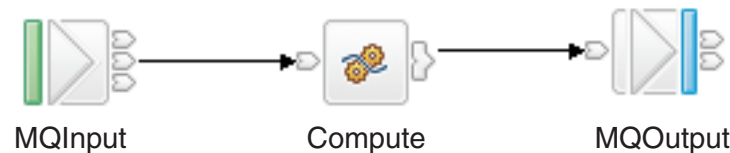
In the basic message flow shown here, you can track messages at three levels:
- Transaction level
- Node level
- Input or output level

At the transaction level, you can track the following events:
- Messages being read into the flow
- Completion of the transaction

At the node level, you can track the following events:
- A message passing from one node to another
- Completion of processing for one node

At the message input or output level, you can track the following events:
- Messages being read into the flow
- Messages being written from the flow

Therefore, you can track five different types of event, which occur in the following sequence:
1. A message is dequeued from the input source (read into the flow).

2. A message is propagated to the node for processing.
3. A request message is sent to the output node's transport, and transport-specific destination information is written to "WrittenDestination" in the LocalEnvironment.
4. Node processing is completed.
5. The transaction ends.

## Which language to use to implement a user-defined extension

You can use Java or C to implement a user-defined extension.

You can use C to implement all types of user-defined extension, but you can use Java to implement only user-defined nodes.

If you can, use Java for user-defined nodes, and use C for everything else.

You must compile user-defined nodes, parsers, and exits that are written in C into a loadable implementation library (LIL): that is, a shared library on Linux and UNIX systems, or a dynamic link library (DLL) on Windows systems. You must package user-defined nodes that are written in Java as a JAR file.

To achieve platform independence, use the ANSI standard C or Java programming languages, and avoid platform-specific code in your user-defined extension.

## Implementing the supplied user-defined extension samples

WebSphere Message Broker provides some sample code to help you understand how to write user-defined nodes and parsers.

The samples consist of a sample parser, and the following sample nodes:

| Switch | A node, implemented in both C and Java versions, that propagates an input message to one of several output terminals depending on the message content. |
|---|---|
| Transform | A node, implemented in both C and Java versions, that performs a simple message transformation. |

Each sample node consists of the source files and some files that you can use to test each node. For the sample parser there are only source files. See "Sample node files" on page 121 and "Sample parser files" on page 123 for details of the sample files and where to find them.

To implement the supplied samples:
1. Compile the samples. For information on how to compile a Java node, see "Compiling a Java user-defined node" on page 74. For information on how to compile a C node or parser, see "Compiling a C user-defined extension" on page 52.
2. Install the user-defined extension in a broker domain. For instructions on completing this step, see "Installing user-defined extension runtime files on a broker" on page 106.
3. On the computer that hosts the workbench, extract the `SampleNodesProject.zip` file, which is located in the samples subdirectory and copy the files to a directory from which the workbench can access them. For example, on Windows, the location is *install_dir*\sample\extensions\

com.ibm.samples.nodes directory. For further information about where to copy the files, see "Installing a user-defined extension to current and past versions of the broker" on page 108.

4. Open the workbench and switch to the Broker Application Development perspective. The category called "Sample nodes" is now visible in the palette, and the sample nodes are shown below it. Documentation about the sample nodes is also visible in the help system under "Samples".

5. Include the sample nodes in a message flow (see Adding a message flow node).

6. Deploy the message flow; see Deploying.

7. For the Switch and Transform nodes, you can put a message to the input queue of the message flow and observe the results, as follows:

   a. Make sure that the message flow containing the sample node is deployed successfully; see Checking the results of deployment.

   b. Use the Enqueue message function to put the sample input messages (the .xml files listed above) to the input queue named on the input node of the message flow; see Debug: putting a test message on an input queue.

   You can also use a Trace node or the Flow debugger to see what is happening in your message flow.

## Implementing user-defined extensions

Create the resources for your user-defined extension. You can write user-defined nodes in C or Java. You can write user-defined parsers and exits only in C.

**Before you start:**

Read the following topics:
- "User-defined extensions overview" on page 3
- "Which type of user-defined extension to use" on page 12

To create a user-defined extension, follow the instructions in the appropriate topic:
- "Implementing a user-defined node" on page 33
- "Implementing a user-defined parser" on page 86
- "Implementing a user-defined exit" on page 99

For user-defined nodes only, you must create a workbench Eclipse plug-in as well as the run time .lil or .jar file. The workbench plug-in adds the user-defined node to the node palette in the Message Flow editor, so that you can include the new node in message flows. This additional task is described in "Creating the user interface representation of a user-defined node in the workbench" on page 76. This step is not required for user-defined parsers or exits.

The following table shows the tasks required for creating the different types of user-defined extension.

| Action | Topics to view |
|---|---|
| To use one of the Java sample nodes | 1. "Compiling a Java user-defined node" on page 74<br>2. "Installing user-defined extension runtime files on a broker" on page 106<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 76<br>4. "Testing a user-defined node" on page 84 |

| Action | Topics to view |
|---|---|
| To use one of the C sample nodes | 1. "Compiling a C user-defined extension" on page 52<br>2. "Installing user-defined extension runtime files on a broker" on page 106<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 76<br>4. "Testing a user-defined node" on page 84 |
| To use the sample parser | 1. "Compiling a C user-defined extension" on page 52<br>2. "Installing user-defined extension runtime files on a broker" on page 106 |
| To create your own Java node by using the workbench | 1. "Creating an input node in Java" on page 60 or "Creating a message processing or output node in Java" on page 65<br>2. "Using error logging from a user-defined extension" on page 115<br>3. "Compiling a Java user-defined node" on page 74<br>4. "Testing a user-defined node" on page 84<br>5. "Packaging a user-defined node workbench project" on page 107<br>6. "Installing a user-defined extension to current and past versions of the broker" on page 108 |
| To create your own C node | 1. "Creating an input node in C" on page 36 or "Creating a message processing or output node in C" on page 44<br>2. "Using error logging from a user-defined extension" on page 115<br>3. "Compiling a C user-defined extension" on page 52<br>4. "Installing user-defined extension runtime files on a broker" on page 106<br>5. "Creating the user interface representation of a user-defined node in the workbench" on page 76<br>6. "Testing a user-defined node" on page 84<br>7. "Packaging a user-defined node workbench project" on page 107<br>8. "Installing a user-defined extension to current and past versions of the broker" on page 108 |
| To create your own parser | 1. "Implementing a user-defined parser" on page 86<br>2. "Using error logging from a user-defined extension" on page 115<br>3. "Compiling a C user-defined extension" on page 52<br>4. "Installing user-defined extension runtime files on a broker" on page 106 |
| To create a user exit | 1. "Developing a user exit" on page 99<br>2. "Compiling a C user-defined extension" on page 52<br>3. "Deploying a user exit" on page 100 |

## Implementing a user-defined node

You can implement a user-defined node to extend the function of WebSphere Message Broker.

**Before you start:**

Read the following topics:
- "User-defined extensions overview" on page 3
- "Why use a user-defined extension?" on page 4
- "User-defined nodes" on page 12
- "Which type of user-defined extension to use" on page 12

Consider the following restrictions and factors when developing user-defined nodes:
- Interfacing a C user-defined node to Java and providing a JNI wrapper is not supported. This restriction exists because the broker internally initializes a JVM, which is unavailable through the user-defined extension interface. The JVM initializes with various parameters that are specific to the requirements of the broker. Because only one JVM exists in a process, whoever initializes it first specifies these parameters. If a user-defined node uses Java, and the broker is initialized first, these parameters might not be suitable for the user-defined node. If the user-defined node creates the JVM before the broker starts, the broker might not function correctly.
- User-defined input nodes can support only XML, BLOB, and the WebSphere MQ parsers.
- Avoid using functions that are specific to an operating system. If you code in this way, your user-defined extensions can work on various operating systems without requiring changes to the source code.

To implement a user-defined node, complete the following tasks in the specified order:
1. "Designing a user-defined node"
2. "Creating a user-defined node" on page 35
3. "Packaging and distributing user-defined extensions" on page 101
4. "Testing a user-defined node" on page 84
5. "Packaging a user-defined node workbench project" on page 107

## Designing a user-defined node

Decide what type of node you need to implement the functions that are required by your application.

**Before you start:**

Read Deciding which nodes to use to understand the different types of node. You might need more than one node to implement all the functions that you require.

The functions that you require might not be satisfied by a template that already exists for several reasons:
- The functions that you require do not relate to interacting with external systems. Most of the node design pattern concentrates on communication with external systems, which is the most likely requirement for a user-defined node.
- The functions that are required are not well suited to the WebSphere Message Broker architecture, so you should implement them in an end application, or an application server.
- The functions require complex control and state information, which you should not implement as a plug-in.

## Creating a user-defined node

You can write user-defined nodes in C or Java.

**Before you start:**

Read "Designing a user-defined node" on page 34.

When you have created a user-defined node, you can test it, as described in "Testing a user-defined node" on page 84. If you want to test or use user-defined nodes or parsers on multiple computers, follow the instructions given in "Packaging and distributing user-defined extensions" on page 101.

Decide whether you want to create a user-defined node in C or in Java, then follow the instructions in the appropriate topic.
- "Creating a user-defined extension in C" on page 36
- "Creating a user-defined extension in Java" on page 59
- "Creating the user interface representation of a user-defined node in the workbench" on page 76

The following table shows the tasks that are involved in creating the different types of user-defined node.

| Objective | Tasks to complete |
|---|---|
| To use one of the Java sample nodes | 1. "Compiling a Java user-defined node" on page 74<br>2. "Installing user-defined extension runtime files on a broker" on page 106<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 76<br>4. "Testing a user-defined node" on page 84 |
| To use one of the C sample nodes | 1. "Compiling a C user-defined extension" on page 52<br>2. "Installing user-defined extension runtime files on a broker" on page 106<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 76<br>4. "Testing a user-defined node" on page 84 |
| To create your own Java node using the workbench | 1. "Creating an input node in Java" on page 60 or "Creating a message processing or output node in Java" on page 65<br>2. "Using error logging from a user-defined extension" on page 115<br>3. "Compiling a Java user-defined node" on page 74<br>4. "Testing a user-defined node" on page 84<br>5. "Packaging a user-defined node workbench project" on page 107<br>6. "Installing a user-defined extension to current and past versions of the broker" on page 108 |

| Objective | Tasks to complete |
|---|---|
| To create your own C node: | 1. "Creating an input node in C" or "Creating a message processing or output node in C" on page 44 |
| | 2. "Using error logging from a user-defined extension" on page 115 |
| | 3. "Compiling a C user-defined extension" on page 52 |
| | 4. "Installing user-defined extension runtime files on a broker" on page 106 |
| | 5. "Creating the user interface representation of a user-defined node in the workbench" on page 76 |
| | 6. "Testing a user-defined node" on page 84 |
| | 7. "Packaging a user-defined node workbench project" on page 107 |
| | 8. "Installing a user-defined extension to current and past versions of the broker" on page 108 |

**Creating a user-defined extension in C:**

You must complete a series of tasks to create user-defined extensions that use the C language.

You can write user-defined nodes and user-defined parsers in C.

Complete the appropriate tasks from the following list:
- "Creating an input node in C"
- "Creating a message processing or output node in C" on page 44
- "Implementing a user-defined parser" on page 86
- "Compiling a C user-defined extension" on page 52

When you have completed this set of tasks, continue with the following tasks:
- If you have compiled a user-defined node, "Creating the user interface representation of a user-defined node in the workbench" on page 76
- "Testing a user-defined node" on page 84
- "Packaging and distributing user-defined extensions" on page 101

*Creating an input node in C:*

Create a user-defined input node in C to receive messages into a message flow.

**Before you start**

Read the following topics:
- "Why use a user-defined extension?" on page 4
- "User-defined input nodes" on page 12

A loadable implementation library, or *LIL*, is the implementation module for a C node. A LIL is implemented as a shared or dynamic link library (DLL), but has the file extension `.lil` not `.dll`.

The implementation functions that you write for the node are listed in "C node implementation functions" on page 124. You can call utility functions, implemented in the runtime broker, to help with the node operation; these functions are listed in "C node utility functions" on page 125.

WebSphere Message Broker provides the source for two sample user-defined nodes called SwitchNode and TransformNode. You can use these nodes in their current state, or you can modify them.

To create an input node in C:
1. "Declaring and defining the node"
2. "Creating an instance of the node" on page 38
3. "Setting attributes" on page 39
4. "Implementing the node functionality" on page 40
5. "Overriding the default message parser attributes (optional)" on page 40
6. "Deleting an instance of the node" on page 41

*Declaring and defining the node:*

To declare and define a user-defined node to the broker, include an initialization function, bipGetMessageflowNodeFactory, in your LIL. The following steps outline how the broker calls your initialization function, and how your initialization function declares and defines the user-defined node:

1. The initialization function, bipGetMessageflowNodeFactory, is called by the broker after the operating system has loaded and initialized the LIL. The broker calls this function to understand what your LIL can do and how the broker can call the LIL. For example:

   ```
   CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix
   bipGetMessageflowNodeFactory()
   ```

2. The bipGetMessageflowNodeFactory function must call the utility function cniCreateNodeFactory. This function passes back a unique factory name (or group name) for all the nodes that your LIL supports. Every factory name (or group name) that is passed back must be unique throughout all the LILs in a single runtime broker.

3. The LIL must call the utility function cniDefineNodeClass to pass the unique name of each node, and a virtual function table of the addresses of the implementation functions.

   For example, the following code declares and defines a single node called InputxNode:

   ```
   {
    CciFactory* factoryObject;
    int rc = 0;
    CciChar factoryName[] = L"MyNodeFactory";
    CCI_EXCEPTION_ST exception_st;

    /* Create the Node Factory for this node */
    factoryObject = cniCreateNodeFactory(0, factoryName);
    if (factoryObject == CCI_NULL_ADDR) {

     /* Any local error handling can go here */
    }
    else {
     /* Define the nodes supported by this factory */
     static CNI_VFT vftable = {CNI_VFT_DEFAULT};

     /* Setup function table with pointers to node implementation functions */
   ```

```
vftable.iFpCreateNodeContext = _createNodeContext;
vftable.iFpDeleteNodeContext = _deleteNodeContext;
vftable.iFpGetAttributeName2 = _getAttributeName2;
vftable.iFpSetAttribute      = _setAttribute;
vftable.iFpGetAttribute2     = _getAttribute2;
vftable.iFpRun               = _run;

cniDefineNodeClass(0, factoryObject, L"InputxNode", &vftable);
}


/* Return address of this factory object to the broker */
return(factoryObject);
}
```

A user-defined node identifies itself as providing the features of an input node by implementing the cniRun implementation function.

User-defined nodes have to implement either a cniRun or a cniEvaluate implementation function. If they do not, the broker does not load the user-defined node, and the cniDefineNodeClass utility function fails, returning CCI_MISSING_IMPL_FUNCTION.

For example:

```
int cniRun(
  CciContext* context,
  CciMessage* localEnvironment,
  CciMessage* exceptionList,
  CciMessage* message
){
  ...
  /* Get data from external source */
  return CCI_SUCCESS_CONTINUE;
}
```

Use the return value periodically to return control to the broker.

When a message flow that contains a user-defined input node is deployed successfully, the cniRun function of the node is called repeatedly to enable the node to retrieve messages and propagate them to the rest of the message flow. For the minimum code required to compile a C user-defined node, see the "C skeleton code" on page 300.

*Creating an instance of the node:*

To instantiate your node:

1. When the broker has received the table of function pointers, it calls the function cniCreateNodeContext for each instantiation of the user-defined node. For example, if three message flows are using your user-defined node, your cniCreateNodeContext function is called for each of them. This function must allocate memory for that instantiation of the user-defined node to hold the values for the configured attributes. For example:

   a. Call the cniCreateNodeContext function:

   ```
   CciContext* _createNodeContext(
     CciFactory* factoryObject,
     CciChar*    nodeName,
     CciNode*    nodeObject
   ){
     static char* functionName = (char *)"_createNodeContext()";
     NODE_CONTEXT_ST* p;
     CciChar          buffer[256];
   ```

   b. Allocate a pointer to the local context and clear the context area:

```
              p = (NODE_CONTEXT_ST *)malloc(sizeof(NODE_CONTEXT_ST));

          if (p) {
              memset(p, 0, sizeof(NODE_CONTEXT_ST));
```
   c. Save the node object pointer in the context:
```
              p->nodeObject = nodeObject;
```
   d. Save the node name:
```
          CciCharNCpy((CciChar*)&p->nodeName, nodeName, MAX_NODE_NAME_LEN);
```
   e. Return the node context:
```
          return (CciContext*) p;
```
2. An input node has a number of output terminals associated with it, but typically does not have any input terminals. Use the utility function cniCreateOutputTerminal to add output terminals to an input node when the node is instantiated. These functions must be invoked within the cniCreateNodeContext implementation function. For example, to define an input node with three output terminals:

```
    {
        const CciChar* ucsOut = CciString("out", BIP_DEF_COMP_CCSID) ;
        insOutputTerminalListEntry(p, (CciChar*)ucsOut);
        free((void *)ucsOut) ;
    }
    {
        const CciChar* ucsFailure = CciString("failure", BIP_DEF_COMP_CCSID) ;
        insOutputTerminalListEntry(p, (CciChar*)ucsFailure);
        free((void *)ucsFailure) ;
    }
    {
        const CciChar* ucsCatch = CciString("catch", BIP_DEF_COMP_CCSID) ;
        insOutputTerminalListEntry(p, (CciChar*)ucsCatch);
        free((void *)ucsCatch) ;     }
```

For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 300.

*Setting attributes:*

Attributes are set whenever you start the broker, or when you redeploy the message flow with new values.

Following the creation of output terminals, the broker calls the cniSetAttribute function to pass the values for the configured attributes of the user-defined node. For example:

```
    {
      const CciChar* ucsAttr = CciString("nodeTraceSetting", BIP_DEF_COMP_CCSID) ;
      insAttrTblEntry(p, (CciChar*)ucsAttr, CNI_TYPE_INTEGER);
      _setAttribute(p, (CciChar*)ucsAttr, (CciChar*)constZero);
      free((void *)ucsAttr) ;
    }
    {
      const CciChar* ucsAttr = CciString("nodeTraceOutfile", BIP_DEF_COMP_CCSID) ;
      insAttrTblEntry(p, (CciChar*)ucsAttr, CNI_TYPE_STRING);
      _setAttribute(p, (CciChar*)ucsAttr, (CciChar*)constSwitchTraceLocation);
      free((void *)ucsAttr) ;
    }
```

The number of configuration attributes that a node can have is unlimited. However, a user-defined node must not implement an attribute that is already implemented as a base configuration attribute. The base attributes are:

- label

- userTraceLevel
- traceLevel
- userTraceFilter
- traceFilter

*Implementing the node functionality:*

When the broker knows that it has an input node, it calls the cniRun function of this node at regular intervals. The cniRun function must then decide what course of action it must take. If data is available for processing, the cniRun function can propagate the message. If no data is available for processing, the cniRun function must return with CCI_TIMEOUT so that the broker can continue configuration changes.

For example, to configure the node to call cniDispatchThread and process the message, or return with CCI_TIMEOUT:

```
If ( anything to do )
 CniDispatchThread;

   /* do the work */

 If ( work done O.K.)
  Return CCI_SUCCESS_CONTINUE;
 Else
  Return CCI_FAILURE_CONTINUE;
Else
  Return CCI_TIMEOUT;
```

*Overriding the default message parser attributes (optional):*

An input node implementation typically determines what message parser initially parses an input message. For example, the MQInput node dictates that an MQMD parser is required to parse the MQMD header. A user-defined input node can select an appropriate header or message parser, and the mode in which the parsing is controlled, by using or overriding the following attributes that are included as default:

**rootParserClassName**
> Defines the name of the root parser that parses message formats that are supported by the user-defined input node. It defaults to GenericRoot, a supplied root parser that causes the broker to allocate and chain parsers together. It is unlikely that a node would need to modify this attribute value.

**firstParserClassName**
> Defines the name of the first parser, in what might be a chain of parsers that are responsible for parsing the bit stream. It defaults to XML.

**messageDomainProperty**
> An optional attribute that defines the name of the message parser that is required to parse the input message. The supported values are the same as the values that are supported by the MQInput node. (See MQInput node for more information.)

**messageSetProperty**
> An optional attribute that defines the message set identifier, or the message set name, in the Message Set field, only if the MRM parser was specified by the messageDomainProperty attribute.

**messageTypeProperty**

An optional attribute that defines the identifier of the message in the `MessageType` field, only if the MRM parser was specified by the `messageDomainProperty` attribute.

**messageFormatProperty**

An optional attribute that defines the format of the message in the `Message Format` field, only if the MRM parser was specified by the `messageDomainProperty` attribute.

If you have written a user-defined input node that always begins with data of a known structure, you can ensure that a specific parser handles the start of the data. For example, the MQInput node reads data only from WebSphere MQ queues, therefore this data always has an MQMD at the beginning, and the MQInput node sets firstParserClassName to MQHMD. If your user-defined node always handles data that begins with a structure that can be parsed by a specific parser, for example "MYPARSER", set firstParserClassName to MYPARSER in the following way::

1. Declare the implementation functions:

```
CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix bipGetMessageflowNodeFactory()
{
  ....
  CciFactory*      factoryObject;
  ....
  factoryObject = cniCreateNodeFactory(0, (unsigned short *)constPluginNodeFactory);
  ...
  vftable.iFpCreateNodeContext = _createNodeContext;
  vftable.iFpSetAttribute      = _setAttribute;
  vftable.iFpGetAttribute      = _getAttribute;
  ...
  cniDefineNodeClass(&rc, factoryObject, (CciChar*)constSwitchNode, &vftable);
  ...
  return(factoryObject);
}
```

2. Set the attribute in the cniCreateNodeContext implementation function:

```
CciContext* _createNodeContext(
  CciFactory* factoryObject,
  CciChar*    nodeName,
  CciNode*    nodeObject
){
  NODE_CONTEXT_ST* p;
  ...

    /* Allocate a pointer to the local context */
    p = (NODE_CONTEXT_ST *)malloc(sizeof(NODE_CONTEXT_ST));
    /* Create attributes and set default values */
    {
      const CciChar* ucsAttrName  = CciString("firstParserClassName", BIP_DEF_COMP_CCSID);
      const CciChar* ucsAttrValue = CciString("MYPARSER", BIP_DEF_COMP_CCSID) ;
      insAttrTblEntry(p, (CciChar*)ucsAttrName, CNI_TYPE_INTEGER);
      /*see sample BipSampPluginNode.c for implementation of insAttrTblEntry*/

      _setAttribute(p, (CciChar*)ucsAttrName, (CciChar*)ucsAttrValue);
      free((void *)ucsAttrName) ;
      free((void *)ucsAttrValue) ;
    }
```

In the code example above, the insAttrTblEntry method is called. This method is declared in the SwitchNode and TransformNode sample user-defined nodes.

*Deleting an instance of the node:*

Nodes are destroyed when a message flow is redeployed, or when the execution group process is stopped (using the mqsistop command). When a node is

destroyed, you must call the cniDeleteNodeContext function to free all used memory and release all held resources. For example:

```
void _deleteNodeContext(
  CciContext* context
){
  static char* functionName = (char *)"_deleteNodeContext()";

  return;
}
```

*Extending the capability of a C input node:*

When you have created a user-defined node, you can extend its capability.

**Before you start**

Read "Creating an input node in C" on page 36.

After you have created a user-defined node, the following options are available:
1. "Receiving external data into a buffer"
2. "Controlling threading and transactions"
3. "Propagating the message" on page 43

*Receiving external data into a buffer:*

An input node can receive data from any type of external source, such as a file system or FTP connection, provided that the output from the node is in the correct format. For connections to queues or databases, use the built-in nodes and the API calls supplied, principally because the built-in nodes are already set up for error handling. Do not use the MQGET or MQPUT calls for direct access to WebSphere MQ queues.

You must provide an input buffer (or bit stream) to contain input data, and associate it with a message object. In the C API, the buffer is attached to the CciMessage object that represents the input message by using the cniSetInputBuffer utility function. For example:

```
{
  static char* functionName = (char *)"_Input_run()";
  void*         buffer;
  CciTerminal* terminalObject;
  int          buflen = 4096;
  int          rc = CCI_SUCCESS;
  int          rcDispatch = CCI_SUCCESS;

  buffer = readFromDevice(&buflen);
  cniSetInputBuffer(&rc, message, buffer, buflen);
}
/*propagate etc*/
```

*Controlling threading and transactions:*

An input node must perform appropriate end-of-message processing when a message has been propagated through a message flow. Specifically, the input node needs to cause any transactions to be committed or rolled back, and return threads to the thread pool.

Each message flow thread is allocated from a pool of threads that is maintained for each message flow, and starts execution in the cniRun function. You determine the behavior of a thread using the cniDispatchThread utility function, together with the appropriate return value.

From the cniRun function, you can call the cniDispatchThread utility function to cause another thread to start executing the cniRun function. The most appropriate time to execute another thread is directly after you have established that data could be available for the function to process on the new thread.

The term *transaction* is used generically to describe either a globally coordinated transaction, or a broker-controlled transaction. Globally coordinated transactions are coordinated by either WebSphere MQ as an XA-compliant Transaction Manager, or Resource Recovery Service (RRS) on z/OS®. WebSphere Message Broker controls transactions by committing (or rolling back) any database resources, and then committing any WebSphere MQ units of work. However, if a user-defined node is used, the broker cannot automatically commit any resource updates. The user-defined node uses return values to indicate whether a transaction has been successful, and to control whether transactions are committed or rolled-back. The broker infrastructure catches any unhandled exceptions, and rolls back the transaction.

The following table describes each of the supported return values, the effect that each one has on any transactions, and what the broker does with the current thread.

| Return value | Affect on transaction | Broker action on the thread |
|---|---|---|
| CCI_SUCCESS_CONTINUE | Committed | Calls the same thread again in the cniRun function. |
| CCI_SUCCESS_RETURN | Committed | Returns the thread to the thread pool. |
| CCI_FAILURE_CONTINUE | Rolled back | Calls the same thread again in the cniRun function. |
| CCI_FAILURE_RETURN | Rolled back | Returns the thread to the thread pool. |
| CCI_TIMEOUT | Not applicable. The function periodically times out while it is waiting for an input message. | Calls the same thread again in the cniRun function. |

The following code is an example of using the SUCCESS_RETURN return code with the cniDispatchThread function:

```
{
  ...
  cniDispatchThread(&rcDispatch, ((NODE_CONTEXT_ST *)context)->nodeObject);
  ...
  if (rcDispatch == CCI_NO_THREADS_AVAILABLE) return CCI_SUCCESS_CONTINUE;
  else return CCI_SUCCESS_RETURN;
}
```

*Propagating the message:*

Before you propagate a message, decide what message flow data you want to propagate, and which terminal is to receive the data.

The terminalObject is derived from a list that the user-defined node maintains.

For example, to propagate the message to the output terminal, use the cniPropagate function:

```
if (terminalObject) {
  if (cniIsTerminalAttached(&rc, terminalObject)) {
    if (rc == CCI_SUCCESS) {
      cniPropagate(&rc, terminalObject, localEnvironment, exceptionList, message);
    }
  }
}
```

In the above example, the cniIsTerminalAttached function is used to test whether the message can be propagated to the specified terminal. If you do not use the cniIsTerminalAttached function, and the terminal is not attached to another node by a connector, the message is not propagated and no warning message is returned. Use the cniIsTerminalAttached function to prevent this error occurring.

*Creating a message processing or output node in C:*

A message processing node is used to process a message in some way, and an output node is used to produce a message as a bit stream.

**Before you start**

Read the following topics:
- "Why use a user-defined extension?" on page 4
- "User-defined message processing nodes" on page 17
- "User-defined output nodes" on page 25

When you code a message processing node or an output node, the nodes provide essentially the same services. You can perform message processing in an output node, and you can send a message to a bit stream by using a message processing node. For simplicity, this topic refers mainly to the node as a message processing node but it does also contain information about the functions of both types of node.

A loadable implementation library (LIL), is the implementation module for a C node. A LIL is implemented as a shared or dynamic link library (DLL), but has the file extension .lil not .dll.

For more information about the C node implementation functions that you write for the node, see "C node implementation functions" on page 124. You can call C node utility functions, implemented in the runtime broker, to help with the node operation; see "C node utility functions" on page 125.

WebSphere Message Broker provides the source for two sample user-defined nodes called SwitchNode and TransformNode. You can use these nodes in their current state, or you can modify them. In addition, you can view the following sample which demonstrates the use of user-defined nodes, including a message processing node written in C.
- User-defined Extension

You can view samples information only when you use the information center that is integrated with the Message Broker Toolkit or the online information center.

To create either type of node complete the following tasks:

1. "Declaring and defining your node"
2. "Creating an instance of the node" on page 46
3. "Setting attributes" on page 48
4. "Implementing the node functionality" on page 49
5. "Deleting an instance of the node" on page 49

*Declaring and defining your node:*

To declare and define a user-defined node to the broker, include an initialization function, bipGetMessageflowNodeFactory, in your LIL. The following steps take place on the configuration thread and outline how the broker calls your initialization function and how your initialization function declares and defines the user-defined node:

1. The broker calls the initialization function bipGetMessageflowNodeFactory after the operating system has loaded and initialized the LIL. The broker calls this function to understand what your LIL can do and how the broker can call the LIL. For example:

   ```
   CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix
   bipGetMessageflowNodeFactory()
   ```

2. The bipGetMessageflowNodeFactory function must call the utility function cniCreateNodeFactory. This function passes back a factory name (or group name) for all the nodes that your LIL supports. The factory name (or group name) must be unique throughout all the LILs in a single runtime broker.
3. The LIL must call the utility function cniDefineNodeClass to pass the unique name of each node and a virtual function table of the addresses of the implementation functions.

   For example, the following code declares and defines a single node called MessageProcessingxNode:

   ```
   {
    CciFactory* factoryObject;
    int rc = 0;
    CciChar factoryName[] = L"MyNodeFactory";
    CCI_EXCEPTION_ST exception_st;

    /* Create the Node Factory for this node */
    factoryObject = cniCreateNodeFactory(0, factoryName);
    if (factoryObject == CCI_NULL_ADDR) {
     /* Any local error handling can go here */
    }
    else {
     /* Define the nodes supported by this factory */
    static CNI_VFT vftable = {CNI_VFT_DEFAULT};

    /* Setup function table with pointers to node implementation functions */
    vftable.iFpCreateNodeContext = _createNodeContext;
    vftable.iFpDeleteNodeContext = _deleteNodeContext;
    vftable.iFpGetAttributeName2 = _getAttributeName2;
    vftable.iFpSetAttribute      = _setAttribute;
    vftable.iFpGetAttribute2     = _getAttribute2;
    vftable.iFpEvaluate          = _evaluate;

    cniDefineNodeClass(0, factoryObject, L"MessageProcessingxNode", &vftable);

    }

    /* Return address of this factory object to the broker */
    return(factoryObject);
   }
   ```

A user-defined node identifies itself as a message processing or output node by implementing the cniEvaluate function. User-defined nodes must implement either a cniEvaluate or a cniRun implementation function, otherwise the broker does not load the user-defined node, and the cniDefineNodeClass utility function fails, returning CCI_MISSING_IMPL_FUNCTION.

When a message flow containing a user-defined message processing node is deployed successfully, the node's cniEvaluate function is called for each message propagated to the node.

Message flow data is received at the input terminal of the node, that is, the message, Environment, LocalEnvironment, and ExceptionList.

For example:

```
void cniEvaluate(
  CciContext* context,
  CciMessage* localEnvironment,
  CciMessage* exceptionList,
  CciMessage* message
){
  ...
}
```

For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 300.

*Creating an instance of the node:*

To instantiate your node:

1. When the broker has received the table of function pointers, it calls the function cniCreateNodeContext for each instantiation of the user-defined node. For example, if three message flows are using your user-defined node, your cniCreateNodeContext function is called for each of them. This function allocates memory for that instantiation of the user-defined node to hold the values for the configured attributes. For example:

   a. The user function cniCreateNodeContext is called:

      ```
      CciContext* _Switch_createNodeContext(
        CciFactory* factoryObject,
        CciChar*    nodeName,
        CciNode*    nodeObject
      ){
        static char* functionName = (char *)"_Switch_createNodeContext()";
        NODE_CONTEXT_ST* p;
        CciChar          buffer[256];
      ```

   b. Allocate a pointer to the local context and clear the context area:

      ```
      p = (NODE_CONTEXT_ST *)malloc(sizeof(NODE_CONTEXT_ST));

      if (p) {
         memset(p, 0, sizeof(NODE_CONTEXT_ST));
      ```

   c. Save the node object pointer in the context:

      ```
      p->nodeObject = nodeObject;
      ```

   d. Save the node name:

      ```
      CciCharNCpy((CciChar*)&p->nodeName, nodeName, MAX_NODE_NAME_LEN);
      ```

   e. Return the node context:

      ```
      return (CciContext*) p;
      ```

2. The broker calls the appropriate utility functions to find out about the node's input terminals and output terminals. A node has a number of input terminals and output terminals associated with it. Within the user function

cniCreateNodeContext, calls must be made to cniCreateInputTerminal and cniCreateOutputTerminal to define the user node's terminals. These functions must be started within the cniCreateNodeContext implementation function. For example, to define a node with one input terminal and two output terminals:

```
{
  const CciChar* ucsIn = CciString("in", BIP_DEF_COMP_CCSID) ;
  insInputTerminalListEntry(p, (CciChar*)ucsIn);
  free((void *)ucsIn) ;
}
{
  const CciChar* ucsOut = CciString("out", BIP_DEF_COMP_CCSID) ;
  insOutputTerminalListEntry(p, (CciChar*)ucsOut);
  free((void *)ucsOut) ;
}
{
  const CciChar* ucsFailure = CciString("failure", BIP_DEF_COMP_CCSID) ;
  insOutputTerminalListEntry(p, (CciChar*)ucsFailure);
  free((void *)ucsFailure) ;
}
```

The previous code starts the insInputTerminalListEntry and insOutputTerminalListEntry functions. You can find these functions in the sample code Common.c; see "Sample node files" on page 121. These functions define the terminals to the broker and store handles to the terminals. Handles are stored in the structure referenced by the value returned in CciContext*. The node can then access the terminal handles from within the other implementation functions (for example CciEvaluate) because CciContext is passed to those implementation functions.

The following code shows the definition of insInputTerminalListEntry:

```
TERMINAL_LIST_ENTRY *insInputTerminalListEntry(
  NODE_CONTEXT_ST* context,
  CciChar*         terminalName
){
  static char* functionName = (char *)"insInputTerminalListEntry()";
  TERMINAL_LIST_ENTRY* entry;
  int                  rc;

  entry = (TERMINAL_LIST_ENTRY *)malloc(sizeof(TERMINAL_LIST_ENTRY));
  if (entry) {

    /* This entry is the current end of the list */
    entry->next = 0;

    /* Store the terminal name */
    CciCharCpy(entry->name, terminalName);

    /* Create terminal and save its handle */
    entry->handle = cniCreateInputTerminal(&rc, context->nodeObject, (CciChar*)terminalName);

    /* Link an existing previous element to this one */
    if (context->inputTerminalListPrevious) context->inputTerminalListPrevious->next = entry;
    else if ((context->inputTerminalListHead) == 0) context->inputTerminalListHead = entry;

    /* Save the pointer to the previous element */
    context->inputTerminalListPrevious = entry;
  }
  else {
    /* Error: Unable to allocate memory */
  }

  return(entry);
}
```

The following example shows the code for insOutputTerminalListEntry:

```
TERMINAL_LIST_ENTRY *insOutputTerminalListEntry(
  NODE_CONTEXT_ST* context,
  CciChar*         terminalName
){
  static char* functionName = (char *)"insOutputTerminalListEntry()";
  TERMINAL_LIST_ENTRY* entry;
  int                  rc;

  entry = (TERMINAL_LIST_ENTRY *)malloc(sizeof(TERMINAL_LIST_ENTRY));
  if (entry) {

    /* This entry is the current end of the list */
    entry->next = 0;

    /* Store the terminal name */
    CciCharCpy(entry->name, terminalName);

    /* Create terminal and save its handle */
    entry->handle = cniCreateOutputTerminal(&rc, context->nodeObject, (CciChar*)terminalName);

    /* Link an existing previous element to this one */
    if (context->outputTerminalListPrevious) context->outputTerminalListPrevious->next = entry;
    else if ((context->outputTerminalListHead) == 0) context->outputTerminalListHead = entry;

    /* Save the pointer to the previous element */
    context->outputTerminalListPrevious = entry;
  }
  else {
    /* Error: Unable to allocate memory */
  }

  return(entry);
}
```

For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 300.

*Setting attributes:*

Attributes are set whenever you start the broker, or when you redeploy a message flow with new values. Attributes are set by the broker calling user code on the configuration thread. Your code needs to store these attributes in its node context area, for later use when processing messages.

Following the creation of input and output terminals, the broker calls the cniSetAttribute function to pass the values for the configured attributes for this instantiation of the user-defined node. For example:

```
{
  const CciChar* ucsAttr = CciString("nodeTraceSetting", BIP_DEF_COMP_CCSID) ;
  insAttrTblEntry(p, (CciChar*)ucsAttr, CNI_TYPE_INTEGER);
  _setAttribute(p, (CciChar*)ucsAttr, (CciChar*)constZero);
  free((void *)ucsAttr) ;
}
{
  const CciChar* ucsAttr = CciString("nodeTraceOutfile", BIP_DEF_COMP_CCSID) ;
  insAttrTblEntry(p, (CciChar*)ucsAttr, CNI_TYPE_STRING);
  _setAttribute(p, (CciChar*)ucsAttr, (CciChar*)constSwitchTraceLocation);
  free((void *)ucsAttr) ;
}
```

The number of configuration attributes that a node can have is unlimited. However, a node must not implement an attribute that is already implemented as a base configuration attribute. The following list shows base attributes:

- label
- userTraceLevel
- traceLevel
- userTraceFilter
- traceFilter

*Implementing the node functionality:*

When the broker retrieves a message from the queue, and that message arrives at the input terminal of your user-defined message processing or output node, the broker calls the implementation function cniEvaluate. This function is called on the message processing thread and it must decide what to do with the message. This function might be called on multiple threads, especially if additional instances are used.

*Deleting an instance of the node:*

If a node is deleted, the broker calls the cniDeleteNodeContext function. This function is started on the same thread as cniCreateNodeContext. Use this function to release resources used by your user-defined node. For example:

```
void _deleteNodeContext(
  CciContext* context
){
  static char* functionName = (char *)"_deleteNodeContext()";
  free ((void*) context);
  return;
}
```

*Extending the capability of a C message processing or output node:*

When you have created a user-defined message processing or output node in C, you can extend its capability.

**Before you start**

After you have created a user-defined node, the following options are available:

*Accessing message data:*

In many cases, the user-defined node must access the contents of the message that is received on its input terminal. The message is represented as a tree of syntax elements. Groups of utility functions are provided for message management, message buffer access, syntax element navigation, and syntax element access. (See for details of the utility functions.)

The types of query that you are likely to want to perform include:
- Obtaining the root element of the required message object

- Accessing the bit stream representation of an element tree
- Navigating or querying the tree by asking for child or sibling elements by name
- Getting the type of the element
- Getting the value of the element

For example, to query the name and type of the first child of body:

```
void cniEvaluate( ...
){
  ...
/* Navigate to the target element */
  rootElement = cniRootElement(&rc, message);
  bodyElement = cniLastChild(&rc, rootElement);
  bodyFirstChild = cniFirstChild(&rc, bodyElement);

/* Query the name and value of the target element */
  cniElementName(&rc, bodyFirstChild, (CciChar*)&elementname, sizeof(elementName));
  bytes = cniElementCharacterValue(
  &rc, bodyfirstChild, (CciChar*)&eValue, sizeof(eValue));
  ...
}
```

To access the bit stream representation of an element tree you can use the cniElementAsBitstream function. Using this function, you can obtain the bit stream representation of any element in a message. See "cniElementAsBitstream" on page 148 for details of how to use this function, and sample code.

*Transforming a message object:*

The received input message is read-only, therefore before a message can be transformed, you must write it to a new output message using the cniCreateMessage function. You can copy elements from the input message, or you can create new elements and attach them to the message. New elements are typically in a parser's domain.

For example:

1. To write the incoming message to a new message:
   ```
   {
     ...
     context = cniGetMessageContext(&rc, message));
     outMsg = cniCreateMessage(&rc, context));
     ...
   }
   ```
2. To make a copy of the new message:
   ```
   cniCopyElementTree(&rc, sourceElement, targetElement);
   ```
3. To modify the value of a target element:
   ```
   cniSetElementIntegerValue(&rc, targetElement, L"newValue", 8);
   ```
4. After finalizing and propagating the message, you must delete the output message using the cniDeleteMessage function:
   ```
   cniDeleteMessage(&rc, outMsg);
   ```

As part of the transformation, you might want to create a new message body. To create a new message body, use one of the following functions, which assign a parser to a message tree folder:

```
cniCreateElementAsFirstChildUsingParser
cniCreateElementAsLastChildUsingParser
cniCreateElementAfterUsingParser
cniCreateElementBeforeUsingParser
```

When creating a message body, do not use the following functions because they do not associate an owning parser with the folder:

```
cniCreateElementAsFirstChild
cniCreateElementAsLastChild
cniCreateElementAfter
cniCreateElementBefore
```

*Accessing ESQL:*

Nodes can invoke ESQL expressions using Compute node ESQL syntax. You can create and modify the components of the message using ESQL expressions, and you can refer to elements of both the input message and data from an external database using the cniSqlCreateStatement, cniSqlSelect, cniSqlDeleteStatement, and cniSqlExecute functions.

For example, to populate the Result element from the contents of a column in a database table:

```
{
  ...
  sqlExpr = cniSqlCreateStatement(&rc,
   (NODE_CONTEXT_ST *)context->nodeObject,
   L"DB", CCI_SQL_TRANSACTION_AUTO,
   L"SET OutputRoot.XMLNS.Result[] = (SELECT T.C1 AS Col1 FROM Database.TABLE AS T;");
  ...
  cniSqlSelect(&rc, sqlExpr, localEnvironment, exceptionList, message, outMsg);
  cniSqlDeleteStatement(&rc, sqlExpr);
  ...
}
```

For more information about ESQL, see ESQL overview.

If your user-defined node primarily uses ESQL, consider using a Compute node.

*Propagating a message:*

Before you propagate a message, decide what message flow data you want to propagate, and which terminal is to receive the data.

1. If the message has changed, finalize the message before you propagate it using the cniFinalize function. For example:

   ```
   cniFinalize(&rc, outMsg, CCI_FINALIZE_NONE);
   ```

2. The terminalObject is derived from a list that the user-defined node maintains itself. To propagate the message to the output terminal, use the cniPropagate function:

   ```
   if (terminalObject) {
     if (cniIsTerminalAttached(&rc, terminalObject)) {
       if (rc == CCI_SUCCESS) {
         cniPropagate(&rc, terminalObject, localEnvironment, exceptionList, outMsg);
       }
     }
   ```

   In the above example, the cniIsTerminalAttached function is used to test whether the message can be propagated to the specified terminal. If you do not use the cniIsTerminalAttached function and the terminal is not attached to another node by a connector, the message is not propagated and no warning message is returned. Use the cniIsTerminalAttached function to prevent this error occurring.

3. If you created a new output message using cniCreateMessage, after propagating the message, delete the output message using the cniDeleteMessage function:

```
cniDeleteMessage(&rc, outMsg);
```

*Writing to an output device:*

A transformed message must be serialized to a bit stream; a message can be serialized only once.

The bit stream can then be accessed and written to an output device. Write the message to a bit stream using the cniWriteBuffer function. For example:

```
{
  ...
  cniWriteBuffer(&rc, message);
  writeToDevice(cniBufferPointer(&rc, message), cniBufferSize(&rc, message));
  ...
}
```

In this example, the method writeToDevice is a user-written method which writes a bit stream to an output device.

Do not write a user-defined output node to write messages to WebSphere MQ queues; use the supplied MQOutput node in this scenario. The broker internally maintains a WebSphere MQ connection and open queue handles on a thread-by-thread basis, and these are cached to optimize performance. In addition, the broker handles recovery scenarios when certain WebSphere MQ events occur; this recovery would be adversely affected if WebSphere MQ MQI calls are used in a user-defined output node.

*Compiling a C user-defined extension:*

Compile user-defined extensions in C for all supported operating systems.

**Before you start:**

If you create your own user-defined nodes, parsers, and user exits in C, compile them on the operating system on which the target broker is running. Samples are provided for both nodes and parsers, and are described in "Sample node files" on page 121 and "Sample parser files" on page 123. Use the instructions here to compile the samples. If you want to create your own extensions, see the following topics:

* "Creating a user-defined extension in C" on page 36
* "Implementing a user-defined parser" on page 86
* "Implementing a user-defined exit" on page 99

These instructions use the file names of the supplied samples. If you are compiling your own user-defined extensions, substitute your own file names.

When you compile a user-defined extension that is written in C, you need a compatible compiler. For details of supported compilers, see Optional software support.

*Header files:*

The following header files define the C interfaces:

**BipCni.h**

Message processing nodes

**BipCpi.h**

Message parsers

**BipCci.h**

Interfaces common to both nodes and parsers

**BipCos.h**

Platform-specific definitions

*Compiling:*

Compile the source for your user-defined extension on each of the supported operating systems to create the executable file that the broker calls to implement your user-defined extension. On Linux, UNIX, and z/OS systems, this file is a loadable implementation library (LIL) file; on Windows systems, it is a dynamic load library (DLL) file.

The libraries that you build to contain user-defined nodes or parsers must have the extension `.lil` on all operating systems so that the broker can load them. Libraries that contain user exits must have the extension `.lel` on all operating systems. The examples in this topic show libraries with the extension `.lil`

Refer to the documentation for the compiler that you are using for full details of available compile and link options that might be required for your programs.

Navigate to the directory where your user-defined extension source code is located, and follow the instructions for your operating system:
- AIX®
- HP-UX on PA-RISC
- HP-Itanium®
- Linux
- Solaris
- Windows
- z/OS

*Compiling on AIX:*

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a default 64-bit execution group. If the extension is called from a 32-bit execution group, make the following substitutions in the compile and link examples:
- Replace `xlc_r -q64 -qwarn64` with `xlc_r -q32`
- Replace `-l imbdfplg64` with `-l imbdfplg`

```
xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c SwitchNode.c \
   -o SwitchNode.o

xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c BipSampPluginUtil.c \
   -o BipSampPluginUtil.o
```

```
xlc_r -q64 -qwarn64 \
    -I. \
    -I/install_dir/include/plugin \
    -c Common.c \
    -o Common.o

xlc_r -q64 -qwarn64 \
    -I. \
    -I/install_dir/include/plugin \
    -c NodeFactory.c \
    -o NodeFactory.o

xlc_r -q64 -qwarn64 \
    -qmkshrobj \
    -bM:SRE \
    -bexpall \
    -bnoentry \
    -o SwitchNode.lil SwitchNode.o \
        BipSampPluginUtil.o Common.o NodeFactory.o \
    -L /install_dir/lib \
    -l imbdfplg64

chmod a+r SwitchNode.lil
```

*Compiling on HP-UX on PA-RISC:*

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a default 64-bit execution group. If the extension is called from a 32-bit execution group, make the following substitutions in the compile and link examples:

- Replace +DD64 with +DD32
- Replace -l imbdfplg64 with -l imbdfplg

```
cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c BipSampPluginUtil.c \
    -o output_dir/BipSampPluginUtil.o \
    -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c Common.c \
    -o output_dir/Common.o \
    -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c NodeFactory.c \
    -o output_dir/NodeFactory.o \
    -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c SwitchNode.c \
```

```
   -o output_dir/SwitchNode.o \
   -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT\
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c TransformNode.c \
   -o output_dir/TransformNode.o \
   -Ae

ld -b \
   -o output_dir/SwitchNode.lil \
   output_dir/BipSampPluginUtil.o \
   output_dir/Common.o \
   output_dir/NodeFactory.o \
   output_dir/SwitchNode.o \
   output_dir/TransformNode.o \
   -L install_dir/lib \
   -L install_dir/xml4c/lib \
   -L install_dir/merant/lib \
   -L install_dir/jre/lib/PA_RISC2.0 \
   -L install_dir/jre/lib/PA_RISC2.0/server \
   -l imbdfplg64

chmod a+r output_dir/SwitchNode.lil
```

*Compiling on HP-Itanium:*

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a 64-bit execution group; 32-bit execution groups are not supported.

```
cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
   -I. \
   -I/install_dir/include \
```

```
  -I/install_dir/include/plugin \
  -c TransformNode.c \
  -o output_dir/TransformNode.o


ld -b \
  -o output_dir/SwitchNode.lil \
  output_dir/BipSampPluginUtil.o \
  output_dir/Common.o \
  output_dir/NodeFactory.o \
  output_dir/SwitchNode.o \
  output_dir/TransformNode.o \
  -L install_dir/lib \
  -L install_dir/xml4c/lib \
  -L install_dir/merant/lib \
  -L install_dir/jre/lib/IA64N\
  -L install_dir/jre/lib/IA64N/server \
  -l imbdfplg

chmod a+r output_dir/SwitchNode.lil
```

*Compiling on Linux:*

When you compile a user-defined extension that is written in C, use a supported compiler.

To determine the 32-bit or 64-bit linkage that is required for an execution group created in a broker on your specific operating system, see Support for 32-bit and 64-bit platforms.

When you compile programs on Linux on POWER®, replace the option -fpic with -fPIC if you want to use dynamic linking and avoid any limit on the size of the global offset table.

The following instructions are for compiling an extension for a 64-bit execution group on Linux on POWER and Linux on System z®.

- To compile the extension for a 64-bit execution group on Linux on x86-64, replace -limbdfplg with -limbdfplg64 in the link example.
- To compile the extension for a 32-bit execution group on Linux on x86, replace -m64 with -m32 in the compile and link examples.

```
g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
  TransformNode.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
  SwitchNode.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
  BipSampPluginUtil.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
```

```
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   Common.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   NodeFactory.c

g++ -m64 -o samples.lil \
   TransformNode.o \
   SwitchNode.o \
   BipSampPluginUtil.o \
   Common.o NodeFactory.o \
   -shared -lc -lnsl -ldl \
   -L/install_dir/lib -limbdfplg
```

These commands create the file `samples.lil` that provides TransformNode and SwitchNode objects.

*Compiling on Solaris:*

When you compile a user-defined extension that is written in C, use a supported compiler.

To determine the 32-bit or 64-bit linkage that is required for an execution group created in a broker on your Solaris operating system, see Support for 32-bit and 64-bit platforms.

The following instructions are for compiling an extension for a 64-bit execution group on Solaris on SPARC.
- To compile the extension for a default 64-bit execution group on Solaris on x86-64, replace -xarch=v9 with -xarch=amd64 in the compile examples.
- To compile the extension for a 32-bit execution group on Solaris on SPARC:
  - Replace -xarch=v9 with -xarch=v8plus in the compile examples.
  - Replace -l imddfplg64 with -l imddfplg in the link example.

```
cc -xarch=v9 -mt \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc -xarch=v9 -mt \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

cc -xarch=v9 -mt \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc -xarch=v9 -mt \
```

```
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

cc -xarch=v9  -xcode=pic32 -mt -G \
   -o output_dir/SwitchNode.lil \
      output_dir/SwitchNode.o \
      output_dir/BipSampPluginUtil.o \
      output_dir/NodeFactory.o \
      output_dir/Common.o \
   -L /install_dir/lib \
   -l imbdfplg64

chmod a+r output_dir/SwitchNode.lil
```

*Compiling on Windows:*

When you compile a user-defined extension that is written in C, use a supported compiler.

Ensure that you include a space between `SwitchNode.c` and `BipSampPluginUtil.c`, and also between `-link` and `/DLL`.

Enter the command as a single line of input; in the following example the lines have been split to improve readability.

```
cl /VERBOSE /LD /MD /Zi /GX /I.
   /Iinstall_dir\include\plugin
   SwitchNode.c BipSampPluginUtil.c Common.c
   NodeFactory.c TransformNode.c
   -link /DLL install_dir\lib\imbdfplg.lib
   /OUT:SwitchNode.lil
```

If you have correctly set the *LIB* environment variable, you do not have to specify the full paths to the LIB files.

*Compiling on z/OS:*

When you compile a user-defined extension that is written in C, use a supported compiler.

Force your link to use prelinker or linker by setting the _CC_STEPS variable to **-1**:

```
export _CC_STEPS=-1
```

Alternatively, add these two lines to your makefile to export it:

```
_CC_STEPS=-1
.EXPORT : _CC_STEPS
```

To create optimized builds, use **-2** in place of **-g** in the following commands:

```
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./SwitchNode.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./SwitchNode.o ./SwitchNode.c
```

```
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./BipSampPluginUtil.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./BipSampPluginUtil.o ./BipSampPluginUtil.c
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./Common.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./Common.o ./Common.c
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./NodeFactory.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./NodeFactory.o ./NodeFactory.c
cc \
 -Wl,DLL -g -Wl,p,map -Wl,LIST=ALL,MAP,XREF,REUS=RENT \
 -Wl,xplink \
 -o ./SwitchNode.lil ./SwitchNode.o ./BipSampPluginUtil.o \
 ./Common.o ./NodeFactory.o \
 ${install_dir}/lib/libimbdfplg.x
```

Run the following command to set the file permissions of the user-defined
extension to group read and to be executable:

```
chmod a+rx {output_dir}/SwitchNode.lil
```

**Creating a user-defined extension in Java:**

You must complete a series of tasks to create user-defined nodes that use the Java
language.

Complete the appropriate tasks from the following list:
- "Creating an input node in Java" on page 60
- "Creating a message processing or output node in Java" on page 65
- "Compiling a Java user-defined node" on page 74
- "Packaging a Java user-defined node" on page 102

You can write only user-defined nodes in Java: you must write user-defined
parsers in C.

When you have completed this set of tasks, continue with the following tasks:
- "Creating the user interface representation of a user-defined node in the
  workbench" on page 76
- "Testing a user-defined node" on page 84
- "Packaging and distributing user-defined extensions" on page 101

*Restrictions when creating Java nodes:*

In Java user-defined nodes and the JavaCompute node, calling the System.exit(...) method is not supported. Calling this method results in a `SecurityException`.

*Creating an input node in Java:*

An input node is used to receive a message into a message flow, typically from a source that is not supported by the built-in input nodes.

**Before you start**

Read the following topics:
- "Why use a user-defined extension?" on page 4
- "User-defined input nodes" on page 12

To create an input node in the Java language:
1. "Creating a Java project"
2. "Declaring the input node class"
3. "Defining the node constructor" on page 61
4. "Receiving external data into a buffer" on page 61
5. "Propagating the message" on page 62
6. "Controlling threading and transactionality" on page 62
7. "Declaring the node name" on page 63
8. "Declaring attributes" on page 64
9. "Implementing the node functionality" on page 64
10. "Overriding default message parser attributes (optional)" on page 64
11. "Deleting an instance of the node" on page 65

A Java user-defined node is distributed as a `.jar` file.

*Creating a Java project:*

Before you can create Java nodes in the workbench, you must create a Java project:
1. Switch to the Java perspective.
2. Click **File** → **New** → **Project**. Select **Java** from the left menu, and then select **Java Project** from the right menu.
3. Give the project a name. The Java Settings panel is displayed.
4. Select the **Libraries** tab, and click **Add External JARs**.
5. Select *install_dir*`\classes\jplugin2.jar`
6. Follow the prompts on the other tabs to define any other build settings.
7. Click **Finish**.

You can now develop the source for your Java node in this project.

*Declaring the input node class:*

Every class that implements MbInputNodeInterface and is contained in the broker LIL path is registered with the broker as an input node. When you implement MbInputNodeInterface, you must also implement a run method for this class. The run method represents the start of the message flow, contains the data that

formulates the message, and propagates it down the flow. The broker calls the run method when threads become available in accordance with your specified threading model.

For example, to declare the input node class:

```
package com.ibm.jplugins;

import com.ibm.broker.plugin.*;

public class BasicInputNode extends MbInputNode implements MbInputNodeInterface
{
...
```

Follow these steps to complete this action in the workbench:

1. Click **File → New → Class**.
2. Set the package and class name fields to appropriate values.
3. Delete the text in the **Superclass** text field and click the **Browse** button.
4. Select **MbInputNode**.
5. Click the **Add** button next to Interfaces text field, and select **MbInputNodeInterface**.
6. Click **Finish**.

*Defining the node constructor:*

When the node is instantiated, the constructor of the node class is called. This class is where you create the terminals of the node, and initialize the default values for the attributes.

An input node has a number of output terminals associated with it, but does not typically have any input terminals. Use the createOutputTerminal method to add output terminals to a node when the node is instantiated. For example, to create a node with three output terminals:

```
public BasicInputNode() throws MbException
{
 createOutputTerminal ("out");
 createOutputTerminal ("failure");
 createOutputTerminal ("catch");
 setAttribute ("firstParserClassName","myParser");
 attributeVariable  = "none";
}
```

*Receiving external data into a buffer:*

An input node can receive data from any type of external source, such as a file system, a queue, or a database, in the same way as all other Java programs, if the output from the node is in the correct format.

Provide an input buffer (or bit stream) to contain input data, and associate it with a message object. Create a message from a byte array by using the createMessage method of the MbInputNode class, and then generate a valid message assembly from this message. For example, to read the input data from a file:

1. Create an input stream to read from the file:

   ```
   FileInputStream inputStream = new FileInputStream("myfile.msg");
   ```

2. Create a byte array the size of the input file:

   ```
   byte[] buffer = new byte[inputStream.available()];
   ```

3. Read from the file into the byte array:

   `inputStream.read(buffer);`

4. Close the input stream:

   `inputStream.close();`

5. Create a message to put on the queue:

   `MbMessage msg = createMessage(buffer);`

6. Create a message assembly to hold this message:

   ```
   msg.finalizeMessage(MbMessage.FINALIZE_VALIDATE);
   MbMessageAssembly newAssembly =
         new MbMessageAssembly(assembly, msg);
   ```

*Propagating the message:*

When you have created a message assembly, you can propagate it to one of the output terminals that you have defined on the node.

For example, to propagate the message assembly to the terminal named **out**:

```
MbOutputTerminal out = getOutputTerminal("out");
out.propagate(newAssembly);
```

To delete the message:

`msg.clearMessage();`

To clear the memory that is allocated for the message tree, call the clearMessage() function within the finally block of `try/catch`.

*Controlling threading and transactionality:*

The broker infrastructure handles transaction issues such as controlling the commit of a WebSphere MQ or database unit of work when message processing has completed. However, resources modified from within a user-defined node are not necessarily under the transactional control of the broker.

Each message flow thread is allocated from a pool of threads maintained for each message flow, and starts in the run method.

The user-defined node uses return values to indicate whether a transaction has been successful, to control whether transactions are committed or rolled back, and to control when the thread is returned to the pool. The broker infrastructure catches all unhandled exceptions, and rolls back the transaction.

You determine the behavior of transactions and threads by using the appropriate return value:

**MbInputNode.SUCCESS_CONTINUE**
> The transaction is committed and the broker calls the run method again using the same thread.

**MbInputNode.SUCCESS_RETURN**
> The transaction is committed and the thread is returned to the thread pool, assuming that it is not the only thread for this message flow.

**MbInputNode.FAILURE_CONTINUE**
> The transaction is rolled back and the broker calls the run method again using the same thread.

**MbInputNode.FAILURE_RETURN**
> The transaction is rolled back and the thread is returned to the thread pool, assuming that it is not the only thread for this message flow.

**MbInputNode.TIMEOUT**
> The run method must not block indefinitely while waiting for input data to arrive. While the flow is blocked by user code, you cannot shut down or reconfigure the broker. The run method must yield control to the broker periodically by returning from the run method. If input data has not been received after a certain period (for example, 5 seconds), the method must return with the TIMEOUT return code. Assuming that the broker does not need to reconfigure or shut down, the run method of the input node gets called again immediately.

To create multithreaded message flows, you call the dispatchThread method after a message has been created, but before the message is propagated to an output terminal. This action ensures that only one thread is waiting for data while other threads are processing the message. New threads are obtained from the thread pool up to the maximum limit specified by the Additional Instances property of the message flow. For example:

```
public int run( MbMessageAssembly assembly ) throws MbException
{
  byte[] data = getDataWithTimeout();  // user supplied method
                              // returns null if timeout

  if( data == null )
    return TIMEOUT;

  MbMessage msg = createMessage( data );
  msg.finalizeMessage( MbMessage.FINALIZE_VALIDATE );
  MbMessageAssembly newAssembly =
      new MbMessageAssembly( assembly, msg );

  dispatchThread();

  getOutputTerminal( "out" ).propagate( newAssembly );

  return SUCCESS_RETURN;
}
```

*Declaring the node name:*

You must declare the name of the node for use and identification by the workbench. All node names must end with the characters "Node". Declare the name by using the following method:

```
public static String getNodeName()
{
    return "BasicInputNode";
}
```

If this method is not declared, the Java API framework creates a default node name by using the following rules:

- The class name is appended to the package name.
- The periods are removed, and the first letter of each part of the package and class name is capitalized.

For example, by default, the following class is assigned the node name "ComIbmPluginsamplesBasicInputNode":

```
package com.ibm.pluginsamples;
public class BasicInputNode extends MbInputNode implements MbInputNodeInterface
{
   ...
```

*Declaring attributes:*

Declare node attributes using the same method that you use for Java bean
properties. You are responsible for writing get and set methods for the attributes;
the API framework infers the attribute names by using the Java bean introspection
rules. For example, if you declare the following two methods:

```
private String attributeVariable;

public String getFirstAttribute()
{
  return attributeVariable;
}

publc void setFirstAttribute(String value)
{
  attributeVariable = value;
}
```

The broker infers that this node has an attribute called firstAttribute. This name is
derived from the names of the get or set methods, not from the variable names of
any internal class members. Attributes can be exposed only as strings, so convert
numeric types to and from strings in the get or set methods. For example, the
following method defines an attribute called timeInSeconds:

```
int seconds;

public String getTimeInSeconds()
{
  return Integer.toString(seconds);
}

public void setTimeInSeconds(String value)
{
  seconds = Integer.parseInt(value);
}
```

*Implementing the node functionality:*

As already described, the run method is called by the broker to create the input
message. This method must provide all the processing function for the input node.

*Overriding default message parser attributes (optional):*

An input node implementation normally determines which message parser initially
parses an input message. For example, the built-in MQInput node dictates that an
MQMD parser is required to parse the MQMD header. A user-defined input node
can select an appropriate header or message parser, and the mode in which the
parsing is controlled, by using the following default attributes that are included,
which you can override:

**rootParserClassName**
>        Defines the name of the root parser that parses message formats supported
>        by the user-defined input node. It defaults to GenericRoot, a supplied root
>        parser that causes the broker to allocate and chain parsers together. It is
>        unlikely that a node would have to modify this attribute value.

**firstParserClassName**
> Defines the name of the first parser, in what might be a chain of parsers that are responsible for parsing the bitstream. It defaults to XML.

**messageDomainProperty**
> An optional attribute that defines the name of the message parser required to parse the input message. The supported values are the same as the values that are supported by the MQInput node.

**messageSetProperty**
> An optional attribute that defines the message set identifier, or the message set name, in the `Message Set` field, only if the MRM parser was specified by the `messageDomainProperty` attribute.

**messageTypeProperty**
> An optional attribute that defines the identifier of the message in the `MessageType` field, only if the MRM parser was specified by the `messageDomainProperty` attribute.

**messageFormatProperty**
> An optional attribute that defines the format of the message in the `Message Format` field, only if the MRM parser was specified by the `messageDomainProperty` attribute.

*Deleting an instance of the node:*

An instance of the node is deleted when either:
- You shut down the broker.
- You remove the node or the message flow that contains the node, and redeploy the configuration.

When the node is deleted, it can perform cleanup operations, such as closing sockets, if it implements the optional onDelete method. This method, if present, is called by the broker just before the node is deleted.

Implement the onDelete method as follows:

```
public void onDelete()
{
  // perform node cleanup if necessary
}
```

*Creating a message processing or output node in Java:*

A message processing node is used to process a message, and an output node is used to output a message as a bit stream.

**Before you start**

Read the following topics:
- "Why use a user-defined extension?" on page 4
- "User-defined message processing nodes" on page 17
- "User-defined output nodes" on page 25
- "Restrictions when creating Java nodes" on page 59

WebSphere Message Broker provides the source for two sample user-defined nodes called SwitchNode and TransformNode. You can use these nodes in their current state, or you can modify them.

When you code a message processing node or an output node, the two types provide essentially the same functions. You can perform message processing within an output node, and likewise you can propagate an output message to a bit stream from a message processing node. For simplicity, this topic refers mainly to the node as a message processing node, but it does discuss the functionality of both types of node.

Complete the following steps:

A Java user-defined node is distributed as a `.jar` file.

*Creating a new Java project:*

Before you can create Java nodes in the workbench, you must create a new Java project:

1. Switch to the **Java** perspective.
2. Click **File** → **New** → **Project**. Select **Java** from the left menu, and then select **Java Project** from the right menu.
3. Give the project a name.

   The Java Settings panel is displayed.
4. Select the **Libraries** tab, and click **Add External JARs**.
5. Select install_dir\classes\jplugin2.jar.
6. Follow the prompts on the other tabs to define any other build settings.
7. Click **Finish**.

You can now develop the source for your Java node within this project.

*Declaring the message processing node class:*

Any class that implements MbNodeInterface, and is contained in the broker's LIL path, is registered with the broker as a message processing node. When you implement MbNodeInterface, you must also implement an evaluate method for this class. The evaluate method is called by the broker for each message that passes through the flow.

For example, to declare the message processing node class:

```
package com.ibm.jplugins;

import com.ibm.broker.plugin.*;

public class BasicNode extends MbNode implements MbNodeInterface
```

Declare the class in the workbench:

1. Click **File** → **New** → **Class**.
2. Set the package and class name fields to appropriate values.
3. Delete the text in the Superclass text field and click **Browse**.
4. Select **MbNode** and click **OK**.
5. Click the **Add** button next to Interfaces text field, and select **MbNodeInterface**.
6. Click **Finish**.

*Defining the node constructor:*

When the node is instantiated, the constructor of the user's node class is called. Create the terminals of the node, and initialize any default values for attributes in this constructor.

A message processing node has a number of input terminals and output terminals that are associated with it. Use the methods createInputTerminal and createOutputTerminal to add terminals to a node when the node is instantiated.

For example, to create a node with one input terminal and two output terminals:

```
public MyNode() throws MbException
{
  // create terminals here
  createInputTerminal ("in");
  createOutputTerminal ("out");
  createOutputTerminal ("failure");
}
```

*Accessing message data:*

In many cases, the user-defined node needs to access the contents of the message received on its input terminal. The message is represented as a tree of syntax elements. Use the supplied utility function to evaluate methods for message management, message buffer access, syntax element navigation, and syntax element access.

The MbElement class provides the interface to the syntax elements.

For example:

1. To navigate to the relevant syntax element in the XML message:

```
MbElement rootElement = assembly.getMessage().getRootElement();
MbElement switchElement =
rootElement.getLastChild().getFirstChild().getFirstChild();
```

2. To select the terminal indicated by the value of this element:

```
String terminalName;
String elementValue = (String)switchElement.getValue();
if(elementValue.equals("add"))
  terminalName = "add";
else if(elementValue.equals("change"))
  terminalName = "change";
else if(elementValue.equals("delete"))
  terminalName = "delete";
else if(elementValue.equals("hold"))
  terminalName = "hold";
else
  terminalName = "failure";

MbOutputTerminal out = getOutputTerminal(terminalName);
```

*Transforming a message object:*

The received input message is read-only, so before you can transform a message, you must write it to a new output message. You can copy elements from the input message, or you can create new elements in the output message.

The MbMessage class provides the copy constructors, and the methods to get the root element of the message. The MbElement class provides the interface to the syntax elements.

For example, if you have an incoming message assembly with embedded messages, you could have the following code in the evaluate method of your user-defined node:

1. To create a new copy of the message assembly and its embedded messages:
   ```
   MbMessage newMsg = new MbMessage(assembly.getMessage());
   MbMessageAssembly newAssembly = new MbMessageAssembly(assembly, newMsg);
   ```

2. To navigate to the relevant syntax element in the XML message:
   ```
   MbElement rootElement = newAssembly.getMessage().getRootElement();
   MbElement switchElement =
   rootElement.getFirstElementByPath("/XML/data/action");
   ```

3. To change the value of an existing element:
   ```
   String elementValue = (String)switchElement.getValue();
     if(elementValue.equals("add"))
       switchElement.setValue("change");
     else if(elementValue.equals("change"))
       switchElement.setValue("delete");
     else if(elementValue.equals("delete"))
       switchElement.setValue("hold");
     else
       switchElement.setValue("failure");
   ```

4. To add a new tag as a child of the switch tag:
   ```
   MbElement tag = switchElement.createElementAsLastChild(MbElement.TYPE_NAME,
                                                  "PreviousValue",
                                                  elementValue);
   ```

5. To add an attribute to this new tag:
   ```
   tag.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE,
                             "NewValue",
                             switchElement.getValue());

   MbOutputTerminal out = getOutputTerminal("out");
   ```

As part of the transformation, you might need to create a new message body. To create a new message body, use one of the following methods, which specifically assigns a parser to a message tree folder:

```
createElementAfter(String)
createElementAsFirstChild(String)
createElementAsLastChild(String)
createElementBefore(String)
createElementAsLastChildFromBitstream(byte[], String, String, String, String, int, int, int)
```

Do not use the following methods, which do not associate an owning parser with the folder:

```
createElementAfter(int)
createElementAfter(int, String, Object)
createElementAsFirstChild(int)
createElementAsFirstChild(int, String, Object)
```

```
createElementAsLastChild(int)
createElementAsLastChild(int, String, Object)
createElementBefore(int)
createElementBefore(int, String, Object)
```

*Propagating the message:*

Before you propagate a message, decide what message flow data you want to propagate, and whether to propagate to a node terminal, or to a Label node.

For example:
1. To propagate the message to the output terminal "out":
   ```
   MbOutputTerminal out = getOutputTerminal("out");
           out.propagate(newAssembly);
   ```
2. To propagate the message to a Label node:
   ```
   MbRoute label1 = getRoute ("label1");
   Label1.propagate(newAssembly);
   ```

Call the clearMessage() function within the finally block of try/catch to clear the memory that is allocated for the message tree.

To propagate the same MbMessage object multiple times, call the finalizeMessage() method on the MBMessage object, so that any changes made to the message are reflected in the bit stream that is generated downstream of the Java node; for example:
```
MbMessage newMsg = new MbMessage(assembly.getMessage());
MbMessageAssembly newAssembly = new MbMessageAssembly(assembly, newMsg);
...
newMsg.finalizeMessage(MbMessage.FINALIZE_NONE);
out.propagate(newAssembly;
...
newMsg.finalizeMessage(MbMessage.FINALIZE_NONE);
out.propagate(newAssembly);
```

*Declaring the node name:*

The name of the node must be the same as the one that is used in the workbench. All node names must end with "Node". Declare the name using the following method:
```
public static String getNodeName()
{
    return "BasicNode";
}
```

If this method is not declared, the Java API framework creates a default node name using the following rules:
- The class name is appended to the package name.
- The dots are removed, and the first letter of each part of the package and class name are capitalized.

For example, by default, the following class is assigned the node name "ComIbmPluginsamplesBasicNode":
```
package com.ibm.pluginsamples;
public class BasicNode extends MbNode implements MbNodeInterface
{
    ...
```

*Declaring attributes:*

Declare node attributes in the same way as Java Bean properties. You must write getter and setter methods for the attributes. The API framework infers the attribute names using the Java Bean introspection rules. For example, if you declare the following two methods:

```
private String attributeVariable;

public String getFirstAttribute()
{
  return attributeVariable;
}

publc void setFirstAttribute(String value)
{
  attributeVariable = value;
}
```

the broker infers that this node has an attribute called firstAttribute. This name is derived from the names of the get or set methods, not from any internal class member variable names. Attributes can only be exposed as strings, therefore, you must convert any numeric types to and from strings in the get or set methods. For example, the following method defines an attribute called timeInSeconds:

```
int seconds;

public String getTimeInSeconds()
{
  return Integer.toString(seconds);
}

public void setTimeInSeconds(String value)
{
  seconds = Integer.parseInt(value);
}
```

*Implementing the node functionality:*

The evaluate method, defined in MbNodeInterface, is called by the broker to process the message. All the processing function for the node is included in this method.

The evaluate method has two parameters that are passed in by the broker:
1. The MbMessageAssembly, which contains the following objects that are accessed using the appropriate methods:
   - The incoming message
   - The LocalEnvironment
   - The global Environment
   - The ExceptionList
2. The input terminal on which the message has arrived.

For example, the following code extract shows how you might write the evaluate method:

```
public void evaluate(MbMessageAssembly assembly, MbInputTerminal inTerm) throws MbException
  {
    // add message processing code here

    getOutputTerminal("out").propagate(assembly);
  }
```

The message flow data, which consists of the message, Environment, LocalEnvironment, and ExceptionList, is received at the input terminal of the node.

*Deleting an instance of the node:*

An instance of the node is deleted when either:
- You shut down the broker.
- You remove the node or the message flow that contains the node, and redeploy the configuration.

If you want the node to perform any clean up operations, for example closing sockets, include an implementation of the onDelete method:

```
public void onDelete()
{
  // perform node cleanup if necessary
}
```

This method is called by the broker immediately before it deletes the node.

*Extending the capability of a Java message processing or output node:*

Within a message processing or output node, you can add extended functions to your Java node.

**Before you start**

Read "Creating a message processing or output node in Java" on page 65.

You can add one or more of the following functions:
- "Accessing ESQL"
- "Interacting with databases" on page 72
- "Handling exceptions" on page 72
- "Writing to an output device" on page 73

*Accessing ESQL:*

Nodes can invoke ESQL expressions using Compute node ESQL syntax. You can create and modify the components of the message using ESQL expressions, and you can refer to elements of both the input message and data from an external database.

The following procedure demonstrates how to use ESQL to control transactions from the evaluate method in your user-defined node:

1. Set the name of the ODBC data source to use. For example:

   ```
   String dataSourceName = "myDataSource";
   ```

2. Set the ESQL statement to run:

   ```
   String statement =
     "SET OutputRoot.XMLNS.data =
           (SELECT Field2 FROM Database.Table1 WHERE Field1 = 1);";
   ```

   Or, if you want to run a statement that returns no result:

   ```
   String statement = "PASSTHRU(
                         'INSERT INTO Database.Table1 VALUES(
                               InputRoot.XMLNS.DataField1,
                               InputRoot.XMLNS.DataField2)');";
   ```

3. Select the transaction you want from the following types:

**MbSQLStatement.SQL_TRANSACTION_COMMIT**
Immediately commit the transaction after the ESQL statement has completed.

**MbSQLStatement.SQL_TRANSACTION_AUTO**
Commit the transaction when the message flow has completed. (Rollbacks are performed if necessary.)

For example:

```
int transactionType = MbSQLStatement.SQL_TRANSACTION_AUTO;
```

4. Get the ESQL statement. For example:

```
MbSQLStatement sql =
        createSQLStatement(dataSourceName, statement, transactionType);
```

You can use the method createSQLStatement(data source, statement to default the transaction type to MbSQLStatement.SQL_TRANSACTION_AUTO).

5. Create the new message assembly to be propagated:

```
MbMessageAssembly newAssembly =
        new MbMessageAssembly(assembly, assembly.getMessage());
```

6. Run the ESQL statement:

```
sql.select(assembly, newAssembly);
```

Or, if you want to run an ESQL statement that returns no result:

```
sql.execute(assembly);
```

*Interacting with databases:*

You can interact with databases from the Java code in your message processing node. The support that is provided is identical to the support for Java code that you write for the JavaCompute node; for details of the available options, and the advantages and restrictions that apply, see Interacting with databases by using the JavaCompute node.

*Handling exceptions:*

Use the MbException class to catch and access exceptions. The MbException class returns an array of exception objects that represent the children of an exception in the broker exception list. Each element returned specifies its exception type. An empty array is returned if an exception has no children. The following code sample shows an example of how you might use the MbException class in the evaluate method of your user-defined node.

```
public void evaluate(MbMessageAssembly assembly, MbInputTerminal inTerm) throws MbException
  {
    try
      {

        // plug-in functionality

      }
    catch(MbException ex)
      {
        traverse(ex, 0);

        throw ex; // if re-throwing, it must be the original exception that was caught
      }
  }

  void traverse(MbException ex, int level)
  {
```

```
   if(ex != null)
     {
       // Do whatever action here
       System.out.println("Level: " + level);
       System.out.println(ex.toString());
       System.out.println("traceText:  " + ex.getTraceText());

       // traverse the hierarchy
       MbException e[] = ex.getNestedExceptions();
       int size = e.length;
       for(int i = 0; i < size; i++)
         {
           traverse(e[i], level + 1);
         }
     }
  }
```

You can develop a user-defined message-processing or output node in such a way that it can access all current exceptions. For example, to catch database exceptions you can use the MbSQLStatement class. This class sets the value of the 'throwExceptionOnDatabaseError' attribute, which determines broker behavior when it encounters a database error. When it is set to true, if an exception is thrown it can be caught and handled by the evaluate method in your user-defined extension.

The following code sample shows an example of how to use the MbSQLStatement class.

```
public void evaluate(MbMessageAssembly assembly, MbInputTerminal inTerm) throws MbException
  {
    MbMessage newMsg = new MbMessage(assembly.getMessage());
    MbMessageAssembly newAssembly = new MbMessageAssembly(assembly, newMsg);

    String table =
        assembly.getMessage().getRootElement().getLastChild().getFirstChild().getName();

    MbSQLStatement state = createSQLStatement( "dbName",
        "SET OutputRoot.XMLNS.integer[] = PASSTHRU('SELECT * FROM " + table + "');" );

    state.setThrowExceptionOnDatabaseError(false);
    state.setTreatWarningsAsErrors(true);

    state.select( assembly, newAssembly );

    int sqlCode = state.getSQLCode();
    if(sqlCode != 0)
      {
        // Do error handling here

        System.out.println("sqlCode = " + sqlCode);
        System.out.println("sqlNativeError = " + state.getSQLNativeError());
        System.out.println("sqlState = " + state.getSQLState());
        System.out.println("sqlErrorText = " + state.getSQLErrorText());
      }

    getOutputTerminal("out").propagate(newAssembly);
  }
```

*Writing to an output device:*

To write to an output device, the logical (hierarchical) message must be converted back into a bit stream in your evaluate method. Use the getBuffer method in MbMessage to perform this task:

```
                  public void evaluate( MbMessageAssembly assembly, MbInputTerminal in)
                                                         throws MbException
{
  MbMessage msg = assembly.getMessage();
  byte[] bitstream = msg.getBuffer();

  // write the bitstream out somewhere
  writeBitstream( bitstream );   // user method

}
```

Typically, for an output node the message is not propagated to any output
terminal, therefore you can just return at this point.

You must use the supplied MQOutput node when writing to WebSphere MQ
queues, because the broker internally maintains a WebSphere MQ connection and
the open queue handles on a thread-by-thread basis. These handles are cached to
optimize performance. In addition, the broker handles exception scenarios when
certain WebSphere MQ events occur, and this recovery is adversely affected if
WebSphere MQ MQI calls are used in a user-defined output node.

*Getting and setting the specific type of an Mb element:*

Two methods are provided for handling the specific type of an Mb syntax element.

The following methods are available:
- getSpecificType
- setSpecificType

Use these methods to access or set the specific type of an XML element. For
example, to update the current value:

1. Call getSpecificType on the syntax element.

   The getSpecificType method does not take any parameters, but returns the
   specific type of the element as an int value.

2. Call setSpecificType on the syntax element.

   The setSpecificType method takes one parameter of the type int, which is the
   specific type that you want the Mb element to be. This method has no return
   value.

Specific type values for the XML and MRM parsers are listed in "XML, MRM, and
XMLNSC parser constants" on page 306.

*Compiling a Java user-defined node:*

When you have created the code for your Java user-defined node, you must
compile it for your operating system.

**Before you start**

You must have a user-defined node written in Java. This node can be one of the
provided sample nodes that are described in "Sample node files" on page 121, or a
node that you have created yourself by using the instructions in either "Creating a
message processing or output node in Java" on page 65 or "Creating an input node
in Java" on page 60.

You can compile a Java user-defined node either from the command line, or from within the project in the workbench. Both options are described later in this section.

When you compile a Java user-defined node from the command line, you must have a compatible IBM Software Developer Kit for Java on the current operating system. For details of supported Java versions, see Additional software requirements.

*Compiling a Java user-defined node from the workbench:*

Use the following procedure to compile your Java user-defined node from the workbench:

1. Switch to the Java Development Perspective.
2. In the Package Explorer, select the `/src` directory inside your node project, and click **File** → **Export**.
3. From the list displayed, select **JAR file**. Click **Next**. The resources that are available for you to export as a JAR file are listed.
4. Verify that **Export generated class files and resources** is selected.
5. Specify a name and location for your JAR file. Place the file inside the root directory of your node project, and give the file the same name as the name of the project (with a `.jar` extension). You can use the default values for the rest of the options. Click **Finish**.

The `.jar` file that you have created is displayed in your node project, and is ready for you to install on one or more brokers (see "Installing user-defined extension runtime files on a broker" on page 106), or to package for distribution (see "Packaging a user-defined node workbench project" on page 107).

*Compiling a Java user-defined node from the command line:*

Use the following procedure to compile your Java user-defined node from the command line:

1. Add the location of `jplugin2.jar` to the CLASSPATH for your current platform:

   `Windows`    `install_dir\classes\jplugin2.jar`

   `Linux`    `install_dir/classes/jplugin2.jar`

   `UNIX`    `install_dir/classes/jplugin2.jar`

   `z/OS`    `install_dir/classes/jplugin2.jar`

2. Put your Java user-defined node class into the following directory:

   `Windows`    `install_dir\sample\extensions\nodes`

   `Linux`    `install_dir/sample/extensions/nodes`

   `UNIX`    `install_dir/sample/extensions/nodes`

   `z/OS`    `install_dir/sample/extensions/nodes`

3. Change to the directory that now contains your user-defined node class.
4. Compile the `.java` file by using the javac command; for example:

   `javac nodename.java`

5. Package the resulting `.class` file into a `.par` file. See "Packaging a Java user-defined node" on page 102.

The .par file that you have created is ready for you to install on one or more brokers (see "Installing user-defined extension runtime files on a broker" on page 106), or to package for distribution (see "Packaging a user-defined node workbench project" on page 107).

**Creating the user interface representation of a user-defined node in the workbench:**

When you are developing a user-defined node, you must create the user interface representation of the node in the workbench.

**Before you start**:
• Perform the steps in "Developing user-defined extensions" on page 3.

The following topics describe the steps that you must complete:
1. "Creating a user-defined node project"
2. "Creating a user-defined node in the workbench" on page 77

For user-defined parsers, install the compiled .lil file. You do not manipulate parsers from within the workbench and therefore you do not create a user interface representation of user-defined parsers.

When you have created the workbench representation, test your user-defined node, see "Testing a user-defined node" on page 84.

*Creating a user-defined node project:*

When you create user-defined nodes, you must first create a user-defined node project to contain the nodes and their supporting files.

To create a new project for your user-defined node, perform the following steps:
1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **Project**, then **Message Flow Node Development** → **Message Flow Plug-in Node Project**. Click **Next** to start the Message Flow Plug-in Node Project wizard. The New Message Flow Plug-in Node Project window opens.
3. Specify the name of the category for the nodes that you are creating. The node is displayed under this category in the message flow node palette. Either choose the name of an existing category, or enter a name to create a new category. Click **Next**. The New Plug-in Project window opens.
4. Specify a name for your project. To be consistent with the supplied nodes, and to avoid conflict with the names of node projects that are supplied by other independent software vendors, use your Internet domain name for your organization as part of the name. For example, the name must be of the form *com.xyz.nodegroup*, where *com.xyz* is the company Internet domain name.

   Do not enter a blank space or an underscore character in your project name. These characters are not valid. You can save any number of nodes in a single project.

   If the Message Broker Toolkit is at a previous version, set the Eclipse version to 3.0 before you deploy this node plug-in project. You have now created a name for your project. Click **Next**.

   The Plug-in content panel of the New Plug-in Project window opens.

   Ensure that the **Plug-in ID** field does not contain an underscore character. This character is not valid.

5. Accept all default values and click **Next**. The Templates panel of the New Plug-in Project window opens.

6. Click **Finish**.

7. If warnings are displayed in the Problems view, that are associated with the newly created project, perform the following steps:

   a. Click **Window** → **Preferences**.

   b. Expand **Plug-In Development** and click **Target Platform**.

   c. Click **Add required Plug-ins** to select all loaded plug-ins and click **OK**.

   d. Select your user-defined node project in the Package Explorer view, and click **Project** → **Clean**. A dialog box opens in which you can select either of the following options:

      - **Clean all projects**.
      - **Clean projects selected below**. If you choose this option, select the projects that you want to clean.

A project folder containing all the supporting files that are required for your user-defined node is displayed in the Package Explorer view. The project is stored in the default workspace directory.

*Creating a user-defined node in the workbench:*

Create the representation of a user-defined node in the workbench.

**Before you start:**

Complete the following task: "Creating a user-defined node project" on page 76.

To create the visual representation of your user-defined node in the workbench, complete the following tasks:

1. "Creating the user-defined node plug-in files"

2. "Defining the node properties" on page 78

3. Optional: "Adding help to the node" on page 81

4. Optional: "Creating node icons" on page 82

5. Optional: "Adding a property editor or compiler" on page 83

When you have created the representation of the node, you cannot move it to another folder.

*Creating the user-defined node plug-in files:*

Create the files that contain the message processing logic.

**Before you start:**

Complete the task following task: "Creating a user-defined node project" on page 76.

1. Switch to the Broker Application Development perspective.

2. Launch the wizard by clicking **File** → **New** → **Other**. The **New** window opens.

3. Expand **Message Flow Node Development**, select **Message Flow Plug-in Node**. Click **Next**. The **New Message Flow Plug-in Node** window opens.

4. Select the parent folder for the node from the list of names that are displayed. This folder is the project that you created to contain this node.

5. Specify a file name for the node. The file name must be the name of the node, excluding the `Node` at the end. For example, if you have created a node called `BasicNode`, the file name must be `Basic`.

6. Click **Finish**. A `.msgnode` file for the new node is created and is added to the project in the Broker Development view. The `.msgnode` file is opened in the Message Node editor.

**Next:**

When you have completed this task, define the node properties, see "Defining the node properties."

*Defining the node properties:*

Define the properties for a user-defined node, and add input and output terminals so that you can connect it to other nodes in a message flow.

**Before you start:**

Complete the following tasks:
1. "Creating a user-defined node project" on page 76
2. "Creating the user-defined node plug-in files" on page 77

When you complete the task described in "Creating the user-defined node plug-in files" on page 77, a `.msgnode` file is created for the new node, and is opened in the Message Node editor of the Broker Application Development perspective. You can now add terminals and properties to the node.

*Adding terminals to the node:*

1. If the Terminals page is not already displayed, click the **Terminals** tab at the bottom of the Message Node editor.

2. Click **Add** to the right of the **In Terminals** or **Out Terminals** fields to add an input or output terminal.

   You must define at least one input terminal, but output terminals are optional.

3. To rename a terminal, click the terminal name so that it is highlighted and shows a flashing cursor after the name, and enter a new name.

4. If your node supports dynamic input or output terminals, select the appropriate check box.

   Dynamic terminals are terminals that you can add to certain nodes after you have added them to a message flow in the Message Flow editor. For more information, see Message flow node terminals.

*Defining properties for the node:*

1. Click the **Properties** tab at the bottom of the Message Node editor.

   On the Properties page, you can set the properties for the node, for example, a database name, a host server name, or a password. The properties that you set here must match the properties that you specified in the user-defined node itself by using the get and set methods.

2. If the node is an input node, click the node name in the hierarchy to highlight it, and select **Input node**. Select **Use broker default values** if you want the node to initialize with the default values for the broker.

3. By default, all properties are grouped under the Basic group. You can add new groups in which to place properties. When your custom node is selected in the

workbench, each group of properties is rendered as a separate tab in the Properties view. To create additional groups of properties, click **Add Property Group**  .

4. To add a simple property, click the name of a property group in the hierarchy

   to highlight it, and click **Add Simple Property** 

   The new property is added to the hierarchy as a child of the property group. Its name is highlighted so that you can change it. A number of fields are displayed in the Details section, where you can configure the property.

   a. Select the correct attribute type: one of the built-in types, or a type to match the list of values that the property can have.

   b. Enter any default values, which are shown in the Properties view when the node is included in a message flow.

   c. Specify the location of these resources in the relevant field to generate a property editor or a compiler. Use the IPropertyEditor interface for your property editor; for more details, see Property editor API.

   d. Use the IPropertyCompiler interface to create a custom compiler; see Property editor API.

   e. Specify the system property for each attribute that you define:

      **Hidden**
      The property is not displayed in the Properties view or the Promote Property dialog box.

      **Read only**
      The property is displayed, but cannot be changed.

      **Mandatory**
      A value is required. The field cannot be left blank. Boolean and enum properties are always mandatory.

      **Configurable**
      The property can be configured at deployment time

5. To add a table property, select the name of a property group in the Properties

   view and click **Add Table Property**  .

   In addition to simple properties, a node can also have complex properties. A table property represents a repeating property of a complex type. The new property is added to the hierarchy as a child of the property group. A number of fields are displayed in the Details section where you can configure the table property.

6. To add a column to an existing table, select the name of the table property in

   the hierarchy, and click **Add Simple Property**  .

   For example, the following figure shows the Property Hierarchy of the usernode, where Filter and Route columns have been added.
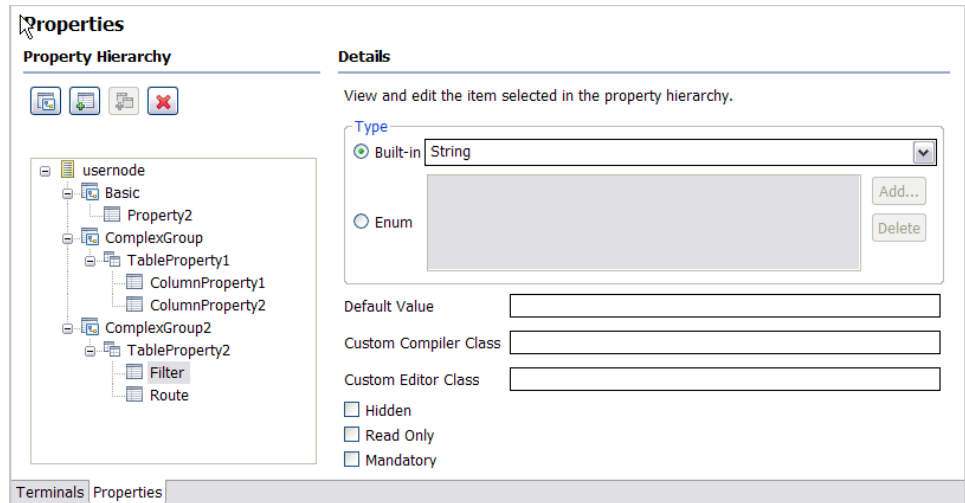
Properties

**Property Hierarchy**

**Details**

View and edit the item selected in the property hierarchy.

- usernode
  - Basic
    - Property2
  - ComplexGroup
    - TableProperty1
      - ColumnProperty1
      - ColumnProperty2
  - ComplexGroup2
    - TableProperty2
      - Filter
      - Route

Type
- ⦿ Built-in  String
- ◯ Enum

Add...
Delete

Default Value
Custom Compiler Class
Custom Editor Class
☐ Hidden
☐ Read Only
☐ Mandatory

Terminals   Properties

A number of fields are displayed in the Details section where you can configure the properties of the column. Define table columns, where each column is assigned to a type.

a. Select the correct attribute type in the **Type** field: one of the built-in types, or enumeration.

b. Enter the default value, in the **Default Value** field. This value is shown in the Properties view when the node is included in a message flow.

c. Specify a qualified class name in the **Custom Compiler Class** field for a property compiler. To create a custom compiler, use the IColumnPropertyCompiler interface. For more information about custom property editors and property compilers, see "Adding a property editor or compiler" on page 83.

d. Specify a qualified class name in the **Custom Editor Class** field to generate a custom property editor. The property editor specified in this field implements the IColumnPropertyEditor interface responsible for cell editing behavior. Leaving the **Custom Editor Class** field blank means that a property editor matching-column type is used. Specify your own IColumnPropertyEditor only if you need custom cell editing behavior.

e. Specify the following attributes for each column that you define:

   **Hidden**
   > Use a hidden column when you want to store, for each row, metadata that is not being displayed.

   **Read only**
   > The column is displayed, but cannot be changed.

   **Mandatory**
   > A value is required. The field cannot be left blank. Boolean and enum properties are always mandatory.

Leave the **Custom Editor Class** field of the Details section of a table property blank, unless you want to overwrite the behavior of the entire table. For example, if the table becomes disabled in response to a change in another property editor.

The following figure shows how the Table properties are rendered as a table in the Properties view, where you can add, edit, and delete values, and change the order of the values in the table.
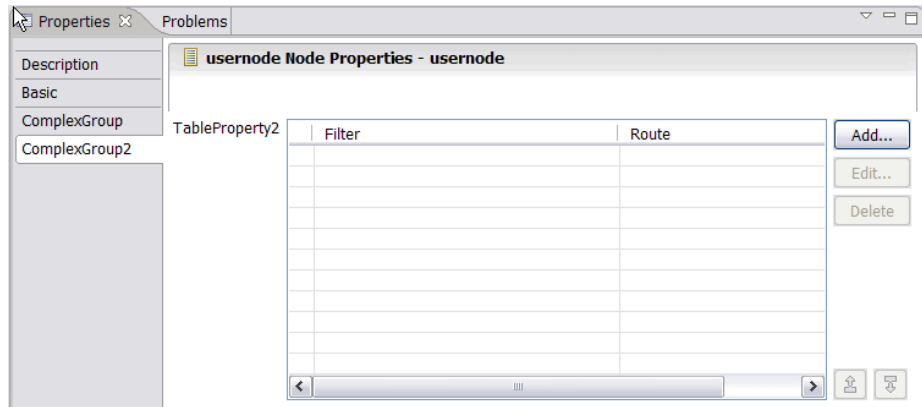
7. Optional: Drag the properties in the properties hierarchy to change the order in which they are listed on the properties page.

8. Close the *nodename*.msgnode file.

9. Optional: You can customize the text that is displayed in the node properties view for each property. To set the text, open the *nodename*.properties file and edit the line:

```
Property.propertyName = your descriptive text
```

**Next:**

The following tasks are optional:
* "Adding help to the node"
* "Creating node icons" on page 82
* "Adding a property editor or compiler" on page 83

You can now test your node, see "Testing a user-defined node" on page 84.

*Adding help to the node:*

Add help information by adding an HTML file to the node that you have created.

**Before you start:**

Complete the following tasks:
1. "Creating a user-defined node project" on page 76
2. "Creating the user-defined node plug-in files" on page 77
3. "Defining the node properties" on page 78

Add help information for the node that you have created to explain why and when to use the node, and how it must be configured:
* Topic information that is displayed within the information center.
* Context-sensitive help that is displayed when you press F1.
* Hover help that is displayed when you hold the mouse pointer over the node.

All three forms of help are optional; you can create any one or more of the three resources described in the following section.

1. Create a `help.html` file within the project to contain the online help that explains what the node does and how you can use it. If you have several files, create a separate doc subdirectory in the plug-in project, and store the online help files in that directory.

   You can make the online help for the node look like it is integrated with the product-supplied information center, under the leaf node called "User-defined nodes", which you can find in **Reference** → **Message flows**. To make the online help for your node show at that point:

   a. Modify the `plugin.xml` file to include the following extension point to the information center:

      ```
      <extension point="org.eclipse.help.toc">
      <toc file="toc.xml"/>
      </extension>
      ```

   b. Create a `toc.xml` file in your user-defined node project, and modify the `link_to` attribute to link to the "UDNodes" anchor that is already defined in the information center contents views:

      ```
      <toc label="My Plugin Node" topic="my_node.htm"
       link_to="../com.ibm.etools.mft.doc/toc.xml#UDNodes">
      <topic label="Mytopic 1" href="topic1.htm>
      </toc>
      ```

      Your help topic is now displayed in the contents views under **Reference** → **Message flows** → **User-defined nodes**.

      The sample nodes that are provided with the product demonstrate this option.

      For further explanation of extension points and how to use them, see the PDE Guide.

2. Add context sensitive (F1) help to the node. Context-sensitive help is displayed when you click a node in the Broker Application Development perspective and press F1.

   When a node is created, a `HelpContexts.xml` file is created. This file assigns a context identifier based on the name of the node. Modify the `HelpContexts.xml` file for your node by changing the text in the description field. The name of the `HelpContexts.xml` file must be unique within the project, but can contain multiple context entries; for example, if you have several nodes within a single project, each node can have its context-sensitive help in the file.

   Context-sensitive help is limited in length. A useful way of providing more help to the user is to link from the F1 help to an HTML file that contains further information; for example, to the online help for the node, described previously. Use the following code for the link:

   ```
   <topic href"../plug-in directory/html file" label="Link title">
   ```

3. Add hover help (known as ToolTip help on Windows) to the node. When you create a user-defined node, a `palette.properties` file is created. Modify this file to contain the hover help for your node, which shows the node name when the palette is not wide enough to display it all.

You can add another optional feature, a node icon or a property editor or compiler, or you can test your node, see "Creating node icons," "Adding a property editor or compiler" on page 83, and "Testing a user-defined node" on page 84.

*Creating node icons:*

Create the icons that are displayed in the workbench when your user-defined node is present.

**Before you start:**

You must complete the following tasks:
1. "Creating a user-defined node project" on page 76
2. "Creating the user-defined node plug-in files" on page 77
3. "Defining the node properties" on page 78

When you create a node, a set of default icons is created for you; files `clc16.gif` and `obj16.gif` are used for the node in the palette on the Broker Application Development perspective, and `obj30.gif` is used for the node in the Message Flow editor (that is, when it is dragged into a message flow). To change the default icons to your own icons, replace the supplied `.gif` files in the `icons` subdirectory of the plug-in project by your files.

You can add help, a property editor or compiler, or you can test your node, see "Adding help to the node" on page 81 or "Adding a property editor or compiler," and "Testing a user-defined node" on page 84.

*Adding a property editor or compiler:*

Create a property editor by using the IPropertyEditor interface to control how the properties of your user-defined node are displayed in the workbench. Create a custom compiler by using the IPropertyCompiler interface; for example, to encrypt a value before sending it to the server.

**Before you start:**

You must complete the following tasks:
1. "Creating a user-defined node project" on page 76
2. "Creating the user-defined node plug-in files" on page 77
3. "Defining the node properties" on page 78

The IPropertyEditor interface is used as the basis for all the node property editors in the workbench. You can customize the property editor to contain different kinds of controls, such as text fields and lists. See the IPropertyEditor and IPropertyCompiler interfaces in the Property editor API.

*Importing the plug-in API into the workbench:*

To create a property editor or compiler, you must first import the plug-in API into the workbench:
1. Click **File** → **Import** → **External Plug-ins and Fragments**.
2. Click **Next**.
3. Select the `com.ibm.etools.mft.api` plug-in.
4. When the plug-in is imported in the workspace, right-click the plug-in, and click **Update Classpath**.
5. Click **Finish**.
6. From the Window menu, click **Preferences**.
7. Expand **Plug-in development** and select **Target Platform**.
8. Click **Not in Workspace** to select all plug-ins except the `com.ibm.etools.mft.api` plug-in that you have imported into the workbench.
9. Click **OK**.

10. Switch to the Java perspective.
11. Select your user-defined node project in the Package Explorer, and click **Project** → **Clean Project**.
12. Right-click your user-defined node project, and click **Update Classpath**.

*Creating a Java class:*

To create a new Java class for your property editor or compiler, complete the following steps.

1. Switch to the Java perspective.
2. Select your user-defined node project in the Package Explorer, and click **Project** → **Clean Project**
3. Right-click your user-defined node project, and click **Update Classpath**
4. In the user-defined node project, select the /src directory, and click **File** → **New** → **Class**.
5. Type a name for your class in the **Name** text field.
6. Perform the following steps, according to whether you are creating a property editor or a property compiler.
   - If you are creating a property editor:
   a. Delete any text in the **Superclass** text field, and click **Browse**.
   b. Select the AbstractPropertyEditor class and click **OK**.
   - If you are creating a property compiler:
   a. Click **Add** next to the **Interfaces** text field.
   b. Select the IPropertyCompiler interface and click **OK**.
7. Click **Finish**.

*Testing your property editor or compiler:*

To test your property editor, see "Testing a user-defined node."

To test your property compiler, deploy to a broker the flow that contains your user-defined node.

A custom property editor can use Rational Application Developer or Eclipse APIs. When you migrate to a new version of WebSphere Message Broker, your custom property editor might not work if the Rational Application Developer or Eclipse APIs change. Update your property editor code to comply with the changed API.

## Testing a user-defined node
When you have created and installed the required resources, you can test your user-defined node.

**Before you start**

Complete the following tasks:
- "Creating a user-defined extension in C" on page 36 or "Creating a user-defined extension in Java" on page 59
- "Creating the user interface representation of a user-defined node in the workbench" on page 76
- "Installing user-defined extension runtime files on a broker" on page 106

Use the following steps to test your user-defined node:

1. Click **Run** → **Run**, or click the arrow to the right of the **Run** icon  in the toolbar and click **Run**. The Run dialog opens.

2. Select **Eclipse Application** in the left pane, and click **New** the left pane (the icon at the left end). The configuration page is displayed for a new configuration; the default name New_configuration is shown. To start a second workbench instance, click **Run**.

3. Open the Message Flow editor. Your new nodes are displayed in the node palette.

4. Create a message flow that includes your node. See Adding a message flow node.

5. Deploy the message flow to a broker. See Deploying a message flow application.

6. Send a test message through the flow and look for the results that you expect (for example, a message put to a target queue). You might have to write an application to send the test message to the message flow.

7. Use the diagnostic tools that are provided to determine whether your node works, or if not, what went wrong:

   a. For a description of some common problems and their solutions. See Resolving problems with user-defined extensions.

   b. Check the Event log. See Event log editor.

   c. Write entries to the Event log from your node. See "Using error logging from a user-defined extension" on page 115.

   d. Switch on user trace at debug level. See Using trace.

      The following debug messages are generated by a user trace to help you to understand the execution of your user-defined nodes and parsers:

      - BIP2233 and BIP2234: A pair of messages traced before and after a user-defined extension implementation function is started. These messages report the input parameters and the returned value.

        In these messages, an "implementation function" can be interpreted as either a C implementation function or a Java implementation method.

      - BIP3904: A message traced before starting the Java evaluate() method of a user-defined node.

      - BIP3905: A message traced before starting the C cniEvaluate() implementation function (iFpEvaluate member of CNI_VFT) of a user-defined node.

      - BIP4142: A debug message that is traced when starting a user-defined node utility function, where the utility function alters the state of a syntax element. All utility functions that start with cniSetElement*, where * represents all nodes with that stem, are included.

      - BIP4144 and BIP4145: A pair of messages traced by certain implementation functions that, when started by a user-defined extension, can modify the internal state of an object in the broker. Possible broker objects include syntax element, node, and parser.

        In these messages, an "implementation function" can be interpreted as either a C implementation function or a Java implementation method.

      - BIP4146: A debug message that is traced when starting a user-defined parser utility function, where the utility function alters the state of a syntax element. All utility functions that start with `cpiSetElement*`, where * represents all nodes with that stem, are included.

- BIP4147: An error message that is traced when a user-defined extension passes an invalid input object to a user-defined extension utility API function.
- BIP4148: An error message that is traced when a user-defined extension damages an object in a broker.
- BIP4149: An error message that is traced when a user-defined extension passes an invalid input data pointer to a user-defined extension utility API function.
- BIP4150: An error message that is traced when a user-defined extension passes invalid input data to a user-defined extension utility API function.
- BIP4151: A debug message that is traced when cniGetAttribute2 or cniGetAttributeName2 sets the return code to an unexpected value. Expected values are CCI_SUCCESS, CCI_ATTRIBUTE_UNKNOWN, and CCI_BUFFER_TOO_SMALL. Any other value is an unexpected value.
- BIP4152: A debug message that is traced in the following situations:
    1) cniGetAttribute2 or cniGetAttributeName2 sets the return code to CCI_BUFFER_TOO_SMALL.
    2) cniGetAttribute2 or cniGetAttributeName2 is called again with the correct size buffer, however the return code is set to CCI_BUFFER_TOO_SMALL.
  e. Add a Trace node to your message flow, and check the output that is generated.
  f. Use the flow debugger to debug the flow that contains your node. See Testing and debugging message flow applications.

When your node behavior is complete and correct, add the new node into your normal palette of nodes in the Message Flow editor, see "Packaging a user-defined node workbench project" on page 107. Until you complete this task, the new nodes are available only in your test workbench session on your local system.

## Implementing a user-defined parser

Create a user-defined parser to interpret messages with a different format and structure.

**Before you start**

Read the following topics:
- "Why use a user-defined extension?" on page 4
- "User-defined parsers" on page 26

A loadable implementation library, or a *LIL*, is the implementation module for a C parser (or node). A LIL is a Linux or UNIX shared object or Windows dynamic link library (DLL), that does not have the file extension .dll but .lil.

The implementation functions that you must write are listed in "C parser implementation functions" on page 197. The utility functions that are provided by WebSphere Message Broker to help you are listed in "C parser utility functions" on page 198.

WebSphere Message Broker provides the source for a sample user-defined parser called BipSampPluginParser.c. This example is a simple pseudo-XML parser that you can use in its current state, or you can modify.

The task of writing a parser varies considerably according to the complexity of the bit stream to be parsed. Only the basic steps are described here:

1. "Declaring and defining the parser"
2. "Creating an instance of the parser" on page 88
3. "Deleting an instance of the parser" on page 89

## Declaring and defining the parser

To declare and define a user-defined parser to the broker, you must include an initialization function, bipGetParserFactory, in your LIL. The following steps outline how the broker calls your initialization function and how your initialization function declares and defines the user-defined parser:

The following procedure shows you how to declare and define your parser to the broker:

1. The initialization function, bipGetParserFactory, is called by the broker after the LIL has been loaded and initialized by the operating system. The broker calls this function to understand what your LIL is able to do, and how it must be called. For example:

   ```
   CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix
   bipGetParserFactory()
   ```

2. The bipGetParserFactory function calls the utility function cpiCreateParserFactory. This function passes back a unique factory name (or group name) for all the parsers that your LIL supports. Every factory name (or group name) passed back must be unique throughout all the LILs in the broker.

3. The LIL calls the utility function cpiDefineParserClass to pass the unique name of each parser, and a virtual function table of the addresses of the implementation functions.

   For example, the following code declares and defines a single parser called InputxParser:

   ```
   {
    CciFactory* factoryObject;
    int rc = 0;
    CciChar factoryName[] = L"MyParserFactory";
    CCI_EXCEPTION_ST exception_st;

    /* Create the Parser Factory for this parser */
    factoryObject = cpiCreateParserFactory(0, factoryName);
    if (factoryObject == CCI_NULL_ADDR) {

     /* Any local error handling can go here */
    }
    else {
     /* Define the parsers supported by this factory */
     static CNI_VFT vftable = {CNI_VFT_DEFAULT};

    /* Setup function table with pointers to parser implementation functions */
    vftable.iFpCreateContext          = cpiCreateContext;
    vftable.iFpParseBufferEncoded     = cpiParseBufferEncoded;
    vftable.iFpParseFirstChild        = cpiParseFirstChild;
    vftable.iFpParseLastChild         = cpiParseLastChild;
    vftable.iFpParsePreviousSibling   = cpiParsePreviousSibling;
    vftable.iFpParseNextSibling       = cpiParseNextSibling;
    vftable.iFpWriteBufferEncoded     = cpiWriteBufferEncoded;
    vftable.iFpDeleteContext          = cpiDeleteContext;
    vftable.iFpSetElementValue        = cpiSetElementValue;
    vftable.iFpElementValue           = cpiElementValue;
    vftable.iFpNextParserClassName    = cpiNextParserClassName;
    vftable.iFpSetNextParserClassName = cpiSetNextParserClassName;
   ```

```
vftable.iFpNextParserEncoding        = cpiNextParserEncoding;
vftable.iFpNextParserCodedCharSetId = cpiNextParserCodedCharSetId;

cpiDefineParserClass(0, factoryObject, L"InputxParser", &vftable);
}

/* Return address of this factory object to the broker */
return(factoryObject);
}
```

The initialization function must create a parser factory by starting
cpiCreateParserFactory. The parser classes supported by the factory are defined
by calling cpiDefineParserClass. The address of the factory object (returned by
cpiCreateParserFactory) must be returned to the broker as the return value
from the initialization function.

For example:

a. Create the parser factory using the cpiCreateParserFactory function:

```
factoryObject = cpiCreateParserFactory(&rc, constParserFactory);
```

b. Define the classes of message supported by the factory using the
cpiDefineParserClass function:

```
if (factoryObject) {
  cpiDefineParserClass(&rc, factoryObject, constPXML, &vftable);
}
else {
   /* Error: Unable to create parser factory */
}
```

c. Return the address of this factory object to the broker:

```
return(factoryObject);
}
```

## Creating an instance of the parser

When the broker has received the table of function pointers, it calls the function
cpiCreateContext for each instantiation of the user-defined parser. If you have three
message flows that use your user-defined parser, your cpiCreateContext function is
called for each of them. This function allocates memory for that instantiation of the
user-defined parser to hold the values for the configured attributes. For example:

1. Call the cpiCreateContext function:

```
CciContext* _createContext(
  CciFactory* factoryObject,
  CciChar*    parserName,
  CciNode*    parserObject
){
  static char* functionName = (char *)"_createContext()";
  PARSER_CONTEXT_ST* p;
  CciChar          buffer[256];
```

2. Allocate a pointer to the local context and clear the context area:

```
p = (PARSER_CONTEXT_ST *)malloc(sizeof(PARSER_CONTEXT_ST));

if (p) {
   memset(p, 0, sizeof(PARSER_CONTEXT_ST));
```

3. Save the parser object pointer in the context:

```
p->parserObject = parserObject;
```

4. Save the parser name:

```
CciCharNCpy((CciChar*)&p->parserName, parserName, MAX_NODE_NAME_LEN);
```

5. Return the parser context:

```
return (CciContext*) p;
```

## Deleting an instance of the parser

Parsers are destroyed when a message flow is deleted or redeployed, or when the execution group process is stopped (using the mqsistop command). When a parser is destroyed, it must free any used memory and release any held resources using the cpiDeleteContext function. For example:

```
void cpiDeleteContext(
  CciParser*  parser,
  CciContext* context
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc = 0;

  return;
}
```

## Extending the capability of a C user-defined parser
When you have created a C parser, you can extend its capability.

**Before you start**

Ensure that you have read and understood the following topic:
* "Implementing a user-defined parser" on page 86

You can extend the capability of a C parser in the following ways:
* "Implementing the parser functionality"
* "Implementing input functions"
* "Implementing parse functions" on page 90
* "Implementing output functions" on page 91
* "Implementing a message header parser" on page 91

**Implementing the parser functionality:**

A parser needs to implement the following types of implementation function:
1. Input functions
2. Parse functions
3. Output functions

**Implementing input functions:**

Your parser must implement one, and only one, of the following input functions:
* "cpiParseBuffer" on page 225
* "cpiParseBufferEncoded" on page 227
* "cpiParseBufferFormatted" on page 228

The broker invokes the *input* function when your user-defined parser is required to parse an input message. The parser must tell the broker how much of the input bitstream buffer that it claims to own. In the case of a fixed-size header, the parser claims the size of the header. If the parser is intended to handle the whole message, it claims the remainder of the buffer.

For example:
1. The broker invokes the cpiParseBufferEncoded input function:

```
int cpiParseBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc;
```

2. Get a pointer to the message buffer and set the offset using the cpiBufferPointer utility function:

```
pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
pc->iIndex = 0;
```

3. Save the format of the buffer:

```
pc->iEncoding = encoding;
pc->iCcsid = ccsid;
```

4. Save the size of the buffer using the cpiBufferSize utility function:

```
pc->iSize = cpiBufferSize(&rc, parser);
```

5. Prime the first byte in the stream using the cpiBufferByte utility function:

```
pc->iCurrentCharacter = cpiBufferByte(&rc, parser, pc->iIndex);
```

6. Set the current element to the root element using the cpiRootElement utility function:

```
pc->iCurrentElement = cpiRootElement(&rc, parser);
```

7. Reset the flag to ensure parsing is reset correctly:

```
pc->iInTag = 0;
```

8. Claim ownership of the remainder of the buffer:

```
  return(pc->iSize);
}
```

**Implementing parse functions:**

General parse functions (for example, cpiParseFirstChild) are those invoked by the broker when the syntax element tree needs to be created in order to evaluate an ESQL or Java expression. For example, a Filter node uses an ESQL field reference in an ESQL expression. This field reference must be resolved in order to evaluate the expression. Your parser's general parse function is called, perhaps repeatedly, until the requested element is either created, or is known by the parser not to exist.

For example:

```
void cpiParseFirstChild(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc;

  if ((!cpiElementCompleteNext(&rc, element)) &&
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

    while ((!cpiElementCompleteNext(&rc, element))    &&
           (!cpiFirstChild(&rc, element)) &&
           (pc->iCurrentElement))
    {
      pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
    }
  }
  return;
}
```

**Implementing output functions:**

Your parser must implement one, and only one, of the following output functions:
- "cpiWriteBuffer" on page 247
- "cpiWriteBufferEncoded" on page 248
- "cpiWriteBufferFormatted" on page 249

The broker invokes the *output* function when your user-defined parser is required to serialize a syntax element tree to an output bit stream. For example, a Compute node might have created a tree in the domain of your user-defined parser. When a node, such as an MQOutput node, needs to serialize this tree, the parser is responsible for appending the output bitstream buffer with data that represents the tree that has been built.

For example:

```
int cpiWriteBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                initialSize = 0;
  int                rc = 0;
  const void* a;
  CciByte b;

  initialSize = cpiBufferSize(&rc, parser);
  a = cpiBufferPointer(&rc, parser);
  b = cpiBufferByte(&rc, parser, 0);

  cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);

  return cpiBufferSize(0, parser) - initialSize;
}
```

**Implementing a message header parser:**

Typically, the incoming message data is of a single message format, therefore one parser is responsible for parsing the entire contents of the message. The class name of the parser that is needed is defined in the Format field in the MQMD or the MQRFH2 header of the input message.

However, the message might consist of multiple formats, for example where there is a header in one format followed by data in another format. In this case, the first parser has to identify the class name of the parser that is responsible for the next format in the chain, and so on. In a user-defined parser, the implementation function cpiNextParserClassName is invoked by the broker when it navigates down a chain of parser classes for a message that is composed of multiple message formats.

If your user-defined parser supports parsing a message format that is part of a multiple message format, the user-defined parser *must* implement the cpiNextParserClassName function.

For example:
1. Call the cpiNextParserClassName function:

```
        void cpiNextParserClassName(
          CciParser*  parser,
          CciContext* context,
          CciChar*    buffer,
          int         size
        ){
          PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
          int                rc = 0;
```
2. Copy the name of the next parser class to the broker:
```
          CciCharNCpy(buffer, pc->iNextParserClassName, size);

          return;
        }
```

## Compiling a C user-defined extension

Compile user-defined extensions in C for all supported operating systems.

**Before you start:**

If you create your own user-defined nodes, parsers, and user exits in C, compile
them on the operating system on which the target broker is running. Samples are
provided for both nodes and parsers, and are described in "Sample node files" on
page 121 and "Sample parser files" on page 123. Use the instructions here to
compile the samples. If you want to create your own extensions, see the following
topics:

- "Creating a user-defined extension in C" on page 36
- "Implementing a user-defined parser" on page 86
- "Implementing a user-defined exit" on page 99

These instructions use the file names of the supplied samples. If you are compiling
your own user-defined extensions, substitute your own file names.

When you compile a user-defined extension that is written in C, you need a
compatible compiler. For details of supported compilers, see Optional software
support.

**Header files:**

The following header files define the C interfaces:

**BipCni.h**
> Message processing nodes

**BipCpi.h**
> Message parsers

**BipCci.h**
> Interfaces common to both nodes and parsers

**BipCos.h**
> Platform-specific definitions

**Compiling:**

Compile the source for your user-defined extension on each of the supported
operating systems to create the executable file that the broker calls to implement
your user-defined extension. On Linux, UNIX, and z/OS systems, this file is a
loadable implementation library (LIL) file; on Windows systems, it is a dynamic
load library (DLL) file.

The libraries that you build to contain user-defined nodes or parsers must have the extension `.lil` on all operating systems so that the broker can load them. Libraries that contain user exits must have the extension `.lel` on all operating systems. The examples in this topic show libraries with the extension `.lil`

Refer to the documentation for the compiler that you are using for full details of available compile and link options that might be required for your programs.

Navigate to the directory where your user-defined extension source code is located, and follow the instructions for your operating system:
- AIX
- HP-UX on PA-RISC
- HP-Itanium
- Linux
- Solaris
- Windows
- z/OS

**Compiling on AIX:**

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a default 64-bit execution group. If the extension is called from a 32-bit execution group, make the following substitutions in the compile and link examples:
- Replace `xlc_r -q64 -qwarn64` with `xlc_r -q32`
- Replace `-l imbdfplg64` with `-l imbdfplg`

```
xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c SwitchNode.c \
   -o SwitchNode.o

xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c BipSampPluginUtil.c \
   -o BipSampPluginUtil.o

xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c Common.c \
   -o Common.o

xlc_r -q64 -qwarn64 \
   -I. \
   -I/install_dir/include/plugin \
   -c NodeFactory.c \
   -o NodeFactory.o

xlc_r -q64 -qwarn64 \
     -qmkshrobj \
     -bM:SRE \
     -bexpall \
     -bnoentry \
     -o SwitchNode.lil SwitchNode.o \
        BipSampPluginUtil.o Common.o NodeFactory.o \
```

```
   -L /install_dir/lib \
   -l imbdfplg64
```

```
chmod a+r SwitchNode.lil
```

**Compiling on HP-UX on PA-RISC:**

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a default 64-bit execution group. If the extension is called from a 32-bit execution group, make the following substitutions in the compile and link examples:

- Replace +DD64 with +DD32
- Replace -l imbdfplg64 with -l imbdfplg

```
cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o \
   -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c Common.c \
   -o output_dir/Common.o \
   -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o \
   -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o \
   -Ae

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT\
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   -c TransformNode.c \
   -o output_dir/TransformNode.o \
   -Ae

ld -b \
   -o output_dir/SwitchNode.lil \
   output_dir/BipSampPluginUtil.o \
   output_dir/Common.o \
   output_dir/NodeFactory.o \
   output_dir/SwitchNode.o \
   output_dir/TransformNode.o \
   -L install_dir/lib \
   -L install_dir/xml4c/lib \
```

```
    -L install_dir/merant/lib \
    -L install_dir/jre/lib/PA_RISC2.0 \
    -L install_dir/jre/lib/PA_RISC2.0/server \
    -l imbdfplg64

chmod a+r output_dir/SwitchNode.lil
```

**Compiling on HP-Itanium:**

When you compile a user-defined extension that is written in C, use a supported compiler.

The following instructions are for compiling an extension for a 64-bit execution group; 32-bit execution groups are not supported.

```
cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c BipSampPluginUtil.c \
    -o output_dir/BipSampPluginUtil.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c Common.c \
    -o output_dir/Common.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c NodeFactory.c \
    -o output_dir/NodeFactory.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c SwitchNode.c \
    -o output_dir/SwitchNode.o

cc +z +e +DD64 -D_HPUX_SOURCE -DTHREADS -D_REENTRANT -Ae \
    -I. \
    -I/install_dir/include \
    -I/install_dir/include/plugin \
    -c TransformNode.c \
    -o output_dir/TransformNode.o


ld -b \
    -o output_dir/SwitchNode.lil \
    output_dir/BipSampPluginUtil.o \
    output_dir/Common.o \
    output_dir/NodeFactory.o \
    output_dir/SwitchNode.o \
    output_dir/TransformNode.o \
    -L install_dir/lib \
    -L install_dir/xml4c/lib \
    -L install_dir/merant/lib \
    -L install_dir/jre/lib/IA64N\
    -L install_dir/jre/lib/IA64N/server \
    -l imbdfplg

chmod a+r output_dir/SwitchNode.lil
```

**Compiling on Linux:**

When you compile a user-defined extension that is written in C, use a supported compiler.

To determine the 32-bit or 64-bit linkage that is required for an execution group created in a broker on your specific operating system, see Support for 32-bit and 64-bit platforms.

When you compile programs on Linux on POWER, replace the option -fpic with -fPIC if you want to use dynamic linking and avoid any limit on the size of the global offset table.

The following instructions are for compiling an extension for a 64-bit execution group on Linux on POWER and Linux on System z.

- To compile the extension for a 64-bit execution group on Linux on x86-64, replace -limbdfplg with -limbdfplg64 in the link example.
- To compile the extension for a 32-bit execution group on Linux on x86, replace -m64 with -m32 in the compile and link examples.

```
g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   TransformNode.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   SwitchNode.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   BipSampPluginUtil.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   Common.c

g++ -c -m64 -ansi -Wall -Wno-format-y2k -fpic \
   -I. \
   -I/install_dir/include \
   -I/install_dir/include/plugin \
   - DLINUX -D_THREADS -D_POSIX_PTHREAD_SEMANTICS -D_REENTRANT \
   NodeFactory.c

g++ -m64 -o samples.lil \
   TransformNode.o \
   SwitchNode.o \
   BipSampPluginUtil.o \
   Common.o NodeFactory.o \
   -shared -lc -lnsl -ldl \
   -L/install_dir/lib -limbdfplg
```

These commands create the file `samples.lil` that provides TransformNode and SwitchNode objects.

**Compiling on Solaris:**

When you compile a user-defined extension that is written in C, use a supported compiler.

To determine the 32-bit or 64-bit linkage that is required for an execution group created in a broker on your Solaris operating system, see Support for 32-bit and 64-bit platforms.

The following instructions are for compiling an extension for a 64-bit execution group on Solaris on SPARC.
- To compile the extension for a default 64-bit execution group on Solaris on x86-64, replace `-xarch=v9` with `-xarch=amd64` in the compile examples.
- To compile the extension for a 32-bit execution group on Solaris on SPARC:
  - Replace `-xarch=v9` with `-xarch=v8plus` in the compile examples.
  - Replace `-l imddfplg64` with `-l imddfplg` in the link example.

```
cc -xarch=v9 -mt \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  -c SwitchNode.c \
  -o output_dir/SwitchNode.o

cc -xarch=v9 -mt \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  -c BipSampPluginUtil.c \
  -o output_dir/BipSampPluginUtil.o

cc -xarch=v9 -mt \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  -c NodeFactory.c \
  -o output_dir/NodeFactory.o

cc -xarch=v9 -mt \
  -I. \
  -I/install_dir/include \
  -I/install_dir/include/plugin \
  -c Common.c \
  -o output_dir/Common.o

cc -xarch=v9  -xcode=pic32 -mt -G \
  -o output_dir/SwitchNode.lil \
     output_dir/SwitchNode.o \
     output_dir/BipSampPluginUtil.o \
     output_dir/NodeFactory.o \
     output_dir/Common.o \
  -L /install_dir/lib \
  -l imbdfplg64

chmod a+r output_dir/SwitchNode.lil
```

**Compiling on Windows:**

When you compile a user-defined extension that is written in C, use a supported compiler.

Ensure that you include a space between `SwitchNode.c` and `BipSampPluginUtil.c`, and also between `-link` and `/DLL`.

Enter the command as a single line of input; in the following example the lines have been split to improve readability.

```
cl /VERBOSE /LD /MD /Zi /GX /I.
    /Iinstall_dir\include\plugin
    SwitchNode.c BipSampPluginUtil.c Common.c
    NodeFactory.c TransformNode.c
    -link /DLL install_dir\lib\imbdfplg.lib
    /OUT:SwitchNode.lil
```

If you have correctly set the *LIB* environment variable, you do not have to specify the full paths to the LIB files.

**Compiling on z/OS:**

When you compile a user-defined extension that is written in C, use a supported compiler.

Force your link to use prelinker or linker by setting the _CC_STEPS variable to -1:

```
export _CC_STEPS=-1
```

Alternatively, add these two lines to your makefile to export it:

```
_CC_STEPS=-1
.EXPORT : _CC_STEPS
```

To create optimized builds, use **-2** in place of **-g** in the following commands:

```
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./SwitchNode.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./SwitchNode.o ./SwitchNode.c

cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./BipSampPluginUtil.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./BipSampPluginUtil.o ./BipSampPluginUtil.c

cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./Common.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./Common.o ./Common.c
```

```
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,float\ieee\) \
 -Wc,xplink \
 -W0,LIST\(./NodeFactory.lst\) \
 -I. -I${install_dir}/include \
 -I${install_dir}/include/plugin \
 -I${install_dir}/sample/include \
 -I${install_dir}/sample/plugin \
 -o ./NodeFactory.o ./NodeFactory.c
cc \
 -Wl,DLL -g -Wl,p,map -Wl,LIST=ALL,MAP,XREF,REUS=RENT \
 -Wl,xplink \
 -o ./SwitchNode.lil ./SwitchNode.o ./BipSampPluginUtil.o \
 ./Common.o ./NodeFactory.o \
 ${install_dir}/lib/libimbdfplg.x
```

Run the following command to set the file permissions of the user-defined extension to group read and to be executable:

```
chmod a+rx {output_dir}/SwitchNode.lil
```

# Implementing a user-defined exit

You can develop and deploy a user-defined exit.

**Before you start:**
• Read "User exits" on page 30.

The following topics describe the steps required to develop and deploy a user-defined exit.

1. "Developing a user exit"
2. "Deploying a user exit" on page 100

## Developing a user exit

Develop a user exit by declaring it, implementing its behavior, then compiling it.

To develop a user exit, follow these steps.

1. Declare the user exit.

   Declare a user exit by using the bipInitializeUserExits function to specify the following properties:

   a. Name (used to register and control the active state of the exit)

   b. User context storage

   c. A function to be invoked (for one or more Event Types)

2. Implement the user exit behavior.

   When the user exit is declared, a set of functions is registered, and these functions are invoked when specific events occur. The behavior of the user exit is provided by implementing these functions. The following table lists the events and their associated functions:

| Event | Function |
|---|---|
| A message is dequeued from the input source | cciInputMessageCallback |
| A message is propagated to the node for processing | cciPropagatedMessageCallback |
| A request message is sent to the output node's transport, and transport-specific destination information is written to "WrittenDestination" in the LocalEnvironment | cciOutputMessageCallback |
| The node completes processing | cciNodeCompletionCallback |

| Event | Function |
|---|---|
| The transaction ends | cciTransactionEventCallback |

3. Your user exit code must implement the cleanup function.

   The user exit library must implement the bipTerminateUserExits function. This function is invoked as the ExecutionGroup's process is ending, and your user exit must clear up any resources allocated during the bipInitializeUserExits function.

4. Compile.

   Use your existing process for your environment to compile your user exit. The supported C compilers are shown in Optional software support. See "Compiling a C user-defined extension" on page 52 for more details.

5. Link the compiled code to a library with the extension .lel that exports the bipInitializeUserExits and bipTerminateUserExits functions.

## Deploying a user exit
Deploy your user exit to the broker.

**Before you start:**
- Write and compile the user exit code. See "Developing a user exit" on page 99.
- Ensure that the exit:
  1. Is in a library that has the extension .lel
  2. Exports the functions bipInitializeUserExits and bipTerminateUserExits

You can set the state of the user exit dynamically to active, or inactive, on a per-message flow basis without restarting the broker.

To deploy the user exit:

1. Install the user exit code on a broker.

   The library containing the user exit code must be installed on a file system that can be accessed by the broker. For example, the file must have read and execute authority for the user ID under which the broker runs. The broker looks in the following places for libraries that contain user exits:
   - The broker property UserExitPath defines a list of directories separated by colons (semicolons on Windows). Use the –x flag on the mqsicreatebroker or mqsichangebroker command to set this property for 32-bit execution groups for each broker.

     Alternatively, you can append the directory containing the directory that holds the extension files to the environment variable MQSI_USER_EXIT_PATH associated with the environment in which the broker is running.

     If both are set, the environment variable takes precedence. All the directories in the environment variable are searched in the order in which they appear in the variable, then all the directories in the broker property are searched in the order in which they appear in the property.
   - For 64-bit extensions, you cannot use the –x parameter to modify the exit path. Append the directory containing the directory that holds the extension files to the environment variable MQSI_USER_EXIT_PATH64.

2. Load the user exit library into the broker's processes.

   When the user exit library has been installed on the broker, you must load it in one of the following ways:

- Stop and restart the broker.
- Run the mqsireload command to restart the execution group processes.

3. Activate the user exit.

   User exits can be active or inactive, and are inactive by default. You can change the state of a user exit dynamically by using the mqsichangeflowuserexits command on a per-flow basis, without having to restart the broker. You can also change the default state for a set of user exits to active on a per-broker basis by using the mqsichangebroker command; in this case, you do have to restart the broker.

   To set the default user exit state for a broker:

   a. Stop the broker.

   b. Set the activeUserExits property of the broker by using the mqsichangebroker command.

   c. Start the broker and check the system log to ensure that all execution groups start without error. If any invalid user exit names are specified (that is, the user exit is not provided by any library loaded by the execution group), a BIP2314 message is written to the system log and all flows in the execution groups fail to start unless you take one of the following actions:

      - Provide a library in the user exit path that implements the exit; then run the mqsireload command, or restart the broker, to load an exit from the library.
      - Run the mqsichangeflowuserexits command to remove the exit from both the active and inactive lists.

   You can also override the default user exit state for a broker. You can use the mqsichangeflowuserexits command to activate, or deactivate, user exits on a per-execution group or per-message flow basis, with the order of precedence being message flow then execution group. When multiple exits are active for a flow, the broker starts them in the order that is defined by the mqsichangeflowuserexits command.

## Packaging and distributing user-defined extensions

When you have created and tested a user-defined extension, you can package and distribute it.

**Before you start:**

Complete the following tasks:
- "Implementing user-defined extensions" on page 32
- "Testing a user-defined node" on page 84

When you have created and tested your user-defined extension, you can distribute these resources to other computers in your broker domain by using one of the following options:

- Copy the files generated by the compilation step to all the computers on which you have created brokers that might need these resources, see "Installing user-defined extension runtime files on a broker" on page 106. For a more automated approach, see "Installing a user-defined extension to current and past versions of the broker" on page 108.
- Package and install the user-defined extensions:

1. Package the resources that make up the workbench representation of your user-defined node to create an Eclipse plug-in, see "Creating the user interface representation of a user-defined node in the workbench" on page 76.

2. Install the plug-in on all the computers on which your workbench users might want to use them, following the instructions in "Installing a user-defined extension to current and past versions of the broker" on page 108. This step is not required for user-defined parsers.

## Packaging a Java user-defined node

How to package a Java user-defined node.

**Before you start**

You must have a user-defined node written in Java. This node can be one of the provided sample nodes that are described in "Sample node files" on page 121, or a node that you have created yourself using the instructions in either "Creating a message processing or output node in Java" on page 65 or "Creating an input node in Java" on page 60.

You can package a user-defined node in two ways:

- **PAR**

  A Plug-in Archive (PAR) is the deployment unit for Java user-defined nodes. The PAR contains the user-defined node classes and, if required as dependencies, can contain JAR files. A PAR file is a compressed file with a `.par` file extension. The directory structure in the `.par` file has the following format:

  – `/classes`

    The user-defined node classes are stored in this location.

  – `/lib`

    JAR files that are required by the user-defined node are stored in this location. This directory is optional because it might not be necessary to include JAR files.

  The following procedure describes how to package an example user-defined node, *parexamplenode*. In this example, the PAR is to be contained in *par.example.parexamplenode.class* with a JAR file dependency *dependency.jar*.

  1. Create the directory structure; for example:
     – `/classes/par/example/parexamplenode.class`
     – `/lib/dep.jar`

  2. Issue a file compression command to create the PAR; for example:
     `jar cvf parexample.par classes lib`

  The PAR should be placed in the LIL path that is specified in "Installing user-defined extension runtime files on a broker" on page 106.

- **JAR**

  User-defined nodes can be packaged using a simple JAR. For example, if your node is defined in `example/jarexamplenode.class`, create the JAR by using the `jar cvf jarexample.jar example` command.

  The preferred way to package a Java user-defined node is to use a PAR file, because all dependencies can be packaged with the node, and each node is loaded in a separate classloader. Refer to "User-defined node classloading" on page 103 for information on classloading.

  The JAR should be placed in the LIL path that is specified in "Installing user-defined extension runtime files on a broker" on page 106.

**Deployment dependencies:**

If a user-defined node requires an external package, the package can be deployed in one of following ways:

- The external packages can be added to the /lib directory in the deployed PAR.
- For external packages that are shared between several node types, the packages can be added to one of the following locations:
  - The<workpath>/shared-classes/ directory
  - The CLASSPATH environment variable, where all user-defined nodes that are in the broker installation can access the packages

**User-defined node classloading:**

When a Java user-defined node is packaged as a PAR file, the Java user-defined node is loaded in a separate classloader.

The classloader loads any class that is packaged within the deployed PAR file. The classes that are placed in the JAR file override any classes that are in the shared classes directory or the CLASSPATH environment variable. If the deployed PAR file contains more than one node type, the nodes share the same classloader. Therefore, a set of user-defined nodes that share static data must be packaged in a single PAR file.

Java user-defined nodes that are packaged as simple JAR files are loaded in the same classloader. The classes and the location from which they are loaded are written to user trace, therefore you can use this information to check that the correct classes are being loaded.

The broker uses the following classloader tree:

```
                    Bootstrap  ⎫
                        |      ⎬  JVM classloaders
                     System    ⎭
                        |
                     Common
                    /        \
              Broker          Shared
                          /     |      \
                   Grouped      |       EGShared
                            /       \
                    NodeType1  ...  NodeTypeN
```

**Java user-defined nodes**
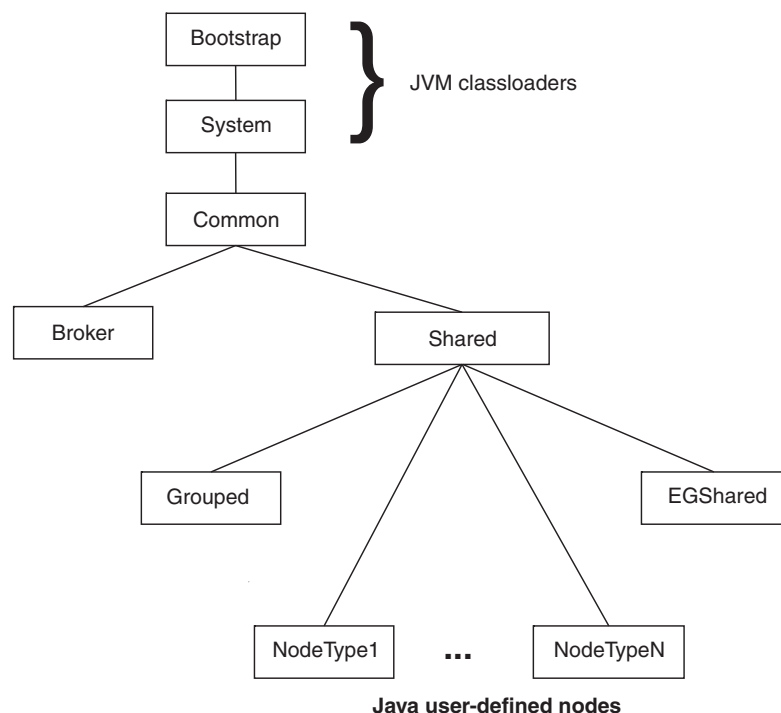
- **Common classloader**: This component loads the classes that are shared between the broker and user code. For example, the classes that are contained in `jplugin2.jar` are common to the broker and the user code.
- **Broker classloader**: This component loads the broker internal classes. These classes cannot be accessed by user classes.
- **Shared classloader** : This component loads classes from JAR files that have been placed in the `WorkPath/shared-classes/` directory and the CLASSPATH environment variable. These classes are available to all Java user-defined nodes and JavaCompute nodes within the broker.

  The shared classloader also loads context classes. It uses the CLASSPATH and the `WorkPath/shared-classes/` directory to search for classes.

  The CLASSPATH environment variable can contain the wildcard character (*) at the end of a directory path specifier. The wildcard is expanded to include all files in that directory with the extension `.jar` or `.JAR`.

  The broker classloader and the shared classloader are children of the common classloader. Therefore, the contents of the shared classloader are not visible to the broker classloader. Do not store the following resources in this directory:
  - User-defined nodes
  - Classes that have a dependency on other classes that have been deployed with a user-defined node.
- **Grouped classloader**: This component loads all user-defined nodes that are packaged as JAR files. If you have packaged user-defined nodes in an earlier version, they are loaded using this loader. User-defined nodes that are packaged in JAR files are loaded into one loader, and can therefore share static data.
- **EGShared classloader**: This component loads classes from JAR files that have been deployed to the broker run time in a BAR file. This method supports the deployment mechanism for the JavaCompute node. Each time a BAR file is deployed, a new instance of the EGShared classloader is created and the old instance is discarded. Therefore, the JavaCompute node can reload modified versions of the same class without the need to restart the broker.

*User-defined nodes classloading search paths:*

**User-defined nodes package in a PAR file**

The broker uses the following search path to find user-defined node classes:
1. `/classes` to locate classes in the deployed PAR file.
2. `/lib` to locate any JAR files in the deployed PAR file.
3. `WorkPath/shared-classes/` to locate any JAR files.
4. `CLASSPATH` environment variable.

**User-defined nodes package in a JAR file**

The broker uses the following search path to find user-defined node classes:
1. The deployed JAR file.
2. `WorkPath/shared-classes/` to locate any JAR files
3. `CLASSPATH` environment variable.

**Endorsed standards for overriding classes**

The endorsed standards overriding mechanism allows the following standard packages to be overridden in the JRE:

- javax.rmi.CORBA
- org.omg.CORBA
- org.omg.CORBA.DynAnyPackage
- org.omg.CORBA.ORBPackage
- org.omg.CORBA.portable
- org.omg.CORBA.TypeCodePackage
- org.omg.CORBA_2_3
- org.omg.CORBA_2_3.portable
- org.omg.CosNaming
- org.omg.CosNaming.NamingContextExtPackage
- org.omg.CosNaming.NamingContextPackage
- org.omg.Dynamic
- org.omg.DynamicAny
- org.omg.DynamicAny.DynAnyFactoryPackage
- org.omg.DynamicAny.DynAnyPackage
- org.omg.IOP
- org.omg.IOP.CodecFactoryPackage
- org.omg.IOP.CodecPackage
- org.omg.Messaging
- org.omg.PortableInterceptor
- org.omg.PortableInterceptor.ORBInitInfoPackage
- org.omg.PortableServer
- org.omg.PortableServer.CurrentPackage
- org.omg.PortableServer.POAManagerPackage
- org.omg.PortableServer.POAPackage
- org.omg.PortableServer.portable
- org.omg.PortableServer.ServantLocatorPackage
- org.omg.SendingContext
- org.omg.stub.java.rmi
- org.w3c.dom
- org.xml.sax
- org.xml.sax.ext
- org.xml.sax.helpers

Refer to the Endorsed Standards Override Mechanism for more information.

To override these packages in the broker, place the JAR files for the API standards in the /lib directory of the PAR file.

*JNDI context:* When looking up a JNDI context, the context classloader is used. If the lookup uses classes that are packaged with the user-defined node, the context classloader must be the same as the classloader that is being used to load the user-defined node. To ensure that each thread uses the same classloader, the following code can be included in the user-defined node class:

```
Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
```

## Installing user-defined extension runtime files on a broker

Install the compiled runtime files for your user-defined extension on the broker on which you want to test its function.

**Before you start**

- Create and compile your user-defined extension using the procedure described in "Compiling a Java user-defined node" on page 74 or "Compiling a C user-defined extension" on page 52.
    - The files that have been created for extension created in C depend on the underlying broker operating system:

        | Windows | A dynamic link library (DLL), named with a file type of .lil.

        | Linux | A shared object, again with a file type of .lil.

        | UNIX | A shared object, again with a file type of .lil.

        | z/OS | A shared object, with a file type of .lil.

    - For Java nodes, a Java Archive file (JAR), with a file type of .jar (on all operating systems).
- If you have created a user-defined node, you must also complete the task "Creating the user interface representation of a user-defined node in the workbench" on page 76.

This task instructs you to stop and restart brokers. This action is required in all but the two circumstances described in step 4 later in this section, although if you do stop and restart the broker, you can ensure that anyone with an interest in a particular execution group is made aware that recent changes have been made.

To install runtime files on the broker:

1. Stop the broker on which you want to install your compiled or packaged user-defined extension file (files with extension .lil, .jar, .par, .pdb, or .lel)
2. Create a directory if you do not already have one for this purpose. Add the directory to the LILPATH by using the mqsichangebroker command.
   **CAUTION:**
   **Do not put the .lil, .jar, .par, .pdb, or .lel files in the WebSphere Message Broker installation directory, because they might be overwritten by the broker.**
3. Put your user-defined file in the directory, and make sure that the broker has access to it. For example, on Linux or UNIX, use the chmod 755* command on the file.
4. Stop and restart the broker to implement the change and to ensure that the existence of the new file is detected. A broker restart is not necessary in the following circumstances:
   - If you have created an execution group in the workbench, and nothing is yet deployed to it, you can add the .lil, .pdb, .jar, .par, or .lel file to your selected directory.
   - If something has already been deployed to the execution group that you want to use, add the .lil, .pdb, .jar, .par, or .lel file to your selected directory, and issue the mqsireload command to restart the group. You cannot overwrite an existing file on the Windows system when the broker is running, because of the file lock that is put in place by the operating system.

Use these two approaches with care, because any execution group that is connected to the same broker also detects the new `.lil`, `.pdb`, `.jar`, `.par`, or `.lel` files when that execution group restarts, or when something is first deployed to that execution group.

5. Repeat the prevuious steps for every broker that needs the user-defined extension file. If all of your brokers are on the same operating system type, you can build the user-defined extension file once and distribute it to each of your systems.

   If you have a cluster, for example, that includes one AIX, one Solaris, and one Windows broker, you must build the files separately on each operating system type.

   Windows  On Windows, the `.pdb` file provides symbolic information that is used when stack diagnostic information is displayed in the event of access violations or other software malfunctions.

6. For C user-defined extensions, store the `.pdb` file in the same directory as the `.lil` file to which it corresponds.

7. Use either the mqsichangebroker command or the mqsicreatebroker command, as appropriate, to specify to the broker the directory that contains the user-defined extension file.

   When you have installed a user-defined extension, it is referred to by its schema and name, just like a message flow.

The broker loads the user-defined extension files during initialization. After loading the files, the broker calls the registration functions in the user-defined extension and records what nodes or parsers the user-defined extension supports.

A C user-defined extension implements a node or parser factory that can support multiple nodes or parser types. For more information, see "Node and parser factory behavior" on page 10. Java users are not required to write a node factory.

## Packaging a user-defined node workbench project

Package a user-defined node workbench project to distribute to other computers for use throughout your broker domain.

**Before you start**

1. Create and compile your user-defined node in Java or C, see "Compiling a Java user-defined node" on page 74 or "Compiling a C user-defined extension" on page 52.
2. Create the representation of your user-defined node in the workbench, see "Creating the user interface representation of a user-defined node in the workbench" on page 76.
3. Test your user-defined node, see "Testing a user-defined node" on page 84.

Use the following instructions to package user-defined node projects for user-defined nodes created in Java or C.

Although you have used and tested your user-defined node on your local computer, you must make its associated files available on other computers when your user-defined node is ready for use throughout your broker domain. A user-defined node consists of two sets of files:

- Files that support the node execution in the broker. You created these files in "Creating a user-defined extension in C" on page 36 or "Creating a user-defined extension in Java" on page 59.

- Files that represent the node in the workbench. You created these files in "Creating the user interface representation of a user-defined node in the workbench" on page 76.

The workbench representation consists of a set of resources that have been created as an Eclipse plug-in. To package the plug-in so that it can be distributed to other computers, use the following steps to generate a JAR file that contains the plug-in project for your node:

1. Switch to the Plug-in Development perspective.
2. Right-click the node project that you want to package for distribution and click **File** → **Export**.
3. From the list displayed, select **Plug-in Development** → **Deployable plug-ins and fragments**.
4. Click **Next**. Select the check box next to your node project and specify a location for your JAR file. The file will be placed under the *plugin* directory relative to your specified location.
5. Click **Finish**.

The JAR file that contains your node plug-in project is generated at the location you specified. To produce a self-contained package for the node distribution, create a `.zip` file and add the generated JAR file, and any C source code or compiled files, to the `.zip` file by using any file compression utility.

To distribute the workbench files, see "Installing a user-defined extension to current and past versions of the broker."

To distribute the runtime components, see "Installing user-defined extension runtime files on a broker" on page 106.

For installation on another system, see "Installing a user-defined extension to current and past versions of the broker."

To distribute your node commercially, for information about issues such as versioning and updating your user-defined node, see the PDE Guide.

## Installing a user-defined extension to current and past versions of the broker

Install user-defined extensions that you have developed yourself, or have acquired from an independent software vendor, with the minimum of user intervention.

**Before you start**

Complete the following tasks:

1. "Compiling a Java user-defined node" on page 74 or "Compiling a C user-defined extension" on page 52
2. "Creating the user interface representation of a user-defined node in the workbench" on page 76
3. "Testing a user-defined node" on page 84
4. "Packaging a user-defined node workbench project" on page 107

You must install user-defined extensions on all appropriate broker computers, and, if the extension is a user-defined node, on the workbench computers (user-defined parsers have no workbench component). Components can be installed separately,

or as part of one process. The components can be on different systems, therefore check that the installations are completed on all affected systems.

If an extension writes messages to user trace, you must update the environment variable MQSI_CONSOLE_NLSPATH (Windows systems), or NLSPATH (all other systems), so that the mqsiformatlog command can find the message catalog.

**The Message Broker Toolkit installation:**

Before installing a user-defined node, check the version of the Message Broker Toolkit to which you are installing, because a specific version of the toolkit might be a prerequisite of the user-defined extension, or it might require specific files to run.

To determine the version, see "Detecting installed versions of WebSphere Message Broker" on page 112.

1. Copy your files to a directory that the Message Broker Toolkit can access, so that you can view your user-defined node in the workbench session. Choose one of the following options:
   - Add a new Eclipse extension location. This option uses standard workbench tasks, and helps you to maintain your files safely and separately from those files provided with the default installation.
     a. Create a directory called `eclipse` in a suitable location in your file system; the directory structure that contains the `eclipse` directory is not significant.
     b. Within the `eclipse` directory, create an empty file named `.eclipseextension`, and directories named `features` and `plugins`.
     c. In the Message Broker Toolkit, click **Help** → **Software Updates** → **Manage Configuration**; the Product Configuration dialog opens.
     d. Click **Add an Extension Location**. Browse to your new folder and select it. You are prompted to restart the Message Broker Toolkit.
     e. Add your new plug-in JAR file to the `plugins` directory in the `eclipse` directory that you have created.
   - Add your new plug-in JAR file directly into the `plugins` directory where the Message Broker Toolkit is installed. For example, on Windows 32–bit editions, add the file to `C:\Program Files\IBM\WMBT610\plugins`.

     If you choose this option, you might find it difficult to manage your plug-ins files safely if you later remove or replace them. Also, your system administrator might want to control security and access on computers that are used by more than one user, and might not set permissions for users to write to the primary installation directories.
   - Create Eclipse link files to the directories in which you maintain your plug-in files.

     For details about how to create link files, see the developerWorks® article about using Eclipse features (the section entitled "Using link files to manage an Eclipse install"). The process is like the first option (using **Manage Configuration**), and creates similar resources, but you must write and maintain the link files in the correct format.
2. Restart your workbench session for the changes to take effect.

**Runtime installation:**

You might be required to detect the version of the runtime components that are installed, to ensure that the correct LIL file is loaded by the correct level of the broker. See "Detecting installed versions of WebSphere Message Broker" on page 112.

To add `.jar` or `.lil` files to runtime installations on WebSphere Business Integration Message Broker Version 5.0 or later, and WebSphere Message Broker Version 6.0 and later, see "Installing user-defined extension runtime files on a broker" on page 106.

**Installing a user-defined extension for single broker:**

**Version 6.1 and earlier**

- To make a 32-bit extension accessible from only one broker on the system, modify the *UserLilPath* setting for the broker by specifying the `-l` parameter on the mqsicreatebroker or mqsichangebroker command.
- To make a 64-bit extension accessible from only one broker on the system, modify the *UserLilPath64* setting for the broker by specifying the `-r` parameter on the mqsicreatebroker or mqsichangebroker command.

For more information, see mqsicreatebroker command and mqsichangebroker command.

**Installing a user-defined extension for multiple brokers:**

**Version 6.1 and earlier**

- To affect all brokers on a system, you modify the system LILPATH. Append the directory containing the directory that holds the extension files to the environment variable MQSI_LILPATH (for 32-bit extensions) or MQSI_LILPATH64 (for 64-bit extensions).

Make this change by creating a custom environment script in the working directory:

- <span style="border:1px solid;padding:1px">Linux</span> <span style="border:1px solid;padding:1px">UNIX</span> On Linux and UNIX systems: `/var/mqsi/common/profiles`

- <span style="border:1px solid;padding:1px">Windows</span> On Windows: `%ALLUSERSPROFILE%\Application Data\IBM\MQSI\common\profiles` where `%ALLUSERSPROFILE%` is the environment variable that defines the system working directory. The default directory depends on the operating system:
  - On Windows XP and Windows Server 2003: `C:\Documents and Settings\All Users\Application Data\IBM\MQSI\common\profiles`
  - On Windows Vista and Windows Server 2008: `C:\ProgramData\IBM\MQSI\common\profiles`

  The actual value might be different on your computer.

You can give the environment script any name, but the file extension must be `.cmd` on Windows and `.sh` on all other platforms. The script can perform all the operations of a shell script, but you must limit the scope to only appending the following variables:

**MQSI_LILPATH**
> Defines the directories to search for 32-bit plug-ins

**MQSI_LILPATH64**
> Defines the directories to search for 64-bit plug-ins

**CLASSPATH**
Defines the locations that Java searches for additional classes

**MQSI_CONSOLE_NLSPATH**
On Windows: Defines the location of message catalogs (DLL files)

**NLSPATH**
On Linux and UNIX: Defines the location of message catalogs (CAT files)

**PATH**
Defines the location of executable files. On Windows: This variable also defines the location of dependent libraries.

**LIBPATH / SHLIB_PATH / LD_LIBRARY_PATH**
On UNIX and Linux: Defines the location of dependent libraries.

**Example scripts:**

`Windows` On Windows: This example shows the environment profile for MyExtension, which is installed in `C:\Program Files\MyExtensions` on Windows 32–bit editions, or in `C:\Program Files(x86)\MyExtensions` on Windows 64–bit editions.

The script is called `MyExtension.cmd` and is stored in the working directory. The default location is `%ALLUSERSPROFILE%\Application Data\IBM\MQSI\common\ profiles` where the default setting for the environment variable`%ALLUSERSPROFILE%` depends on the operating system:

- On Windows XP and Windows Server 2003: `C:\Documents and Settings\All Users\`.
- On Windows Vista and Windows Server 2008: `C:\ProgramData\`.

The actual value might be different on your computer.

The Windows script contains the following content:

```
REM Added by MyExtension install, do not modify
set MQSI_LILPATH=%MQSI_LILPATH%;"C:\Program Files\MyExtension\bin"
```

`UNIX` On Linux: This example shows the environment profile for MyExtension, which is installed in `/opt/MyExtension`. The script is called `MyExtension.sh` and is stored in the working directory `/var/mqsi/common/profiles/`.

The Linux script contains the following content:

```
#!/bin/ksh
# Added by MyExtension install, do not modify
export MQSI_LILPATH=/opt/MyExtension/lil${MQSI_LILPATH:+":"${MQSI_LILPATH}}
```

You can test the following variables in the profile script, for example if you want to ensure that a user-defined extension runs only on a specific version of the broker:

**MQSI_FILEPATH**
The full path to the installed file for WebSphere Message Broker

**MQSI_WORKPATH**
The full path to the configuration data for WebSphere Message Broker

**MQSI_VERSION**
WebSphere Message Broker version, in the form `version.release.modification.fix`

**MQSI_VERSION_V**
> The value of WebSphere Message Broker major version

**MQSI_VERSION_R**
> The value of WebSphere Message Broker release

**MQSI_VERSION_M**
> The value of WebSphere Message Broker modification number

**MQSI_VERSION_F**
> The value of WebSphere Message Broker fix level

**Detecting installed versions of WebSphere Message Broker:**

A user-defined extension can detect which version of WebSphere Message Broker is installed.

Use the conditions described here to test for particular version or versions. If expected conditions are not met, a component might not have installed correctly, or might have become corrupted. Check the status of the installed component and the local logs to identify and resolve any errors.

*Detecting installed versions on Windows:*

Use the following instructions in your installer scripts on Windows to test for the following versions. To detect each version, look for the registry key given for each version. In the examples shown, x can be any integer.

**WebSphere Business Integration Message Broker Version 5.0 toolkit**
> `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\`
> `mqsi.studio\DisplayVersion = 5.x.x.x`

**WebSphere Business Integration Message Broker Version 5.0 runtime components**
> `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\`
> `mqsi.ib\DisplayVersion = 5.x.x.x`

**WebSphere Message Broker Version 6.0 toolkit**
> `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\`
> `WMBT60\DisplayVersion = 6.x.x.x`

**WebSphere Message Broker Version 6.1 (and later) toolkit**

> Check for the presence of the files `\IBM\Installation Manager\`
> `installed.xml` and `\IBM\Installation Manager\installRegistry.xml` in
> the working directory.

> The default working directory is `%ALLUSERSPROFILE%\Application`
> `Data\IBM\MQSI` where `%ALLUSERSPROFILE%` is the environment variable that
> defines the system working directory. The default directory depends on the
> operating system:
> - On Windows XP and Windows Server 2003: `C:\Documents and`
>   `Settings\All Users\Application Data\IBM\MQSI`
> - On Windows Vista and Windows Server 2008: `C:\ProgramData\IBM\MQSI`

> The actual value might be different on your computer; use
> `%ALLUSERSPROFILE%` to ensure that you access the correct location.

**WebSphere Message Broker Version 6.0 (and later) runtime components**
> Open the file `install.properties` in the working directory.

The default working directory is `%ALLUSERSPROFILE%\Application Data\IBM\MQSI` where `%ALLUSERSPROFILE%` is the environment variable that defines the system working directory. The default directory depends on the operating system:

- On Windows XP and Windows Server 2003: `C:\Documents and Settings\All Users\Application Data\IBM\MQSI`
- On Windows Vista and Windows Server 2008: `C:\ProgramData\IBM\MQSI`

The actual value might be different on your computer; use `%ALLUSERSPROFILE%` to ensure that you access the correct location.

Each line in the file is of the following format:

```
PATH_TO_INSTALLATION = VERSION_OF_INSTALLATION
```

For example, if you have installed Version 6.1 Fix Pack 3 in the default installation location, the line contains the following text:

```
C\:\\ Program\ Files\\IBM\\MQSI=6.1.0.3
```

The backslash character \ is interpreted as an escape character. It is inserted before each non-alphabetic and non-numeric character in the string to preserve the character. A colon, a space, and several backslash characters are escaped in this example.

Parse each line of the file to detect all the installed versions and the directory paths for the runtime components, and ignore all duplicates and non-existent directories.

*Detecting installed versions on Linux and UNIX systems:*

Linux and UNIX systems do not have a common packaging method: you must check which files are present in the file system. Look for the following files for each version of WebSphere Message Broker that you want to detect.

**WebSphere Business Integration Message Broker Version 5.0 runtime components**

> AIX   Check for the presence of `/usr/opt/mqsi/bin/mqsilist`. Also check that `/usr/opt/mqsi/bin/mqsiprofile` is not present.
>
> On Linux and other UNIX systems, check for the presence of `/opt/mqsi/bin/mqsilist`, and make sure that `opt/mqsi/bin/mqsiprofile` is not present.

**WebSphere Message Broker Version 6.0 toolkit**

> To detect Version 6 and later toolkits, look for the existence of `/etc/IBM/WebSphereMessageBrokersToolkit/products/com.ibm.wbmt`.
>
> To determine the version, use the following code example. Shell-script notation is used in this code: `'-e'` means `if file exists`.

```
if [ -e /etc/IBM/WebSphereMessageBrokersToolkit/products/com.ibm.webt  ]
  # Event Broker installed
     if [ -e `grep location /etc/IBM/WebSphereMessageBrokersToolkit/products/
  com.ibm.webt | sed 's/location=//'`/webt_prod/version.txt` ]
       # it is FP1 or greater
       get version from version.txt
     else
       #version is 6.0
     fi
  fi
  if [ -e /etc/IBM/WebSphereMessageBrokersToolkit/products/com.ibm.wmbt ]
    # Message Broker installed
```

```
                  if [ -e `grep location /etc/IBM/WebSphereMessageBrokersToolkit/products/
                  com.ibm.wmbt | sed 's/location=//'`/wmbt_prod/version.txt` ]
                    #It is FP1 or greater
                    get version from version.txt
                  else
                    #version is 6.0
                  fi
              fi
```

**WebSphere Message Broker Version 6.1 (and later) toolkit**

> Check for the presence of the files /var/ibm/InstallationManager/
> installed.xml and /var/ibm/InstallationManager/installRegistry.xml.

**WebSphere Message Broker Version 6.0 (and later) runtime components**

> To detect Version 6.0 and later runtime components, look for the file
> /var/mqsi/install.properties. Each line in this file contains an
> installation path and V.R.M.F version information.

> :

## Updating a user-defined extension

On all systems, you can change a user-defined extension file.

You must stop and restart the broker for your changes to show, However, you are
not required to stop and restart the broker in the following two scenarios:

- If you have created an execution group in the workbench, but have not yet
  deployed to it, you can add the .lil, .pdb, and .jar files to your chosen
  directory.
- If an object has already been deployed to the execution group that you want to
  use, add the .lil, .pdb, and .jar files to your chosen directory and use the
  mqsireload command to restart the group. You cannot overwrite an existing file
  on the Windows system when the broker is running because a file lock is put in
  place by the operating system.

These two scenarios must be used with caution, because any execution group that
is connected to the same broker also detects the new .lil, .pdb, and .jar files
when that execution group is restarted, or when an object is first deployed to it. If
you restart the broker, you must ensure that anyone with an interest in a particular
execution group is made aware that recent changes have been made to the broker.

These two scenarios assume that you have used either the mqsichangebroker
command or the mqsicreatebroker command to notify the broker of the directory
in which the user-defined extension files have been placed.

To change a user-defined extension file:
1. Stop the broker by using the mqsistop command.
2. Update or overwrite the .lil or .jar file.
3. Restart the broker by using the mqsistart command.

## Uninstalling a user-defined extension from the broker

Remove a user-defined extension file from the broker.
1. Stop the broker by using the mqsistop command.
2. Delete the .lil or .jar file from the appropriate directory.

   For C user-defined extensions:

| Platform | Location |
| --- | --- |
| Windows | `install_dir\bin` |
| Linux | `install_dir/lil` |
| UNIX | `install_dir/lil` |
| z/OS | `install_dir/lil` |

For Java user-defined nodes:

| Platform | Location |
| --- | --- |
| Windows | `install_dir\jplugin` |
| Linux | `install_dir/jplugin` |
| UNIX | `install_dir/jplugin` |
| z/OS | `install_dir/lil` |

3. Restart the broker by using the mqsistart command.

## Using error logging from a user-defined extension

Program user-defined extensions to write entries in the local error log.

In most circumstances, user-defined extensions should use exceptions to report errors. However, you can provide information about significant events, error or otherwise, for problem determination and operational purposes. The details that you supply are included in predefined message text that is extracted from a message source or catalog.

- In C code, use the utility function CciLog or CciLogW to report events. Two of the arguments that you pass to this function, `messageSource` and `messageNumber`, define the event source (catalog) and the integer representation of a message within that source, respectively.

  You can also write trace information, using CciUserTrace, CciUserTraceW, CciUserDebugTrace, and CciUserDebugTraceW when tracing and debugging is active.

- In Java code, use the class `MbService`, which provides static methods to log information to the event log. To log messages to the event log, package your messages into a standard Java resource bundle. You can use one of the three logging methods, passing in the resource bundle name and the message key. The message is fully resolved, and is then inserted as a single insert into the appropriate broker message as shown:

  – logInformation( ... ) - BIP4360 Java user-defined node information: *user message*

  – logWarning( ... ) - BIP4361 Java user-defined node warning: *user message*

  – logError( ... ) - BIP4362 Java user-defined node error: *user message*

You can write messages that are defined in the product message catalog (`BIPv610`), to which you can add your own text as an argument. If you prefer, you can create your own message catalog, so that you can create more complex messages, or share a message catalog with other applications. If you want to create your own message catalog, see "Creating message catalogs" on page 116.

- Windows  On Windows systems, messages are written to the Windows event log.

Developing user-defined extensions  **115**

- **Linux** **UNIX** **z/OS** On Linux, UNIX, and z/OS systems, messages are written to the SYSLOG facility.

The description here covers exceptions that are raised during normal message flow processing. You must also provide for exceptions that are raised when you deploy and configure a message flow. Messages that result from these configuration exceptions are reported back to the workbench for display to the workbench user. Create an appropriately-named Java properties file to contain your messages, then copy the file to each computer on which you are running the workbench, so that your messages can be displayed.

**Creating message catalogs:**

Create your own message catalogs to write tailored entries to the local error log.

In some error and other situations, you might choose to write information to the error log so that you can track what is happening in a message flow. You can use the Throw and Trace built-in nodes to generate entries in the log, or you can create your own nodes and user exits, and write entries in the log from your user-defined extensions.

You can write either or both of the following sets of messages:
- A fixed set of messages that are provided in the product message catalog. This set provides a range of numbers for Throw nodes (BIP3001 to BIP3001), and a second range for Trace nodes (BIP3051 to BIP3001). A third range (BIP2951 to BIP2999) is provided for the ESQL statements LOG and THROW.

  When you use these messages, you can also provide additional text that is displayed in the message text.
- Your own messages, created in your own message catalog. You can use this additional catalog to define specialized message content, and you can include variables or inserts that are determined by the code that generates the message. You can also share your own message catalog with other applications that are not associated with WebSphere Message Broker.

When you throw an exception from ESQL by using a THROW statement, the ESQL code adds an extra leading insert that contains the name of the current component. The rest of the inserts that are provided by the ESQL script follow this leading insert. Therefore, you must consider this insert when you are writing your own message catalog.

The instructions in this topic describe how to create message catalogs for C programs. If you want to create a Java resource bundle, refer to the documentation for the Java 2 Platform, Standard Edition.

Read the section appropriate to your broker operating system:
- "Building and installing a Windows message source"
- "Creating an XPG/4 catalog for Linux, UNIX, and z/OS" on page 118

*Building and installing a Windows message source:*

On Windows, you must create your additional message catalog as a DLL file. The DLL file contains definitions of your event messages, which the event viewer can display in a readable format, based on the event message written by your application. When you compile a message catalog, a header file is created that

defines symbolic values for each event message number you have created. You must include the header file in your application.

To create an event source for the Windows Event Log Service:

1. Create a message compiler input (`.mc`) file with the source for your event messages. Refer to the Microsoft® Developer Network Web site, and search on `.mc file` for details on the format of this input file.

2. Compile the message file to create a resource compiler input file:

   ```
   mc -v -w -s -h c:\mymessages -r c:\mymessages mymsg.mc
   ```

   where `c:\mymessages` is the location of the output files and *mymsg.mc* is the name of the input file.

   The message compiler produces an output header (`.h`) file that contains symbolic `#defines` that map to each message number that is coded in the input `.mc` file. Include this header file when you compile a user-defined extension source file that uses a utility function (for example, CciLog) to write an event message that you have defined. The `messageNumber` argument to utility function must use the appropriate value that is hash-defined in the output header file.

3. Compile the output file (`.rc`) from the message compiler to create a resource file (`.res`):

   ```
   RC /v output_file.rc
   ```

4. Create a resource DLL file from the `.res` file:

   ```
   LINK /DLL /NOENTRY resource_file.res
   ```

5. Append the location of the resource DLL file to the MQSI_CONSOLE_NLSPATH environment variable, for example:

   ```
   set MQSI_CONSOLE_NLSPATH=%MQSI_CONSOLE_NSPATH%;c:\messages
   ```

   You can do this by creating a custom environment script in your working directory. The default location is `%ALLUSERSPROFILE%\Application Data\IBM\MQSI\common\profiles` where `%ALLUSERSPROFILE%` is the environment variable that defines the system working directory. The default directory depends on the operating system:

   - On Windows XP and Windows Server 2003: `C:\Documents and Settings\All Users\Application Data\IBM\MQSI\common\profiles`
   - On Windows Vista and Windows Server 2008: `C:\ProgramData\IBM\MQSI\common\profiles`

   The actual value might be different on your computer.

6. Install the event source into the Windows Event Log Service:

   a. Start the Windows Registry Editor:

      ```
      regedit
      ```

   b. Create a new registry subkey for your user-defined extension under the existing structure:

   ```
   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application
   ```

   Right-click **Application** and select **New → Key**. The new key is created immediately under the Application key (not under the WebSphere Message Broker key). You must give the key the name that you specify for the `messageSource` on a utility function in your user-defined extension (for example, CciLog) or as the property of the built-in node that you have included in your message flow.

   Create the following values for this entry:

**EventMessageFile**

Set the value of this string to contain the fully qualified path for the DLL file that you have created to contain your messages. This entry represents the message catalog.

**TypesSupported**

Set the DWORD value to "7".

*Creating an XPG/4 catalog for Linux, UNIX, and z/OS:*

On Linux, UNIX, and z/OS systems, messages are written to the SYSLOG facility. If you want to use your own message catalog, you must create an XPG/4 message catalog.

The process for creating a message catalog (a `.cat` file) depends on the operating system on which you are creating it. The commands that you use are typically gencat (create or modify a message catalog) and dspcat (to display all or part of a message catalog). The gencat command merges text files that contain your message text, to create or modify a formatted catalog. The text files typically have a file extension of `.msg`.

You must append the location of the message catalog to the NLSPATH environment variable. You can use %L and %N to represent the locale and the catalog name, for example:

```
export NLSPATH=${NLSPATH}:${MY_INST_PATH}/messages/%L/%N:${MY_INST_PATH}/messages/En_US/%N
```

In this example, the English version is hardcoded later in the search path, ensuring that messages are displayed even in locales for which no `.cat` file exists.

The messages that you define in the `.msg` files can include variables that are substituted at run time. Such variables must be of the format *{number}*, where *{number}* is the message insert number, surrounded by braces. The first message insert is numbered 0. For example:

```
1234 "MSG1234E: \
Syntax Error. \n
The value '{0}' is not valid for property '{1}'.\n
Correct it and then reissue the command.\n"
```

If you create a message catalog on one operating system, you cannot port it to another operating system because the catalogs are binary-encoded. However, you can use the same `.msg` files as input to the gencat command on another system.

See the relevant information in the documentation for your operating system. For example:

- **AIX** For AIX, see the *Commands Reference* in the information center.

- **z/OS** For z/OS, see the *UNIX System Services Command Reference* in the z/OS V1R7.0 LibraryCenter.

You must also check the information about additional supported locales, if you want to use messages in locales other than US English.

# Part 2. Reference

# User-defined extensions

Reference material that supports the creation and management of your user-defined extensions.

This section contains the following information:
- "Sample node files"
- "Sample parser files" on page 123
- "C Header files" on page 123
- "C language user-defined node API" on page 124
- "C language user-defined parser API" on page 197
- "C user exit API" on page 251
- "C common API" on page 269
- "C skeleton code" on page 300
- Java user-defined extensions API
- "Utility function return codes and values" on page 303
- "Available parsers" on page 305
- "XML, MRM, and XMLNSC parser constants" on page 306
- "Trace logging from a user-defined C extension" on page 308
- "Multicultural support considerations for message catalogs" on page 309

## Sample node files

Several sample node files are provided on all platforms.

`Windows` On Windows, the following sample node files are in the *install_dir*\sample\extensions\nodes directory.

`Linux` On Linux, the following files are in the *install_dir*/sample/extensions/nodes directory.

`UNIX` On UNIX, the following files are in the *install_dir*/sample/extensions/nodes directory.

`z/OS` On z/OS, the following files are in the *install_dir*/sample/extensions/nodes directory.

| | |
|---|---|
| SwitchNode.c | C source file containing a sample implementation of a message processing node that routes a message to one of five output terminals, depending on the content. |
| SwitchNode.h | The header file for the SwitchNode.c file. |
| TransformNode.c | C source file containing a sample implementation of a simple fixed transformation of an input message into an output message. |
| TransformNode.h | The header file for the TransformNode.c file. |

| | |
|---|---|
| `BipSampPluginUtil.c` | Sample utility functions used by the Switch and Transform nodes. |
| `BipSampPluginUtil.h` | The header file for BipSampPluginNode and BipSampPluginUtil. |
| `NodeFactory.c` | Common C functions for `SwitchNode.c`, `TransformNode.c`, and `BipSampPluginUtil.c` |
| `NodeFactory.h` | The header file for `NodeFactory.c` |
| `Common.c` | Common C functions for `SwitchNode.c`, `TransformNode.c`, and `BipSampPluginUtil.c` |
| `Common.h` | The header file for `Common.c` |
| `PluginSample.add.xml` | A sample XML input message that you can use to test the C sample nodes. |
| `PluginSample.change.xml` | A sample XML input message that you can use to test the C sample nodes. |
| `PluginSample.delete.xml` | A sample XML input message that you can use to test the C sample nodes. |
| `JavaPlugin.add.xml` | A sample XML input message that you can use to test the Java sample nodes. |
| `JavaPlugin.change.xml` | A sample XML input message that you can use to test the Java sample nodes. |
| `JavaPlugin.delete.xml` | A sample XML input message that you can use to test the Java sample nodes. |
| `JavaPlugin.hold.xml` | A sample XML input message that you can use to test the Java sample nodes. |

**Windows** On Windows, the following sample node files are in the `install_dir`\sample\extensions\nodes directory.

**Linux** On Linux, the following files are in the `install_dir`/sample/Javaplugin/com/ibm/samples directory.

**UNIX** On UNIX, the following files are in the `install_dir`/sample/Javaplugin/com/ibm/samples directory.

**z/OS** On z/OS, the following files are in the `install_dir`/sample/Javaplugin/com/ibm/samples directory.

| | |
|---|---|
| `JavaSwitchPluginNode.java` | Java source file containing a sample implementation of a message processing node that routes a message to one of five output terminals, depending on the content. |
| `JavaTransformPluginNode.java` | Java source file containing a sample implementation of a simple fixed transformation of an input message into an output message. |

The files that the workbench needs to recognize the Switch node and Transform node are in the `install_dir`\sample\extensions\nodes\com.ibm.samples.nodes directory. You can add this directory to your workspace using the Update Manager, or you can copy it across to your workspace directory and restart the workbench

to see the nodes. The help files (`HelpContexts.xml`, `SwitchNode.htm`, and `TransformNode.htm`) demonstrate some features of Eclipse help by adding themselves into the main topic tree, referencing topics in the main tree, and so on.

GIF files that are used to represent the sample nodes in the workbench, which you can use, or replace with your own, are supplied. The GIF files come in three different sizes and can be found in individual directories under the `sample\extensions\nodes\com.ibm.samples.nodes\icons\full\` directory.

## SupportPacs

Many other sample nodes are available as SupportPac offerings. For a complete list of available SupportPac offerings see WebSphere MQ SupportPacs Web page.

# Sample parser files

Several sample parser files are provided on all platforms, to help you to create your own parsers.

| Windows | On Windows, the following sample parser files are in the `install_dir\sample\extensions\parser` directory.

| Linux | On Linux, the following sample parser files are in the `install_dir/sample/extensions/parser` directory:

| UNIX | On UNIX, the following sample parser files are in the `install_dir/sample/extensions/parser` directory:

| z/OS | On z/OS, the following sample parser files are in the `install_dir/sample/extensions/parser` directory:

| `BipSampPluginParser.c` | C source file containing sample implementations of a simple pseudo-XML parser. |
| `BipSampPluginParser.h` | The header file for the `BipSampPluginParser.c` file. |

## SupportPacs

Many other sample parsers are available as SupportPac offerings. For a complete list of available SupportPacs, see http://www.ibm.com/software/integration/support/supportpacs/.

# C Header files

The C interfaces are defined by the following header files.
- **BipCni.h** contains functions for user-defined nodes that have been written in C. For a list of functions, refer to the "C language user-defined node API" on page 124.
- **BipCpi.h** contains functions for user-defined parsers that have been written in C. For a list of functions, refer to the "C language user-defined parser API" on page 197.
- **BipCci.h** contains utility functions common to both user-defined nodes and parsers that have been written in C. For a list of functions, refer to "C common

utility functions" on page 271. This file also contains definitions for utility function return codes and values. For more information, see "Utility function return codes and values" on page 303.

- **BipCos.h** contains operating system specific definitions for user-defined nodes that have been written in C.

# C language user-defined node API

Learn about the different types of call provided by the API.

The C language user-defined node API consists of:

1. A set of implementation functions that provide the functionality of the user-defined node. These functions are called by the broker. The implementation functions are mandatory, and if they are not supplied by the developer, an exception is thrown at run time.

2. A set of utility functions that create resources in the broker, or request a service of the broker. These utility functions are called by a user-defined node.

Most of the utilities are shared by all types of node; however, a few are specific to input nodes. This restriction is marked in the text.

These functions are defined in the `BipCni.h` header file.

This section covers the following topics:

**"C node implementation functions"**

**"C node utility functions" on page 125**

# C node implementation functions

The user-defined node implements a function interface for the broker to call during runtime operation. This interface includes functions to create a local context whenever a node instance is created, functions to set and retrieve attribute values, the function to perform the processing of the node itself, and functions to examine messages.

The following functions are provided:

**Mandatory function**
"cniCreateNodeContext" on page 142

**Optional and conditional functions**
- "cniDeleteNodeContext" on page 146
- Either "cniEvaluate" on page 159 (for message processing and output nodes), or "cniRun" on page 175 (for input nodes)
- "cniGetAttribute" on page 163
- "cniGetAttribute2" on page 164
- "cniGetAttributeName" on page 165
- "cniGetAttributeName2" on page 166
- "cniSetAttribute" on page 180

These implementation functions are called by the broker, and implemented by the node.

For certain implementation functions, you might need to specify the name of a parser that is supplied with WebSphere Message Broker. If so, you must use the correct class name of the parser. The following table provides a summary of the parsers, root element names, and class names for different headers.

| Parser | Root element name | Class name |
| --- | --- | --- |
| BLOB | BLOB | NONE |
| IDOC | IDOC | IDOC |
| JMSMap | JMSMap | JMS_MAP |
| JMSStream | JMSStream | JMS_STREAM |
| MIME | MIME | MIME |
| MQCFH | MQPCF | MQPCF |
| MQCIH | MQCIH | MQCICS |
| MQDLH | MQDLH | MQDEAD |
| MQIIH | MQIIH | MQIMS |
| MQMD | MQMD | MQHMD |
| MQMDE | MQMDE | MQHMDE |
| MQRFH | MQRFH | MQHRF |
| MQRFH2 | MQRFH2 | MQHRF2 |
| MQRMH | MQRMH | MQHREF |
| MQSAPH | MQSAPH | MQHSAP |
| MQWIH | MQWIH | MQHWIH |
| MRM | MRM | MRM |
| Properties | Properties | PropertyParser |
| SMQ_BMH | SMQ_BMH | SMQBAD |
| XML | XML | xml |
| XMLNS | XMLNS | xmlns |
| XMLNSC | XMLNSC | xmlnsC |

# C node utility functions

A user-defined node created in the C programming language can create or define broker objects, such as node factories, nodes, and terminals by using system-provided functions.

Functions are also provided to send messages to an output terminal for propagation to connected nodes, and to examine message content.

These utility functions are called by the node, and implemented by the broker.

**Initialization and resource creation**

- "cniCreateNodeFactory" on page 143
- "cniDefineNodeClass" on page 144
- "cniDispatchThread" on page 147 (for input nodes only)
- "cniCreateInputTerminal" on page 140
- "cniCreateOutputTerminal" on page 143
- "cniIsTerminalAttached" on page 170

# cniAddAfter

Use this function to add an unattached syntax element after a specified syntax element. The currently unattached syntax element, and all child elements it possesses, is connected to the syntax element tree after the specified target element.

The added element becomes the *next sibling* of the target element. The target element must be attached to a tree (that is, it must have a parent element).

## Syntax

```
void cniAddAfter(
int*        returnCode,
CciElement* targetElement,
CciElement* newElement);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
The address of the target syntax element object (input).

**newElement**
The address of the new syntax element object that is to be added to the tree structure (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniAddasFirstChild

Use this function to add an unattached syntax element as the first child of a specified syntax element. The currently unattached syntax element, and all child elements it possesses, is connected to the syntax element tree as the *first child* of the specified target element. The target element need not be attached to a tree.

## Syntax

```
void cniAddAsFirstChild(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
   The return code from the function (output).

   Possible return codes are:
   • CCI_SUCCESS
   • CCI_EXCEPTION
   • CCI_INV_ELEMENT_OBJECT

**targetElement**
   The address of the target syntax element object (input).

**newElement**
   The address of the new syntax element object that is to be added to the tree structure (input).

## Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniAddasLastChild

Use this function to add an unattached syntax element as the last child of a specified syntax element. The currently unattached syntax element, and all child elements it possesses, is connected to the syntax element tree as the *last child* of the specified target element. The new element need not be attached to a tree.

## Syntax

```
void cniAddAsLastChild(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
   The return code from the function (output).

   Possible return codes are:
   • CCI_SUCCESS
   • CCI_EXCEPTION

- CCI_INV_ELEMENT_OBJECT

**targetElement**
   The address of the target syntax element object (input).

**newElement**
   The address of the new syntax element object that is to be added to the tree structure (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniAddBefore

Use this function to add an unattached syntax element before a specified syntax element. The currently unattached syntax element, and all child elements it possesses, is connected to the syntax element tree before the specified target element.

The newly added element becomes the *previous sibling* of the target element. The target element must be attached to a tree (that is, it must have a parent element).

### Syntax

```
void cniAddBefore(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

### Parameters

**returnCode**
   The return code from the function (output).

   Possible return codes are:
   - CCI_SUCCESS
   - CCI_EXCEPTION
   - CCI_INV_ELEMENT_OBJECT

**targetElement**
   The address of the target syntax element object (input).

**newElement**
   The address of the new syntax element object that is to be added to the tree structure (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniBufferByte

Use this function to get a single byte from the data buffer associated with (and owned by) the message object specified in the message argument. The value of the index argument indicates which byte in the byte array is to be returned.

## Syntax

```
CciByte cniBufferByte(
  int*       returnCode,
  CciMessage* message,
  CciSize     index);
```

## Parameters

**returnCode**
>
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_MESSAGE_OBJECT

**message**
>
> The address of the message object for which the size of the data buffer is to be returned (input).

**index**
>
> The offset to use as an index into the buffer (input).

## Return values

The requested byte is returned. If an error occurred, the *returnCode* parameter indicates the reason for the error.

# cniBufferPointer

Use this function to get a pointer to the data buffer associated with (and owned by) the message object specified in the message argument. This function is typically used by output nodes.

## Syntax

```
const CciByte* cniBufferPointer(
  int*        returnCode,
  CciMessage*  message);
```

## Parameters

**returnCode**
>
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_MESSAGE_OBJECT

**message**
>
> The address of the message object for which the address of the data buffer is to be returned (input).

## Return values

If successful, the address of the data buffer is returned. Otherwise, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniBufferSize

Use this function to get the size of the data buffer associated with (and owned by) the message object specified in the message argument.

## Syntax

```
CciSize cniBufferSize(
  int*        returnCode,
  CciMessage* message);
```

## Parameters

**returnCode**
    The return code from the function (output).

    Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT

**message**
    The address of the message object for which the size of the data buffer is to be returned (input).

## Return values

The size of the buffer in bytes, or zero if no buffer exists. If an error occurred, (CCI_FAILURE) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCopyElementTree

Use this function to copy a part of the element tree from the source element to the target element. Only the child elements of the source element are copied. All existing child elements of the target element are deleted, and are replaced by the child elements of the source element.

If the target element has not been fully parsed, or represents an unparsed bit stream, the cniCopyElementTree function results in a parse of the target element before its child elements are detached. The function therefore ensures consistency in message-tree formatting so that all references to detached fields by cciElements remain valid. Therefore, if a parsing exception occurs during the execution of the cniCopyElementTree function the cause might be a problem with either the target element or the source element.

## Syntax

```
void cniCopyElementTree(
  int*        returnCode,
  CciElement* sourceElement,
  CciElement* targetElement);
```

## Parameters

**returnCode**
    The return code from the function (output).

    Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT

**sourceElement**
    The address of the source syntax element object (input).

**targetElement**
    The address of the target syntax element object (input).

## Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example
```
cniCopyElementTree(&rc, inRootElement, outRootElement);
```

# cniCreateElementAfter

Use this function to create a new syntax element and insert it after the specified syntax element. The new element becomes the *next sibling* of the specified element.

Do not use cniCreateElementAfter when you create a message body folder (such as XML, XMLNS, MRM, BLOB), because this function does not associate an owning parser with the folder. To create a message body folder, you can use one of the following functions:

    cniCreateElementAsFirstChildUsingParser

    cniCreateElementAsLastChildUsingParser

    cniCreateElementAfterUsingParser

    cniCreateElementBeforeUsingParser

When the message body folder has been created, you can use cniCreateElementAfter to create elements under the folder. You can use cniCreateElementAfter because the parser, which is associated with the message body folder, is inherited.

## Syntax
```
CciElement* cniCreateElementAfter(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**
    The return code from the function (output).

    Possible return codes are:
    - CCI_SUCCESS
    - CCI_EXCEPTION
    - CCI_INV_ELEMENT_OBJECT

**targetElement**
    The address of the element object (input).

## Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCreateElementAfterUsingParser

Use this function to create a syntax element, insert it after the specified syntax element, and associate it with the specified parser class name. The new element becomes the *next sibling* of the specified element.

A portion of the syntax element tree that is owned by a parser can *only* have its effective root at the first generation of elements (that is, as *immediate children of root*). The user-defined node interface does not restrict the ability to create a subtree that appears to be owned by a different parser. However, it is not possible to serialize these element trees into a bit stream when producing an output message.

If you specify the name of a parser supplied with WebSphere Message Broker, you must use the correct class name of the parser.

The internal name for the BLOB parser is *none*. Therefore, if you use this function to create a BLOB parser folder, the associated parser name should be *none*.

## Syntax

```
CciElement* cniCreateElementAfterUsingParser(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  parserClassName);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_PARSER_NAME

**TargetElement**
The address of the element object (input).

**parserClassName**
The name of the parser class (input).

## Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCreateElementAsFirstChild

Use this function to create a syntax element as the first child of the specified syntax element.

Do not use cniCreateElementAsFirstChild when creating a message body folder (such as XML, XMLNS, MRM, BLOB), because it does not associate an owning parser with the folder. To create a message body folder, you can use one of the following functions:

cniCreateElementAsFirstChildUsingParser

cniCreateElementAsLastChildUsingParser

cniCreateElementAfterUsingParser

cniCreateElementBeforeUsingParser

When the message body folder has been created, you can call
cniCreateElementAsFirstChild to create elements under the folder. You can use
cniCreateElementAsFirstChild because the parser, which is associated with the
message body folder, is inherited.

### Syntax

```
CciElement* cniCreateElementAsFirstChild(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:

- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**

The address of the element object (input).

### Return values

If successful, the address of the new element object is returned. Otherwise, a value
of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the
reason for the error.

## cniCreateElementAsFirstChildUsingParser

Use this function to create a syntax element as the first child of the specified
syntax element, and associates it with the specified parser class name.

A portion of the syntax element tree that is owned by a parser can *only* have its
effective root at the first generation of elements (that is, as *immediate children of
root*). The user-defined node interface does not restrict the ability to create a
subtree that appears to be owned by a different parser. However, it is not possible
to serialize these element trees into a bit stream when producing a message.

If you specify the name of a parser supplied with WebSphere Message Broker, you
must use the correct class name of the parser.

The internal name for the BLOB parser is *none*. Therefore, if you use this function
to create a BLOB parser folder, the associated parser name should be *none*.

### Syntax

```
CciElement* cniCreateElementAsFirstChildUsingParser(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  parserClassName);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_PARSER_NAME

**targetElement**
The address of the element object (input).

**parserClassName**
The name of the parser class (input).

### Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCreateElementAsLastChild

Use this function to create a syntax element as the last child of the specified syntax element.

Do not use cniCreateElementAsLastChild when creating a message body folder (such as XML, XMLNS, MRM, BLOB), because it does not associate an owning parser with the folder. To create a message body folder, you can use one of the following functions:

cniCreateElementAsFirstChildUsingParser

cniCreateElementAsLastChildUsingParser

cniCreateElementAfterUsingParser

cniCreateElementBeforeUsingParser

When the message body folder has been created, you can use cniCreateElementAsLastChild to create elements under the folder. You can use cniCreateElementAsLastChild because the parser, which is associated with the message body folder, is inherited.

### Syntax
```
CciElement* cniCreateElementAsLastChild(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
The address of the element object (input).

### Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned and the *returnCode* parameter indicates the reason for the error.

### Example

```
CciElement* lastChild = cniCreateElementAsLastChild(&rc, outRootElement);
cniSetElementName(&rc, lastChild, elementName);
```

# cniCreateElementAsLastChildFromBitstream

Use this function to create a syntax element tree as the last child of the specified syntax element, and associates it with the specified parser. The syntax element tree is populated by parsing the specified bit stream.

During the execution of this function, the bit stream is copied, so the caller can free or reuse the memory allocated to hold the original bit stream. You can use this function only to create a message body, that is, the last child of the message root. An output message must already exist.

The root element of this output message should be passed in as the target element parameter. Because this call is designed only to be used to create a message body, you cannot use it to build successive elements. For example, it should not be used to create an MQRFH2 header as the last child of root, then an XML message as the last child of root, after the MQRFH2 header.

### Syntax

```
CciElement* cniCreateElementAsLastChildFromBitstream (
  int*                      returnCode,
  CciElement*               targetElement,
  const struct CciByteArray* value,
  const CciChar*            parserClassName,
  CciChar*                  messageType,
  CciChar*                  messageSet,
  CciChar*                  messageFormat,
  int                       encoding,
  int                       ccsid,
  int                       options);
```

### Parameters

**returnCode**
The return code from the function (output). Specifying a NULL pointer signifies that the node does not want to deal with errors. If input is not NULL, the output signifies the success status of the call. Any exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. Call cciGetLastExceptionData for details of the exception.

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_PARSER_NAME
- CCI_INV_DATA_POINTER

**targetElement**
The syntax element under which the new syntax element tree is created (input). This parameter must be the message root.

**parserClassName**

The name of the parser class to use to parse the bit stream (input). You must use the same parser that was used to parse the whole bit stream.

**value**

A pointer to a CciByteArray struct containing a pointer to the bit stream to be parsed, and also the size in CciBytes of this bit stream (output).

**messageType**

The message type definition used to create the element tree from the bit stream (input). A NULL pointer means that this parameter is ignored. Also, if the parser specified has no interest in this value, for example if it is a generic XML parser, the parameter is ignored.

**messageSet**

The message set definition used to create the element tree from the bit stream (input). A NULL pointer means that this parameter is ignored. Also, if the parser specified has no interest in this value, for example if it is a generic XML parser, the parameter is ignored.

**messageFormat**

The format used to create the element tree from the bit stream (input). A NULL pointer means that this parameter is ignored. Also, if the parser specified has no interest in this value, for example if it is a generic XML parser, the parameter is ignored.

**encoding**

The encoding to use when parsing the bit stream (input). This parameter is mandatory. You can specify a value of 0 to indicate that the queue manager's encoding should be used.

**ccsid**

The coded character set identifier to use when parsing the bit stream (input). This parameter is mandatory. You can specify a value of 0 to indicate that the queue manager's ccsid should be used.

**options**

This is reserved for future use. You must specify a value of 0 to maintain forward compatibility.

## Return values

If successful, the address of the new element object is returned. Otherwise, a value zero (CCI_NULL_ADDR) is returned, and the return code parameter indicates the reason for the error. If an exception occurs during execution, *returnCode* is set to CCI_EXCEPTION

## Example

```
outMQMD = cniCreateElementAsFirstChildUsingParser(&rc,
                                          outRootElement,
                                          CciString("MQHMD",BIP_DEF_COMP_CCSID));
checkRC(rc);

cniCopyElementTree(&rc, inMQMD, outMQMD);
checkRC(rc);

outBlobRoot = cniCreateElementAsLastChildFromBitstream(
                                          &rc,
                                          outRootElement,
                                          &bitstream,
                                          inParserClassName,
                                          messageType,
```

```
                                          messageSet,
                                          messageFormat,
                                          encoding,
                                          ccsid,
                                          0);


    checkRC(rc);
    ...

    return;
}
```

# cniCreateElementAsLastChildUsingParser

Use this function to create a syntax element as the last child of the specified syntax element, and associate it with the specified parser class name.

A portion of the syntax element tree that is owned by a parser can *only* have its effective root at the first generation of elements (that is, as *immediate children of root*). The user-defined node interface does not restrict the ability to create a subtree that appears to be owned by a different parser. However, it is not possible to serialize these element trees into a bit stream when producing a message.

If you specify the name of a parser supplied with WebSphere Message Broker, you must use the correct class name of the parser. See "C node implementation functions" on page 124 for a list of the supplied parsers.

The internal name for the BLOB parser is *none*. Therefore, if you use this function to create a BLOB parser folder, the associated parser name must be *none*.

## Syntax

```
CciElement* cniCreateElementAsLastChildUsingParser(
  int*           returnCode,
  CciElement*    targetElement,
  const CciChar* parserClassName);
```

## Parameters

**returnCode**
    The return code from the function (output).

    Possible return codes are:
    • CCI_SUCCESS
    • CCI_EXCEPTION
    • CCI_INV_ELEMENT_OBJECT
    • CCI_INV_PARSER_NAME

**targetElement**
    The address of the element object (input).

**parserClassName**
    The name of the parser class (input).

## Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example

```
cniElementName(&rc, firstChild, elementName);
CciElementType type = cniElementType(&rc, firstChild);
CciElement* lastChild = cniCreateElementAsLastChildUsingParser(
                                                    &rc,
                                                    outRootElement,
                                                    parserName);

cniSetElementName(&rc, lastChild, elementName);
cniSetElementType(&rc, lastChild, elementType);
```

# cniCreateElementBefore

Use this function to create a new syntax element and insert it before the specified syntax element. The new element becomes the *previous sibling* of the specified element, and shares the same parent element.

DO not use cniCreateElementBefore when creating a message body folder (such as XML, XMLNS, MRM, BLOB), because it does not associate an owning parser with the folder. To create a message body folder, you can use one of the following functions:

cniCreateElementAsFirstChildUsingParser

cniCreateElementAsLastChildUsingParser

cniCreateElementAfterUsingParser

cniCreateElementBeforeUsingParser

When the message body folder has been created, `cniCreateElementBefore` can be used to create elements under the folder. `cniCreateElementBefore` can be used because the parser, which is associated with the message body folder, is inherited.

### Syntax

```
CciElement* cniCreateElementBefore(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
The address of the target element object (input).

### Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCreateElementBeforeUsingParser

Use this function to create a syntax element, insert it before the specified syntax element, and associate it with the specified parser class name. The new element becomes the *previous sibling* of the specified element.

A portion of the syntax element tree that is owned by a parser can *only* have its effective root at the first generation of elements (that is, as *immediate children of root*). The user-defined node interface does not restrict the ability to create a subtree that appears to be owned by a different parser. However, it is not possible to serialize these element trees into a bit stream when producing a message.

If you specify the name of a parser supplied with WebSphere Message Broker, you must use the correct class name of the parser.

The internal name for the BLOB parser is *none*. Therefore, if you use this function to create a BLOB parser folder, the associated parser name should be *none*.

### Syntax

```
CciElement* cniCreateElementBeforeUsingParser(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  parserClassName);
```

### Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT
> - CCI_INV_PARSER_NAME

**targetElement**
> The address of the element object (input).

**parserClassName**
> The name of the parser class (input).

### Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniCreateInputTerminal

Use this function to create an input terminal on an instance of a node object and return the address of the terminal object that was created.

The terminal object is destroyed by the broker when its owning node is destroyed. You must call this function only from within the implementation function cniCreateNodeContext.

### Syntax

```
CciTerminal* cniCreateInputTerminal(
  int*       returnCode,
  CciNode*   nodeObject,
  CciChar*   name);
```

### Parameters

**returnCode**
> The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_NODE_OBJECT
- CCI_INV_TERMINAL_NAME

**nodeObject**

Specifies the address of the instance of the node object on which the input terminal is to be created (input). The handle is passed to the cniCreateNodeContext function.

**name**

Specifies a name for the terminal being created (input).

### Return values

If successful, the address of the node terminal object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned.

### Example

```
entry->handle = cniCreateInputTerminal(
                                    &rc,
                                    context->nodeObject,
                                    (CciChar*)terminalName);
```

## cniCreateMessage

Use this function to create an output message object. For every call to this function, you must include a matching call to cniDeleteMessage to return allocated resources when the processing on the output message has been completed.

### Syntax

```
CciMessage* cniCreateMessage(
  int*              returnCode,
  CciMessageContext* messageContext);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_FAILURE
- CCI_EXCEPTION
- CCI_INV_MESSAGE_CONTEXT

**messageContext**

The address of the context for the message (input). Use cniGetMessageContext to get the context from an incoming message; for example, one received in the cniEvaluate function.

### Return values

If successful, the address of the message object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example
```
outMsg = cniCreateMessage(&rc, cniGetMessageContext(&rc, message));
```

# cniCreateNodeContext

This function creates a context for an instance of a node object. It is called by the broker whenever an instance of a node object is constructed. Nodes are constructed when a message flow is deployed by the broker, or when the execution group is started.

The responsibilities of the node, when created, are to:

1. (Optional) Verify that the name of the node specified in the *nodeName* parameter is supported by the factory.
2. Allocate any node instance specific data areas that might be required (for example: context, attribute data, and terminals).
3. Perform all additional resource acquisition or initialization that might be required for the processing of the node.
4. Return the address of the context to the calling function. Whenever an implementation function for this node instance is called, the appropriate context is passed as an argument to that function. Therefore, a user-defined node developed in C does not have to maintain its own static pointers to per-instance data areas.

| Defined In | Type | Member |
|---|---|---|
| CNI_VFT | Mandatory | iFpCreateNodeContext |

### Syntax
```
CciContext* cniCreateNodeContext(
  CciFactory*  factoryObject,
  CciChar*     nodeName,
  CciNode*     nodeObject);
```

### Parameters

**factoryObject**
   The address of the factory object that owns the node being created (input).

**nodeName**
   The name of the node being created (input).

**nodeObject**
   The address of the node object that has just been created (input).

### Return values

If successful, the address of the node context is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned.

### Example
```
static char* functionName = (char *)"_Switch_createNodeContext()";
NODE_CONTEXT_ST* p;

/* Allocate a pointer to the local context */
p = (NODE_CONTEXT_ST *)malloc(sizeof(NODE_CONTEXT_ST));

if (p) {

  /* Clear the context area */
```

```
        memset(p, 0, sizeof(NODE_CONTEXT_ST));

        /* Save our node object pointer in our context */
        p->nodeObject = nodeObject;

        /* Save our node name */
        CciCharNCpy((CciChar*) &p->nodeName, nodeName, MAX_NODE_NAME_LEN);
}
  else
  /* Handle errors */
```

# cniCreateNodeFactory

Use this function to create a node factory in the broker. A single instance of the named message flow node factory is created.

This function must be called only in the initialization function bipGetMessageFlowNodeFactory, which is called when the LIL is loaded by the message broker. If cniCreateNodeFactory is called at any other time, the results are unpredictable.

## Syntax

```
CciFactory* cniCreateNodeFactory(
  int*      returnCode,
  CciChar*  name);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_FAILURE
- CCI_EXCEPTION
- CCI_INV_FACTORY_NAME
- CCI_INV_OBJECT_NAME

**name**
The name of the factory being created (input).

## Return values

If successful, the address of the node factory object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

## Example

```
factoryObject = cniCreateNodeFactory(0, (unsigned short *)constPluginNodeFactory);
if (factoryObject == CCI_NULL_ADDR) {

/* Handle errors */
```

# cniCreateOutputTerminal

Use this function to create an output terminal on an instance of a node object and return the address of the terminal object that was created.

The terminal object is destroyed when its owning node is destroyed. You must call this function only from within the implementation function cniCreateNodeContext.

### Syntax

```
CciTerminal* cniCreateOutputTerminal(
  int*      returnCode,
  CciNode*  nodeObject,
  CciChar*  name);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_FAILURE
- CCI_EXCEPTION
- CCI_INV_NODE_OBJECT
- CCI_INV_TERMINAL_NAME

**nodeObject**

The address of the instance of the node object on which the output terminal is to be created (input). The handle is passed to the cniCreateNodeContext function.

**name**

The name of the terminal being created (input).

### Return values

If successful, the address of the node terminal object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned.

### Example

```
entry->handle = cniCreateOutputTerminal(
                                &rc,
                                context->nodeObject
                                (CciChar*)terminalName);
```

## cniDefineNodeClass

Use this function to define a node class, as specified by the *name* parameter, which is supported by the node factory specified as the*factoryObject* parameter.

This function is called by the node during execution of bipGetMessageFlowNodeFactory, when the LIL file is loaded.

If both cniGetAttribute and cniGetAttribute2, or cniGetAttributeName and cniGetAttributeName2 are implemented, cniDefineNodeClass fails with CCI_INV_IMPL_FUNCTION.

### Syntax

```
void cniDefineNodeClass(
  int*         returnCode,
  CciFactory*  factoryObject,
  CciChar*     name,
  CNI_VFT*     functbl);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_FACTORY_OBJECT
- CCI_INV_NODE_NAME
- CCI_INV_OBJECT_NAME
- CCI_INV_VFTP
- CCI_MISSING_IMPL_FUNCTION
- CCI_NAME_EXISTS

**factoryObject**
> The address of the factory object that supports the named node. The address is returned from cniCreateNodeFactory (input).

**name**
> The name of the node to be defined. The name of the node must end with the characters Node (input).

**functbl**
> The address of the CNI_VFT structure that contains pointers to the node implementation functions (input). Here is an example of a function table:

```
vftable.iFpCreateNodeContext = _Transform_createNodeContext;
vftable.iFpDeleteNodeContext = _deleteNodeContext;
vftable.iFpGetAttributeName2 = _getAttributeName2;
vftable.iFpSetAttribute      = _setAttribute;
vftable.iFpGetAttribute2     = _getAttribute2;
vftable.iFpEvaluate          = _Transform_evaluate; /* if not an input node */
vftable.iFRun                = _run                 /* if an input node */
```

> You would typically define only one of the last 2 entries, that is, you define `vftable.iFpEvaluate = _Transform_evaluate;` for a message processing node, or you define `vftable.iFpRun = _run;` for an input node.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniDeleteMessage

Use this function to delete the specified message object. For every call to the cniCreateMessage function, you must include a matching call to cniDeleteMessage to return allocated resources when the processing on the output message has been completed.

### Syntax

```
void cniDeleteMessage(
  int*        returnCode,
  CciMessage* message);
```

### Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION

- CCI_INV_MESSAGE_OBJECT

**message**
> The address of the message object to be deleted (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example

```
cniDeleteMessage(0, outMsg);
```

# cniDeleteNodeContext

This function deletes any context for an instance of a user-defined node object. It is called by the broker whenever an instance of a node object is destroyed, when a message flow is deleted, or when a configuration is redeployed.

A message flow node might also be deleted when reconfiguring or redeploying a broker.

The responsibilities of the node are to:
1. Release all node instance specific data areas (such as context) that were acquired at construction or during node processing.
2. Release all additional resources that might have been acquired for the processing of the node.

| Defined In | Type | Member |
|------------|------|--------|
| CNI_VFT | Optional | iFpDeleteNodeContext |

### Syntax

```
void cniDeleteNodeContext(CciContext* context);
```

### Parameters

**context**
> The address of the context for the instance of the node, as created and returned by the cniCreateNodeContext function (input).

### Example

```
void _deleteNodeContext(
  CciContext* context
){
  static char* functionName = (char *)"_deleteNodeContext()";

  return;
}
```

# cniDetach

Use this function to detach the specified syntax element from the syntax element tree. The element is detached from its parent and siblings, but all child elements are left attached.

### Syntax

```
void cniDetach(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**

The address of the syntax element object to be detached (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniDispatchThread

Use this function to dispatch a new message flow thread to call another thread instance to run the user-defined message flow input node.

This message flow thread is allocated from a pool of threads maintained for each message flow, under control of the Additional Instances property of the message flow. If no threads are available because they are all in use, CCI_SUCCESS is returned, and *returnCode* is set to CCI_NO_THREADS_AVAILABLE. This value is not an error, but represents one of the following causes:
- The message flow was not configured to run with additional threads.
- All additional threads configured are currently running.

The cniDispatchThread function can be called only from an input node. If it is called at any other time, CCI_FAILURE is returned and *returnCode* is set to CCI_INV_NODE_ENV.

### Syntax

```
int cniDispatchThread(
  int*       returnCode,
  CciNode*   nodeObject);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_NO_THREADS_AVAILABLE
- CCI_INV_NODE_OBJECT
- CCI_INV_NODE_ENV

**nodeObject**

The address of the node object that is run when the broker creates or reuses the thread. This parameter is passed to the node when its cniCreateNodeContext implementation function is called (input).

### Return values

- If a thread was successfully allocated, CCI_SUCCESS is returned, and *returnCode* is set to CCI_SUCCESS.
- If a thread could not be dispatched because insufficient threads are available in the message flow thread pool to satisfy the request, CCI_SUCCESS is returned and *returnCode* is set to CCI_NO_THREADS_AVAILABLE.
- If the function was not called in an input node, CCI_FAILURE is returned, and *returnCode* is set to CCI_INV_NODE_ENV.
- For all other error conditions, CCI_FAILURE is returned, and *returnCode* indicates the reason for the error.

### Example

```
cniDispatchThread(&rcDispatch, ((NODE_CONTEXT_ST *)context)->nodeObject);
```

# cniElementAsBitstream

Use this function to get the bitstream representation of the specified element.

The parser that is associated with the element serializes the element and all its child elements. The result is copied to memory allocated by the caller. In the special case where all the options that are specified match those of the original bit stream, for example a bit stream that is read from a WebSphere MQ queue by the MQInput node, and the message has not been modified since receiving the original bit stream, this original bit stream is copied into the allocated memory. In this case, the parser is not required to parse and reserialize the message.

The algorithm that is used to generate the bit stream depends on the parser that is used, and the options that are specified. All parsers support the following modes:

- RootBitStream, in which the algorithm that generates the bit stream is the same as that used by an output node. In this mode, a meaningful result is obtained only if the element pointed to is at the head of a subtree with an appropriate structure.
- EmbeddedBitStream, in which not only is the algorithm that generates the bit stream the same as that used by an output node, but also the following elements are determined, if not explicitly specified, in the same way as the output node. Therefore they are determined by searching the previous siblings of *element* on the assumption that these elements represent headers:
  - Encoding
  - CCSID
  - Message set
  - Message type
  - Message format

  In this way, the algorithm for determining these properties is essentially the same as that used for the ESQL BITSTREAM function.

Some parsers also support another mode, FolderBitStream, which generates a meaningful bit stream for any subtree, provided that the field pointed to represents a folder.

### Syntax

```
CciSize cniElementAsBitstream(
  int*                    returnCode,
  CciElement*             element,
  const struct CciByteArray* value,
```

```
CciChar*               messageType,
CciChar*               messageSet,
CciChar*               messageFormat,
int                    encoding,
int                    ccsid,
int                    options);
```

## Parameters

**returnCode**

The return code from the function (output). If you specify a NULL pointer on input, the value indicates that the node does not handle errors. If input is not NULL, the output signifies the success status of the call. Any exceptions that are produced during the execution of this call are sent to the next upstream node in the flow. Call cciGetLastExceptionData for details of the exception.

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**element**

The syntax element to be serialized (input.)

**value**

A pointer to a CciByteArray struct that contains a pointer to a region of memory allocated by the caller, and the size in CciBytes of this memory (output).

**messageType**

The message type definition that is used to create the bit stream from the element tree (input). If you specify a NULL pointer, the parameter is ignored. The parameter is also ignored if the value is not relevant to the parser associated with the element; for example, a generic XML parser.

**messageSet**

The message set definition that is used to create the bit stream from the element tree (input). If you specify a NULL pointer, the parameter is ignored. The parameter is also ignored if the value is not relevant to the parser associated with the element; for example, a generic XML parser.

**messageFormat**

The format that is used to create the bit stream from the element tree (input). If you specify a NULL pointer, the parameter is ignored. The parameter is also ignored if the value is not relevant to the parser associated with the element; for example, a generic XML parser.

**encoding**

The encoding to use when writing the bit stream (input). This parameter is mandatory. You can specify a value of 0 to indicate that the encoding for the queue manager must be used.

**ccsid**

The coded character set identifier to use when writing the bit stream (input). This parameter is mandatory. If you specify a value of 0, the CCSID of the queue manager is used. A CCSID of -1 indicates that the bit stream is

generated by using CCSID information contained in the subtree consisting of the field pointed to by the element and its child elements. No supplied parsers support this option.

**options**
The integer value that specifies which bitstream generation mode must be used. Set one of the following values:
- CCI_BITSTREAM_OPTIONS_ROOT
- CCI_BITSTREAM_OPTIONS_EMBEDDED
- CCI_BITSTREAM_OPTIONS_FOLDER

## Return values
- If successful, the correct size of memory that is required to hold the bit stream is returned.
- If the memory allocated by the caller was insufficient, *returnCode* is CCI_BUFFER_TOO_SMALL.
- If an exception occurs during execution, *returnCode* is CCI_EXCEPTION.

## Example

The following example demonstrates how you can use the options parameter to generate the bit stream for different parts of the message tree.

This code can be copied into the _evaluate function of the sample Transform node. For an input message such as:

```
MQMD
MQRFH2
<test><data><aaa>text</aaa></data></test>
```

the node propagates three messages:
- One that contains a copy of the input message in the BLOB domain
- One that contains a copy of the input MQRFH2 as the message body in the BLOB domain
- One that contains the <data></data> folder as the message body in the BLOB domain

```
CciMessage*          outMsg[3];
  CciTerminal*         terminalObject;
  CciElement*          bodyChild;
  CciElement*          inRootElement;
  CciElement*          inSourceElement[3];
  CciElement*          outRootElement;
  CciElement*          outBlobElement;
  CciElement*          outBody;
  struct CciByteArray  bitstream[3];
  int                  bitstreamOptions[3];
  int                  retvalue;
  int                  rc = 0;
  int                  loopCount;
  CCI_EXCEPTION_ST     exception_st = {CCI_EXCEPTION_ST_DEFAULT};
  const CciChar*       constBLOBParserName  =
               cciString("NONE",BIP_DEF_COMP_CCSID);
  const CciChar*       constBLOBElementName =
               cciString("BLOB",BIP_DEF_COMP_CCSID);
  const CciChar*       constEmptyString     =
               cciString("",BIP_DEF_COMP_CCSID);

  /*build up and propagate 3 output messages*/
  /*first message has bit stream for input message body*/
  /*second message has bit stream for input MQRFH2*/
```

```
/*third message has bit stream for sub element from input message*/

/* Get the root element of the input message */
inRootElement = cniRootElement(&rc, message);
/*CCI_CHECK_RC();*/
checkRC(rc);

/*set up the array of source elements and bitstream options*/

/*message body*/
inSourceElement[0] =  cniLastChild(&rc,inRootElement);
checkRC(rc);

/*This is the root of the message body so we use RootBitStream mode*/
bitstreamOptions[0] = CCI_BITSTREAM_OPTIONS_ROOT;


/*last header*/
inSourceElement[1] = cniPreviousSibling(&rc,inSourceElement[0]);
checkRC(rc);

/*This is the root of the MQRFH2 so we use RootBitStream mode*/
bitstreamOptions[1] = CCI_BITSTREAM_OPTIONS_ROOT;


/*body.FIRST(first child of message body) */
inSourceElement[2] = cniFirstChild(&rc,inSourceElement[0]);
checkRC(rc);

/*body.FIRST.FIRST */
inSourceElement[2] = cniFirstChild(&rc,inSourceElement[2]);
checkRC(rc);

/*This is a sub tree within the message body so we use FolderBitStream mode*/
bitstreamOptions[2] = CCI_BITSTREAM_OPTIONS_FOLDER;


for (loopCount=0;loopCount<3;loopCount++) {
  int bufLength;

  /* Create new message for output */
  outMsg[loopCount] = cniCreateMessage(&rc, cniGetMessageContext(&rc, message));
  checkRC(rc);

  /* Get the root element of the output message */
  outRootElement = cniRootElement(&rc, outMsg[loopCount]);
  checkRC(rc);

  /* Copy the contents of the input message to the output message */
  cniCopyElementTree(&rc, inRootElement, outRootElement);
  checkRC(rc);

  /* Get the last child of root (ie the body)  */
  bodyChild = cniLastChild(&rc, outRootElement);
  checkRC(rc);

  /*throw away the message body which was copied from the input message*/
  cniDetach(&rc,
            bodyChild);
  checkRC(rc);

  /*create the new output message body in the BLOB domain*/
  outBody = cniCreateElementAsLastChildUsingParser(&rc,
                                       outRootElement,
                                       constBLOBParserName);
  checkRC(rc);
```

```
   /*create the BLOB element*/
   outBlobElement = cniCreateElementAsLastChild(&rc,
                            outBody);
   checkRC(rc);

   cniSetElementName(&rc,
                  outBlobElement,
                  constBLOBElementName);
   checkRC(rc);

   /*Set the value of the blob element by obtaining the bit stream for the
   element */
   bitstream[loopCount].size=512;
   bitstream[loopCount].pointer=(CciByte*)malloc(sizeof(CciByte) * 512);

   bufLength = cniElementAsBitstream(&rc,
                        inSourceElement[loopCount],
                        &bitstream[loopCount],
                        constEmptyString,/*assume XML message so no interest in*/
                        constEmptyString,/* type, set or format*/
                        constEmptyString,
                        0,/*Use Queue Manager CCSID & Encoding*/
                        0,
                        bitstreamOptions[loopCount]);


   if (rc==CCI_BUFFER_TOO_SMALL)
   {
       free(bitstream[loopCount].pointer);
       bitstream[loopCount].size=bufLength;
       bitstream[loopCount].pointer=(CciByte*)malloc(sizeof(CciByte) * bitstream[loopCount].size);

       bufLength = cniElementAsBitstream(&rc,
                        inSourceElement[loopCount],
                        &bitstream[loopCount],
                        constEmptyString,/*assume XML message so no interest in*/
                        constEmptyString,/* type, set or format*/
                        constEmptyString,
                        0,/*Use Queue Manager CCSID & Encoding*/
                        0,
                        bitstreamOptions[loopCount]);
   }
   checkRC(rc);
   bitstream[loopCount].size=bufLength;

   cniSetElementByteArrayValue(&rc,
                        outBlobElement,
                        &bitstream[loopCount]);
   checkRC(rc);
}

/* Get handle of output terminal */
terminalObject = getOutputTerminalHandle( (NODE_CONTEXT_ST *)context,
                              (CciChar*)constOut);

/* If the terminal exists and is attached, propagate to it */
if (terminalObject) {
  if (cniIsTerminalAttached(&rc, terminalObject)) {
    /* As this is a new, and changed message, it should be finalized... */
    cniFinalize(&rc, outMsg[0], CCI_FINALIZE_NONE);
    cniFinalize(&rc, outMsg[1], CCI_FINALIZE_NONE);
    cniFinalize(&rc, outMsg[2], CCI_FINALIZE_NONE);
    retvalue = cniPropagate(&rc, terminalObject, localEnvironment, exceptionList, outMsg[0]);
    retvalue = cniPropagate(&rc, terminalObject, localEnvironment, exceptionList, outMsg[1]);
    retvalue = cniPropagate(&rc, terminalObject, localEnvironment, exceptionList, outMsg[2]);
    if (retvalue == CCI_FAILURE) {
      if (rc == CCI_EXCEPTION) {
```

```
        /* Get details of the exception */
        memset(&exception_st, 0, sizeof(exception_st));
        cciGetLastExceptionData(&rc, &exception_st);

        /* Any local error handling can go here */

        /* Ensure message is deleted prior to return/throw */
        cniDeleteMessage(0, outMsg[0]);
        cniDeleteMessage(0, outMsg[1]);
        cniDeleteMessage(0, outMsg[2]);

        /* We must "rethrow" the exception; note this does not return */
        cciRethrowLastException(&rc);
      }
      else {

        /* Some other error...the plugin might choose to log it using the CciLog() */
        /* utility function                                                        */

      }
    }
    else {
    }
  }
}
else {
  /* Terminal did not exist...severe internal error. The plugin might want to */
  /* log an error here by using the cciLog() utility function.                */
}

/* Delete the messages we created now we have finished with them */
cniDeleteMessage(0, outMsg[0]);
cniDeleteMessage(0, outMsg[1]);
cniDeleteMessage(0, outMsg[2]);

free((void*) constBLOBParserName);
free((void*) constBLOBElementName);
free((void*) constEmptyString);
return;
```

## cniElementName

Use this function to get the value of the *name* attribute for the specified syntax element. You must sethe syntax element name previously by using cniSetElementName or cpiSetElementName.

### Syntax

```
CciSize cniElementName(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  value,
  Ccisize         length);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN

- CCI_INV_BUFFER_TOO_SMALL

**targetElement**
    The address of the target syntax element object (input).

**value**
    The address of a buffer into which the element name is copied (input).

**length**
    The length, in characters, specified by the *value* parameter (input).

### Return values
- If successful, the element name is copied into the supplied buffer and the number of `CciChar` characters copied is returned.
- If the buffer is not large enough to contain the attribute value, *returnCode* is set to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is returned.
- For any other failures, CCI_FAILURE is returned, and *returnCode* indicates the reason for the error.

## cniElementNamespace

Use this function to get the value of the *namespace* attribute for the specified syntax element.

You must previously have set the syntax element name by using cniSetElementNamespace or cpiSetElementNamespace.

Use this function when you want to convert a message that belongs to a namespace-aware domain to a bit stream.

### Syntax
```
CciSize cniElementNamespace(
 int*          returnCode,
 CciElement*   targetElement,
 const CciChar* value,
 CciSize       length)
```

### Parameters

**returnCode**
    The return code from the function (output). Specifying a NULL pointer signifies that the node does not want to deal with errors. If input is not NULL, the output signifies the success status of the call. All exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. Call cciGetLastExceptionData for details of the exception.

    Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**targetElement**
    Specifies the address of the target syntax element object (input).

**value**

Specifies the address of a buffer into which the element namespace value is
copied (output). A string of characters (including a NULL terminator)
representing the namespace value is copied into this buffer. The buffer must be
a portion of memory previously allocated by the caller.

**length**

The length, in characters, of the buffer specified by the *value* parameter (input).

## Return values

- If successful, the number of CciChars copied into the buffer is returned.
- If the buffer is not large enough to contain the attribute value, *returnCode* is set
  to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is
  returned.
- If an exception occurs during execution, *returnCode* is set to CCI_EXCEPTION.

## Example

```
if (element != 0) {
   /*get name*/
   cniElementName(&rc, element, (CciChar*)&elementName, sizeof(elementName));

   /*get namespace*/
   elementNamespace=(CciChar*)malloc(sizeof(CciChar) * elementNamespaceLength);
   elementNamespaceLength = cniElementNamespace(&rc,
                                                element,
                                                elementNamespace,
                                                elementNamespaceLength);

   if (rc==CCI_BUFFER_TOO_SMALL){
     free(elementNamespace);
     elementNamespace=(CciChar*)malloc(sizeof(CciChar) * elementNamespaceLength);
     elementNamespaceLength = cniElementNamespace(&rc,
                                                  element,
                                                  elementNamespace,
                                                  elementNamespaceLength);

   }
   checkRC(rc);
```

# cniElementType

Use this function to get the value of the *type* attribute for the specified syntax
element. You must set the syntax element type previously by using
cniSetElementType or cpiSetElementType.

## Syntax

```
CciElementType cniElementType(
  int*        returnCode,
  CciElement*  targetElement);
```

## Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**

The address of the target syntax element object (input).

**Return values**

The value of the target element type is returned. If an error occurs, CCI_FAILURE is returned, and the *returnCode* parameter indicates the reason for the error.

# cniElementValue group

Use one or more of the functions in this group to retrieve the value of the specified syntax element.

Specify the appropriate function from this group that matches the type of data to be retrieved:
- cniElementBitArrayValue
- cniElementBooleanValue
- cniElementByteArrayValue
- cniElementCharacterValue
- cniElementDateValue
- cniElementDecimalValue
- cniElementGmtTimestampValue
- cniElementGmtTimeValue
- cniElementIntegerValue
- cniElementRealValue
- cniElementTimestampValue
- cniElementTimeValue

## Syntax

```
CciSize cniElementBitArrayValue(
  int*         returnCode,
  CciElement*  targetElement,
  const struct  CciBitArray* value);

CciBool cniElementBooleanValue(
  int*         returnCode,
  CciElement*  targetElement);

CciSize cniElementByteArrayValue(
  int*         returnCode,
  CciElement*  targetElement,
  const struct  CciByteArray* value);

CciSize cniElementCharacterValue(
  int*         returnCode,
  CciElement*  targetElement,
  const CciChar* value,
  CciSize       length);

struct CciDate cniElementDateValue(
  int*         returnCode,
  CciElement*  targetElement);

CciSize cniElementDecimalValue(
  int*         returnCode,
  CciElement*  targetElement,
  const CciChar* value,
  CciSize       length);

struct CciTimestamp cniElementGmtTimestampValue(
  int*         returnCode,
  CciElement*  targetElement);

struct CciTime cniElementGmtTimeValue(
  int*         returnCode,
  CciElement*  targetElement);
```

```
CciInt cniElementIntegerValue(
  int*        returnCode,
  CciElement*  targetElement);

CciReal cniElementRealValue(
  int*        returnCode,
  CciElement*  targetElement);

struct CciTimestamp cniElementTimestampValue(
  int*        returnCode,
  CciElement*  targetElement);

struct CciTime cniElementTimeValue(
  int*        returnCode,
  CciElement*  targetElement);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**targetElement**
The address of the target syntax element object (input).

**value**
The address of an output buffer into which the value of the syntax element is stored (input). Used on relevant function calls only.

**length**
The length of the output buffer, in characters, specified by the *value* parameter (input). Used on relevant function calls only.

## Return values

- If successful, the value of the target element is returned.
- If the size of an element's data can vary, the correct data size is returned.
- If the specified length is too small, the error code is set to CCI_BUFFER_TOO_SMALL.
- If an error occurs, the *returnCode* parameter indicates the reason for the error.

## Example

```
 numberOfChars = cniElementCharacterValue(
      &rc, firstChild, (CciChar*)&elementValue, sizeof(elementValue)
      );

if (rc==CCI_BUFFER_TOO_SMALL) {
    free(elementValue);
    elementValue      = (CciChar*)malloc(numberOfChars * sizeof(CciChar));
    numberOfChars     = cniElementCharacterValue(
                 &rc, firstChild, (CciChar*)&elementValue, sizeof(elementValue));
  }
```

# cniElementValueState

Use this function to get the state of the value of the specified syntax element.

### Syntax

```
CciValueState cniElementValueState(
  int*        returnCode,
  CciElement*  targetElement);
```

### Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

### Return values

The state of the value of the target syntax element is returned. If an error occurs, CCI_VALUE_STATE_UNDEFINED is returned, and the *returnCode* parameter indicates the reason for the error.

# cniElementValueType

Use this function to get the *type* attribute for the value of the specified syntax element. The state of an element after creation is undefined. When the value of the element is set, its state becomes valid.

### Syntax

```
CciValueType cniElementValueType(
  int*        returnCode,
  CciElement*  targetElement);
```

### Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

### Return values

The type of the value of the target syntax element is returned. If an error occurs, CCI_ELEMENT_TYPE_UNKNOWN is returned, and the *returnCode* parameter indicates the reason for the error.

# cniElementValueValue

Use this function to get the address of the value object owned by the specified syntax element.

## Syntax

```
const CciElementValue* cniElementValueValue(
  int*         returnCode,
  CciElement*  targetElement);
```

## Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

**targetElement**

The address of the target syntax element object (input).

## Return values

The address of the value object of the target syntax element is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniEvaluate

This function performs node processing. The broker calls this function when a message is received on one of the input terminals of an instance of a node object.

The function forms the main logic of the message processing node or output node. It is not used with input nodes. You must define a function table before you call this function.

The responsibilities of the node at this point are to:

1. Process the message in accordance with the values of the attributes on the node instance.
2. Process the message based on content, if desired.
3. Propagate the message to any appropriate output terminals.
4. Throw an exception if an error occurs.

| Defined In | Type | Member |
|---|---|---|
| CNI_VFT | Conditional | iFpEvaluate |

## Syntax

```
void cniEvaluate(
  CciContext  *context,
  CciMessage  *localEnvironment,
  CciMessage  *exceptionList,
  CciMessage  *message);
```

## Parameters

**context**

The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**localEnvironment**
The address of the input local environment object (input).

For compatibility with earlier versions, you can refer to this parameter as destinationList.

**exceptionList**
The address of the exception list for the message (input).

**message**
The address of the input message object (input).

# cniGetComplexAttribute

Returns a pointer to the part of the deployment message representing the user-defined properties of the node.

## Syntax

```
CciElement* cniGetComplexAttribute(
  int*                returnCode,
  CciNode*            nodeObject,
  CciChar*            attributeName);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION

**nodeObject**
The name of the node for which attributes are required (input).

**attributeName**
The name and value of the user-defined attribute for the selected node.

### Return values

Returns a syntax element tree that can be navigated to extract the names and values of all user-defined properties for that node, or NULL if no user-defined properties exist for this node. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniGetOutputTerminal

Returns a pointer to the named output terminal, or NULL if a terminal of this name does not exist.

## Syntax

```
CciTerminal* cniGetOutputTerminal(
  int*                returnCode,
  CciNode*            nodeObject,
  CciChar*            name);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION

**nodeObject**
The name of the node (input).

**name**
The name of the output terminal (input).

### Return values

A pointer to the named output terminal, or NULL if a terminal of this name does not exist. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniGetResourceProperty

Obtains the value of the property propertyName as managed by the resourceManager

### Syntax

```
CciSize cniGetResourceProperty(
  int*            returnCode,
  CciChar*        resourceManager,
  CciChar*        propertyName,
  CciChar*        value,
  CciSize         length);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_BUFFER_TOO_SMALL

**resourceManager**
The address of the resource manager controlling the properties (input).

**propertyName**
The name of the object for which the value is required.

**value**
The value of the buffer, already allocated, that you use.

**length**
The size of the buffer, already allocated, that you use.

If this size is not sufficient, the function fails with *returnCode* set to CCI_BUFFER_TOO_SMALL, and returns the required buffer size; that is, the string length plus one null terminator character.

You should then allocate the required size for the buffer, and call this function again; this function copies the property string into this buffer.

This usage pattern allows you to manage the memory on your system.

This function performs the same task as the mqsireportproperties command.

### Return values

None or CCI_BUFFER_TOO_SMALL. If any other error occurs, the *returnCode* parameter indicates the reason for the error.

## cniFinalize

Use this function to cause the broker to request parsers to perform finalize processing on the specified message. Finalization is a process that fixes header chains and makes the Properties folder match the headers.

The behavior of this processing is specific to each parser. Some parsers do not support finalization processing.

Call cniFinalize before you propagate a message from the node; for example, before you call cniWriteBuffer.

### Syntax

```
void cniFinalize(
  int*        returnCode,
  CciMessage* message,
  int         options);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT

**message**
The address of the message object for which the element tree is to be finalized (input).

**options**
Set this parameter to CCI_FINALIZE_NONE.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example

```
cniFinalize(&rc, outMsg, CCI_FINALIZE_NONE);
retvalue = cniPropagate(
                    &rc,
                    terminalObject,
                    localEnvironment,
                    exceptionList,
                    outMsg);

/* Handle errors */
```

## cniFirstChild

Use this function to retrieve the address of the syntax element object that is the first child of the specified syntax element.

## Syntax

```
CciElement* cniFirstChild(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

## Return values

- If successful, the address of the requested syntax element object is returned.
- If there is no first child, zero is returned, and *returnCode* is set to CCI_SUCCESS.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

## Example

```
if (element != 0) {
  cniElementName(&rc, element, (CciChar*)&elementName, sizeof(elementName));
  firstChild = cniFirstChild(&rc, element);
```

# cniGetAttribute

This function gets the value of an attribute on a specific node instance.

**Restriction:** This function imposes a restriction on the length of the attribute value. This function is provided only for compatibility with earlier versions. You should implement cniGetAttribute2.

This function is invoked by the broker:
- Before the nodes configuration is deployed in order to ascertain default values of any attributes that might override attributes owned by the framework.
- After setting the deployed configuration in order to write the configuration to the broker's database. This call ensures that the configuration persists across shutdown and restarts of the execution group

The responsibilities of the node are to:
1. Return a character representation of the attribute value.
2. Throw an exception if an error occurs.

If both cniGetAttribute and cniGetAttribute2 are implemented, cniDefineNodeClass fails with CCI_INV_IMPL_FUNCTION.

| Defined In | Type | Member |
|---|---|---|
| CNI_VFT | Optional | iFpGetAttribute |

## Syntax

```
int cniGetAttribute(
  CciContext*  context,
  CciChar*     attrName,
  CciChar*     buffer,
  int          bufsize);
```

## Parameters

**context**
>   The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**attrName**
>   The name of the attribute for which the value is to be retrieved (input).

**buffer**
>   The address of a buffer into which the attribute value is copied (output).

**bufsize**
>   The length, in bytes, of the buffer specified in the *buffer* parameter (input).

## Return values

If successful, zero is returned, and the character representation of the value of the attribute is returned in the specified buffer. If the name of the attribute does not identify one supported by the node, a non-zero value is returned.

# cniGetAttribute2

Use this function to get the value of an attribute on a specific node instance. It is called by the broker after all the attributes that the user deploys are set.

The results are written to the broker persistent configuration data to ensure that the node is configured correctly after the execution group process is stopped and started.

The responsibilities of the node are to:

1.  Return a character representation of the attribute value.
2.  Throw an exception if an error occurs.

If both cniGetAttribute and cniGetAttribute2 functions are implemented, cniDefineNodeClass fails with return code CCI_INV_IMPL_FUNCTION.

| Defined In | Type | Member |
|---|---|---|
| CNI_VFT | Optional | iFpGetAttribute2 |

## Syntax

```
CciSize cniGetAttribute2(
  int          returnCode,
  CciContext*  context,
  CciChar*     attrName,
  CciChar*     buffer,
  int          bufsize);
```

## Parameters

**context**
>   The address of the context for the instance of the node, as created by the node and returned by the `cniCreateNodeContext` function (input).

**returnCode (output)**
Pointer to an int. On return, the node must ensure that this int stores a value that describes the status of completion. Possible return codes are:
- CCI_SUCCESS
- CCI_ATTRIBUTE_UNKNOWN
- CCI_BUFFER_TOO_SMALL

**attrName**
The name of the attribute for which the value is to be retrieved (input).

**buffer**
The address of a buffer into which the attribute value is copied (output).

**bufsize**
The length, in `CciChars`, of the buffer specified in the **buffer** parameter (input).

## Return values
- If successful, the attribute value is copied into the supplied buffer and the number of `CciChar` characters copied is returned.
- If the buffer is not large enough to contain the attribute value, *returnCode* is set to CCI_BUFFER_TOO_SMALL, and the number of `CciChars` required is returned.
- If the *attrName* is not known to this node, *returnCode* is set to CCI_ATTRIBUTE_UNKNOWN.

# cniGetAttributeName

This functions returns the name of a node attribute specified by an index.

**Restriction:** This function imposes a restriction on the length of the attribute value. This function is provided only for compatibility with earlier versions. You should implement cniGetAttributeName2.

This function is invoked by the message broker when the broker requires the names of attributes supported by a particular instance of a node. The function must guarantee to return the attributes in a known, defined order, and to return the attribute name represented by the index parameter.

If both cniGetAttributeName and cniGetAttributeName2 are implemented, cniDefineNodeClass fails with CCI_INV_IMPL_FUNCTION.

| Defined In | Type | Member |
|------------|------|--------|
| CNI_VFT | Optional | iFpGetAttributeName |

## Syntax
```
int cniGetAttributeName(
  CciContext*  context,
  int          index,
  CciChar*     buffer,
  int          bufsize);
```

## Parameters

**context**
The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**index**
> Specifies the index of the attribute name (input). The index of the attributes starts from zero.

**buffer**
> The address of a buffer into which the attribute name is copied (output).

**bufsize**
> The length, in bytes, of the buffer specified in the *buffer* parameter (input).

### Return values

If successful, zero is returned, and the name of the attribute is returned in the specified buffer. If the end of the list of attributes is reached, a non-zero value is returned.

## cniGetAttributeName2

This function returns the name of a node attribute specified by an index. It is called by the broker when the broker requires the names of the attributes that are supported by an instance of a node.

The function must return the attributes in a known, defined order, and to return the attribute name that is represented by the index parameter.

If both cniGetAttributeName and cniGetAttributeName2 are implemented, cniDefineNodeClass fails with CCI_INV_IMPL_FUNCTION.

| Defined In | Type | Member |
|------------|------|--------|
| CNI_VFT | Optional | iFpGetAttributeName2 |

### Syntax

```
CciSize cniGetAttributeName2(
  int          returnCode,
  CciContext*  context,
  int          index,
  CciChar*     buffer,
  int          bufsize);
```

### Parameters

**context**
> The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**returnCode (output)**
> Pointer to an int. On return, the node must ensure that this int stores a value that describes the status of completion. Possible return codes are:
> - CCI_SUCCESS
> - CCI_ATTRIBUTE_UNKNOWN
> - CCI_BUFFER_TOO_SMALL

**index**
> Specifies the index of the attribute name (input). The index of the attributes starts from zero.

**buffer**
> The address of a buffer into which the attribute name is copied (output).

**bufsize**

The length, in CciChars, of the buffer specified in the **buffer** parameter (input).

## Return values

- If successful, the attribute name is copied into the supplied buffer and the number of `CciChar` characters copied is returned.
- If the buffer is not large enough to contain the attribute name, *returnCode* is set to CCI_BUFFER_TOO_SMALL, and the number of `CciChars` required is returned.
- If the end of the list of attributes is reached, and the attribute name is not found, *returnCode* is set to CCI_ATTRIBUTE_UNKNOWN. For example, when `index` is greater than n-1, where n is the number of attributes for this node.

# cniGetBrokerInfo

Use this function to query the current broker environment (for example, for information about broker name and message flow name). The information is returned in a structure of type CNI_BROKER_INFO_ST.

## Syntax

```
void cniGetBrokerInfo(
  int*               returnCode,
  CciNode*           nodeObject,
  CNI_BROKER_INFO_ST* broker_info_st);
```

## Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_NODE_OBJECT

**nodeObject**

The message flow processing node for which broker environment information is being requested (input).

**broker_info_st**

The address of a CNI_BROKER_INFO_ST structure that is used to return a message that represents the input destination (input):

```
typedef struct broker_info_st {
int versionId;   /*Structure version identification*/
CCI_STRING_ST brokerName; /*The label of the broker*/
CCI_STRING_ST executionGroupName; /*The label of the current execution group*/
CCI_STRING_ST messageFlowName; /*The label of the current message flow*/
CCI_STRING_ST queueManagerName;  /*The name of the MQ Queue Manager for the broker*/
int commitCount;   /*Commit count value*/
int commitInterval;   /*Commit interval value*/
int coordinatedTransaction;  /*Flag: coordinatedTransaction: 0=no, 1=yes*/
CCI_STRING_ST dataSourceUserId;  /*The userid broker connects to datasource as*/
} CNI_BROKER_INFO_ST;
```

## Return values

None. If an error occurs, the **returnCode** parameter indicates the reason for the error.

### Example

```
cniGetBrokerInfo(&rc, nodeObject, &broker_info_st);
```

where nodeObject is of type CciNode*

# cniGetEnvironmentMessage

Use this function to get the CciMessage object that corresponds to the environment tree for the message flow.

### Syntax

```
CciMessage ImportExportPrefix * ImportExportSuffix
  cniGetEnvironmentMessage(
    int*        returnCode,
    CciMessage*  message);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT

**message**
The address of the message object for which the environment is to be obtained. This message might be an input message that has been received as an argument to the cniEvaluate implementation function, or a message created by the cniCreateMessage utility function.

### Return values

If successful, the address of the message object corresponding to the environment tree is returned. Otherwise, a value of zero is returned, and the *returnCode* parameter indicates the reason for the error.

# cniGetMessageContext

Use this function to get the address of the message context that is associated with the specified message. The context of an existing message is used to create an output message; for example, by using the cniCreateMessage function.

### Syntax

```
CciMessageContext* cniGetMessageContext(
  int*        returnCode,
  CciMessage*  message);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_INV_MESSAGE_OBJECT

**message**
The address of the message object (input).

### Return values

If successful, the address of the message context is returned. Otherwise, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example

```
outMsg = cniCreateMessage(&rc, cniGetMessageContext(&rc, message));
```

# cniGetParserClassName

Use this function to get the parser class name that is associated with the specified syntax element.

### Syntax

```
CciSize cniGetParserClassName(
  int*          returnCode,
  CciElement*   targetElement,
  const CciChar* value,
  CciSize        length);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**targetElement**

The address of the element for which the parser class name is to be returned (input).

**value**

The address of an output buffer into which the parser class name is stored (input).

**length**

The length of the output buffer, expressed as the number of CciChar characters, specified in the *value* parameter (input).

### Return values

- If successful, the *returnCode* parameter indicates CCI_SUCCESS, and the number of characters written to the buffer is returned.
- If the buffer is not large enough to retain the returned name, the *returnCode* parameter indicates CCI_BUFFER_TOO_SMALL, and the returned value indicates the number of characters required to store the name.
- If other errors occur, CCI_FAILURE is returned, and the *returnCode* parameter indicates the reason for the error.

# cniGetThreadContext

This function returns the thread context for the current thread.

### Syntax

```
CciThreadContext *cniGetThreadContext(
  int                *returnCode,
  CciMessageContext  *msgContext);
```

### Parameters

**returnCode**

This parameter is the return code from the function (output). If the input value is NULL, this value signifies that errors are silently handled, or are ignored by the broker. If the input value is not NULL, the output signifies the success status of the call. If the `msgContext` parameter is not valid, `*returnCode` is set to CCI_INV_MESSAGE_CONTEXT, and a NULL CciThreadContext is returned.

**msgContext**

This parameter provides the message context from which to acquire the thread-specific context. It is expected that this parameter is obtained by using the cniGetMessageContext utility function.

### Return values

If this function is successful, it returns a handle to the CciThreadContext for the current thread.

The cciMessageContext value must correspond to a cciMessage, where the cciMessage is passed in to the cniEvaluate or cniRun function on the current thread.

### Example

```
CciMessageContext* messageContext = cniGetMessageContext(NULL,message);
CciThreadContext*  threadContext  = cniGetThreadContext(NULL,messageContext);
```

# cniIsTerminalAttached

Use this function to check whether a terminal is attached to another node by a connector. It returns an integer value that specifies whether the specified terminal object is attached to one or more terminals on other message flow nodes.

Use this function to test whether a message can be propagated to a terminal; you do not have to call this function before you propagate a message with the cniPropagate utility function. Use the cniIsTerminalAttached function to modify the node behavior when a terminal is not connected.

### Syntax

```
int cniIsTerminalAttached(
  int*          returnCode,
  CciTerminal*  terminalObject);
```

### Parameters

**returnCode**

The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_TERMINAL_OBJECT

**terminalObject**
> The address of the input or output terminal to be checked for an attached connector (input). The address is returned from cniCreateOutputTerminal.

### Return values
- If the terminal is attached to another node by a connector, a value of 1 is returned.
- If the terminal is not attached, or a failure occurred, a value of zero is returned.
- If a failure occurs, the value of the *returnCode* parameter indicates the reason for the error.

### Example
```
if (terminalObject) {
  if (cniIsTerminalAttached(&rc, terminalObject)) {
    if (rc == CCI_SUCCESS) {
      retvalue = cniPropagate(
                           &rc,
                           terminalObject,
                           localEnvironment,
                           exceptionList,
                           message);
```

## cniLastChild

Use this function to retrieve the address of the syntax element object that is the last child of the specified syntax element.

### Syntax
```
CciElement* cniLastChild(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_MESSAGE_OBJECT

**targetElement**
> The address of the target syntax element object (input).

### Return values
- If successful, the address of the requested syntax element object is returned.
- If there is no last child, zero is returned, and *returnCode* is set to CCI_SUCCESS.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example
```
bodyChild = cniLastChild(&rc, outRootElement);
```

## cniNextSibling

Use this function to retrieve the address of the syntax element object that is the next sibling (right sibling) of the specified syntax element.

### Syntax

```
CciElement* cniNextSibling(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

### Return values

- If successful, the address of the requested syntax element object is returned.
- If there is no next sibling, zero is returned, and *returnCode* is set to CCI_SUCCESS.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example

```
nextSibling = cniNextSibling(&rc, element);
```

## cniParent

Use this function to retrieve the address of the syntax element object that is the parent of the specified syntax element.

### Syntax

```
CciElement* cniParent(
  int*        returnCode,
  CciElement* targetElement);
```

### Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

### Return values

- If successful, the address of the requested syntax element is returned.
- If there is no parent element, zero is returned.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniPreviousSibling

Use this function to retrieve the address of the syntax element object that is the previous sibling (left sibling) of the specified syntax element.

## Syntax

```
CciElement* cniPreviousSibling(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

## Return values

- If successful, the address of the requested syntax element object is returned.
- If there is no previous sibling, zero is returned, and *returnCode* is set to CCI_SUCCESS.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniPropagate

Use this function to propagate a message to a specified terminal object. If the terminal is not attached to another node by a connector, the message is not propagated, and the function is ignored.

Therefore, you do not have to check whether the terminal is attached before you propagate the message, unless you want the node to take different in this scenario. If required, you can use cniIsTerminalAttached to check whether the terminal is connected before you call this function.

## Syntax

```
int cniPropagate(
  int*         returnCode,
  CciTerminal* terminalObject,
  CciMessage*  localEnvironment,
  CciMessage*  exceptionList,
  CciMessage*  message);
```

## Parameters

**returnCode**
> The return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_TERMINAL_OBJECT
> - CCI_INV_MESSAGE_OBJECT

**terminalObject**
> The address of the output terminal to receive the message (input). The address is returned by cniCreateOutputTerminal.

**localEnvironment**
> The address of the local environment object to be sent with the message (input).
>
> This message object is used by the publish/subscribe node supplied by the broker.
>
> For compatibility with earlier versions, you can refer to this parameter as `destinationList`.

**exceptionList**
> The address of the exception list for the message (input).

**message**
> The address of the message object to be sent (input). If the message being sent is the same as the input message, this address is the one passed on the cniEvaluate implementation function.

## Return values

If successful, CCI_SUCCESS is returned. Otherwise, CCI_FAILURE is returned, and the *returnCode* parameter indicates the reason for the error.

## Example

```
if (terminalObject) {
  if (cniIsTerminalAttached(&rc, terminalObject)) {
    if (rc == CCI_SUCCESS) {
      cniPropagate(&rc, terminalObject, destinationList, exceptionList, message);
```

# cniRootElement

Use this function to get the root syntax element associated with a specified message. It returns the root element that is associated with (and owned by) the message object identified by the *message* parameter. When a message object is constructed by the broker, a root element is automatically created.

## Syntax

```
CciElement* cniRootElement(
  int*        returnCode,
  CciMessage* message);
```

## Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> * CCI_SUCCESS
> * CCI_EXCEPTION
> * CCI_INV_MESSAGE_OBJECT

**message**
> The address of the message object (input).

### Return values

If successful, the address of the root element object is returned. Otherwise, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

### Example

```
inRootElement = cniRootElement(&rc, message);
```

# cniRun

This function declares the node as an input node.

Message processing nodes and output nodes do not use it, and you do not need to call cniEvaluate. The broker allocates a thread, and calls this function on that thread.

| Defined In | Type | Member |
|------------|------|--------|
| CNI_VFT | Conditional | iFpRun |

### Syntax

```
int cniRun(
  CCiContext*  context,
  CCiMessage*  localEnvironment,
  CciMessage*  exceptionList,
  CciMessage*  message
);
```

### Parameters

**context**
The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**localEnvironment**
The address of the input local environment object (input).

For compatibility with earlier versions, you can refer to this parameter as `destinationList`.

**exceptionList**
The address of the exception list for the message (input).

**message**
The address of the message object to which the data is attached (input).

The user-defined node can call cniSetInputBuffer to associate a bit stream with this message. Populating the tree of this message is not supported, therefore calls to functions such as cniAddAsLastChild or cniCreateElementAsLastChildFromBitstream do not work. To build parts of the tree, create a new message using cniCreateMessage rather than providing a buffer to be parsed as the whole message.

For example, if you have a bit stream that is to be used as the payload part of the message, and you also want to add a header, take the following steps:
1. Create a new message using cniCreateMessage.
2. Create the header part in this new message by using the Syntax Element Access Utility functions, for example cniCreateElementAsLastChildUsingParser, and passing in the root element of this new message.

3. Add fields to the header by using functions such as cniCreateElementAsLastChild.

4. Create the body of the message by parsing your bit stream through calling cniCreateElementAsLastChildFromBitstream, and passing in the root element of this new message.

### Return values

This function is called by the broker as part of a loop. The meaning of the return value is as follows:

**CCI_TIMEOUT**
The input node did not receive its input data. This value means that control should be returned to the broker in case message flow reconfiguration is being requested. A user-defined input node should return regularly to give control back to the broker.

**CCI_SUCCESS_CONTINUE**
A message was successfully processed. The broker performs default transaction commit processing. The input node's cniRun implementation function is called immediately so that the node can continue processing.

**CCI_SUCCESS_RETURN**
A message has been successfully processed. The broker performs default transaction commit processing. The input node has determined that the thread is not required, and it is returned to the message flow thread pool. If this processing is performed on the only thread, or the last active thread, the broker prevents this last thread being returned to the pool, otherwise no active threads are available to dispatch another thread. In this situation, the broker invokes the cniRun implementation function immediately, as if CCI_SUCCESS_CONTINUE was returned.

**CCI_FAILURE_CONTINUE**
An error was detected in the processing of a message, and the node is requesting that transaction rollback processing is performed. The input node's cniRun implementation function is called immediately.

**CCI_FAILURE_RETURN**
An error was detected in the processing of a message, and the node is requesting that transaction rollback processing is performed. However, the input node has determined that the thread is not required and it can be returned to the message flow thread pool. If this processing is performed on the last active thread, the broker prevents this last thread being returned to the pool, otherwise no active threads are available to dispatch another thread. In this situation the broker invokes the cniRun implementation function immediately, as if CCI_FAILURE_CONTINUE was returned.

## cniSearchElement group

Use one or more of the functions in this group to search previous siblings of the specified element for an element that matches the specified criteria.

The search is performed starting at the syntax element specified in the *targetElement* parameter, and each of the four functions provides a search in a different tree direction:

- cniSearchFirstChild searches the immediate child elements of the starting element from the first child, until either a match is found, or the end of the child element chain is reached.

- cniSearchLastChild searches the immediate child elements of the starting element from the last child, until either a match is found, or the end of the child element chain is reached.
- cniSearchNextSibling searches from the starting element to the next siblings, until either a match is found, or the end of the sibling chain is reached.
- cniSearchPreviousSibling searches from the starting element to the previous siblings, until either a match is found, or the start of the sibling chain is reached.

If you use this command to search for an element within a message that belongs to a namespace-aware domain, the search is performed only on those elements whose namespace is an empty string. If you want to perform a search for elements in all namespaces, use one of the cniSearchElementNamespace functions.

## Syntax

```
CciElement* cniSearchFirstChild(
  int*            returnCode,
  CciElement*     targetElement,
  CciCompareMode* mode,
  CciElementType  type,
  CciChar         name);

CciElement* cniSearchLastChild(
  int*            returnCode,
  CciElement*     targetElement,
  CciCompareMode* mode,
  CciElementType  type,
  CciChar         name);

CciElement* cniSearchNextSibling(
  int*            returnCode,
  CciElement*     targetElement,
  CciCompareMode* mode,
  CciElementType  type,
  CciChar         name);

CciElement* cniSearchPreviousSibling(
  int*            returnCode,
  CciElement*     targetElement,
  CciCompareMode* mode,
  CciElementType  type,
  CciChar         name);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
The address of the syntax element object from which the search starts (input).

**mode**
The search mode to use (input). This parameter indicates what combination of element type and element name is to be searched for. The possible values are:
- CCI_COMPARE_MODE_FULL
- CCI_COMPARE_MODE_FULL_TYPE
- CCI_COMPARE_MODE_GENERIC_TYPE
- CCI_COMPARE_MODE_SPECIFIC_TYPE

- CCI_COMPARE_MODE_NAME
- CCI_COMPARE_MODE_NAME_SPECIFIC_TYPE
- CCI_COMPARE_MODE_NAME_GENERIC_TYPE
- CCI_COMPARE_MODE_NAME_FULL_TYPE
- CCI_COMPARE_MODE_NULL

**type**
> The element type to search for (input). Use this parameter only if the search mode involves a match on the type.

**name**
> The element name to search for (input). Use this parameter only if the search mode involves a match on the name.

## Example

```
int rc;
CciElement* firstChild = cniSearchFirstChild(
                                        &rc,
                                         inRootElement,
                                         CCI_COMPARE_MODE_NAME,
                                         elementName,
                                         0);
```

## Return values

- If successful, the address of the requested syntax element object is returned.
- If there is no matching element, zero is returned.
- If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniSearchElementInNamespace group

Use this element to search for an element that matches the specified criteria.

The search starts at the syntax element specified in the element argument, and each of the four functions provides a search in a different tree direction:

1. cniSearchFirstChildInNamespace searches the immediate child elements of the starting element from the first child, until either a match is found, or the end of the child element chain is reached.
2. cniSearchLastChildInNamespace searches the immediate child elements of the starting element from the last child, until either a match is found, or the end of the child element chain is reached.
3. cniSearchNextSiblingInNamespace searches from the starting element to the next siblings, until either a match is found, or the end of the sibling chain is reached.
4. cniSearchPreviousSiblingInNamespace searches from the starting element to the previous siblings, until either a match is found, or the start of the sibling chain is reached.

Use this function when you search a message that belongs to a namespace-aware domain.

## Syntax

```
void cniSearchFirstChildInNamespace(
 int*           returnCode,
 CciElement*    targetElement,
 CciCompareMode mode,
 const CciChar* nameSpace,
```

```
 const CciChar* name,
 CciElementType type)

void cniSearchLastChildInNamespace(
 int*          returnCode,
 CciElement*   targetElement,
 CciCompareMode mode,
 const CciChar* nameSpace,
 const CciChar* name,
 CciElementType type)

void cniSearchNextSiblingInNamespace(
 int*          returnCode,
 CciElement*   targetElement,
 CciCompareMode mode,
 const CciChar* nameSpace,
 const CciChar* name,
 CciElementType type)

void cniSearchPreviousSiblingInNamespace(
 int*          returnCode,
 CciElement*   targetElement,
 CciCompareMode mode,
 CciElementType type,
 const CciChar* nameSpace,
 const CciChar* name)
```

## Parameters

**returnCode**
> The return code from the function (output). Specifying a NULL pointer
> signifies that the node does not want to deal with errors. If input is not NULL,
> the output signifies the success status of the call. All exceptions thrown during
> the execution of this call are re-thrown to the next upstream node in the flow.
> Call cciGetLastExceptionData for details of the exception.
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the syntax element object from which the search starts (input).

**mode**
> The search mode to use (input). This parameter indicates what combination of
> element namespace, element name, and element type is to be searched for. The
> possible values are:
> - CCI_COMPARE_MODE_SPACE
> - CCI_COMPARE_MODE_SPACE_FULL_TYPE
> - CCI_COMPARE_MODE_SPACE_GENERIC_TYPE
> - CCI_COMPARE_MODE_SPACE_SPECIFIC_TYPE
> - CCI_COMPARE_MODE_SPACE_NAME
> - CCI_COMPARE_MODE_SPACE_NAME_FULL_TYPE
> - CCI_COMPARE_MODE_SPACE_NAME_GENERIC_TYPE
> - CCI_COMPARE_MODE_SPACE_NAME_SPECIFIC_TYPE
> - CCI_COMPARE_MODE_NULL
>
> When the compare mode does not involve a match on the namespace, all
> namespaces are searched. This behavior differs from that of the

cniSearchElement group, where only the empty string namespace is searched. When you specify one of the valid modes, set the *nameSpace* parameter to the empty string.

**type**

The element type to search for (input). Use this parameter only if the search mode involves a match on the type.

**nameSpace**

The namespace to search (input). Use this parameter only if the search mode involves a match on the namespace.

**name**

The name to search for (input). Use this parameter only if the search mode involves a match on the name.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example

```
mode=CCI_COMPARE_MODE_SPACE ;
...

  if (forward) {
    firstChild = cniSearchFirstChildInNamespace(&rc, element, mode, space, 0,0);
  }else{
    firstChild = cniSearchLastChildInNamespace(&rc, element, mode, space, 0,0);

  }

  if (firstChild) {
    depth++;
    traceElement(firstChild,forward,space);
    depth--;
  }
  currentElement = firstChild;
  do{

    if (forward) {
      nextSibling = cniSearchNextSiblingInNamespace(&rc, currentElement,mode,space,0,0);
    }else{
      nextSibling = cniSearchPreviousSiblingInNamespace(&rc, currentElement,mode,space,0,0);
    }
    if (nextSibling) {
      traceElement(nextSibling,forward,space);
      currentElement=nextSibling;
    }

  }while (nextSibling) ;

}
```

## cniSetAttribute

This function sets the value of an attribute on a specific node instance. It is called by the broker when a configuration request is received that attempts to set the value of a node attribute, or during initialization of the node.

A node receives requests to set attributes for the base. If an unknown attribute value is received, this function *must* return a non-zero value so that the broker processes the request correctly.

The responsibilities of the node are to:

1. Verify that the value of the attribute is correctly specified. If not, a configuration exception should be thrown using the cciThrowException function.

2. Store the value of the attribute within the context, which should have been allocated in the cniCreateNodeContext function.

3. Throw a configuration exception if an error occurs, by using the cciThrowException function.

| Defined In | Type | Member |
|------------|------|--------|
| CNI_VFT | Optional | iFpSetAttribute |

### Syntax

```
int cniSetAttribute(
  CciContext*  context,
  CciChar*     attrName,
  CciChar*     attrValue);
```

### Parameters

**context**
 The address of the context for the instance of the node, as created by the node and returned by the cniCreateNodeContext function (input).

**attrName**
 The name of the attribute whose value is to be set (input).

**attrValue**
 The value of the attribute (input).

### Return values

If successful, zero is returned. If the name of the attribute does not identify one supported by the node, a non-zero value is returned.

## cniSetElementName

Use this function to set the name of the specified syntax element.

### Syntax

```
void cniSetElementName(
  int*          returnCode,
  CciElement*   targetElement,
  const CciChar* name);
```

### Parameters

**returnCode**
 The return code from the function (output).

 Possible return codes are:
 - CCI_SUCCESS
 - CCI_EXCEPTION
 - CCI_INV_ELEMENT_OBJECT
 - CCI_INV_DATA_POINTER

**targetElement**
 The address of the target syntax element object (input).

**name**
 The name of the element (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example

```
CciElement* lastChild = cniCreateElementAsLastChild(&rc, outRootElement);
cniSetElementName(&rc, lastChild, elementName);
cniSetElementType(&rc, lastChild, CCI_ELEMENT_TYPE_NAME);
```

## cniSetElementNamespace

Use this function to set the *namespace* attribute for the specified syntax element when you manipulate a message that belongs to a namespace-aware domain.

### Syntax

```
void cniSetElementNamespace(
 int*          returnCode,
 CciElement*   targetElement,
 const CciChar* nameSpace)
```

### Parameters

**returnCode**
 The return code from the function (output). Specifying a NULL pointer signifies that the node does not want to deal with errors. If input is not NULL, the output signifies the success status of the call. Any exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. Call cciGetLastExceptionData for details of the exception.

 Possible return codes are:
 - CCI_SUCCESS
 - CCI_EXCEPTION
 - CCI_INV_ELEMENT_OBJECT
 - CCI_INV_DATA_POINTER

**targetElement**
 Specifies the address of the target syntax element object (input).

**value**
 Specifies the address of a null terminated string of CciChars representing the namespace value (output). An empty string is a valid value for namespace. By default, elements are created in the empty string namespace, so you could specify an empty string as the namespace, but it has an effect only if the element was previously in another namespace and you want to change the namespace value to empty string.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniSetElementType

Use this function to set the type of the specified syntax element.

## Syntax

```
void cniSetElementType(
  int*          returnCode,
  CciElement*   targetElement,
  CciElementType  type);
```

## Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT

**targetElement**
> The address of the target syntax element object (input).

**type**
> The type of the element (input).

## Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## Example

```
CciElement* lastChild = cniCreateElementAsLastChild(&rc, outRootElement);
cniSetElementName(&rc, lastChild, elementName);
cniSetElementType(&rc, lastChild, CCI_ELEMENT_TYPE_NAME);
```

# cniSetElementValue group

Use one or more of the functions in this group to set the value of the specified syntax element.

Specify the appropriate function from this group that matches the type of data to be retrieved:
- cniSetElementBitArrayValue
- cniSetElementBooleanValue
- cniSetElementByteArrayValue
- cniSetElementCharacterValue
- cniSetElementDateValue
- cniSetElementDecimalValue
- cniSetElementGmtTimestampValue
- cniSetElementGmtTimeValue
- cniSetElementIntegerValue
- cniSetElementRealValue
- cniSetElementTimestampValue
- cniSetElementTimeValue

## Syntax

```
void cniSetElementBitArrayValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciBitArray* value);
```

```
void cniSetElementBooleanValue(
  int*                    returnCode,
  CciElement*             targetElement,
  CciBool                 value);
void cniSetElementByteArrayValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciByteArray* value);
void cniSetElementCharacterValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const CciChar*          value,
  CciSize                 length);
void cniSetElementDateValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciDate*   value);
void cniSetElementDecimalValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const CciChar*          value);
void cniSetElementGmtTimestampValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTimestamp* value);
void cniSetElementGmtTimeValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTime*   value);
void cniSetElementIntegerValue(
  int*                    returnCode,
  CciElement*             targetElement,
  CciInt                  value);
void cniSetElementRealValue(
  int*                    returnCode,
  CciElement*             targetElement,
  CciReal                 value);
void cniSetElementTimestampValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTimestamp* value);
void cniSetElementTimeValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTime*   value);
```

## Parameters

**returnCode**
> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT
> - CCI_INV_DATA_POINTER
> - CCI_INV_DATA_BUFLEN

**targetElement**
> The address of the target syntax element object (input).

**value**
    The value to store in the syntax element (input).

**length**
    The length of the data value (input). Used on relevant function calls only.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example

```
static char* functionName = (char *)"_Input_run()";
void* buffer;
CciTerminal* terminalObject;
int buflen = 4096;
int rc = CCI_SUCCESS;
int rcDispatch = CCI_SUCCESS;
char xmlData[] = "<A>data</a>";
buffer = malloc(buflen);
memcpy(buffer, &xmlData, sizeof(xmlData));

cniSetInputBuffer(&rc, message, buffer, buflen);
```

# cniSetElementValueValue

Use this function to set the value object of the specified syntax element.

### Syntax

```
void cniSetElementValueValue(
  int*             returnCode,
  CciElement*      targetElement,
  CciElementValue* value);
```

### Parameters

**returnCode**
    The return code from the function (output).

    Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

**targetElement**
    The address of the target syntax element object (input).

**value**
    The address of a value object that is used to set the value of the syntax element specified by the *targetElement* parameter (input). You can get the address of the value object by calling cniElementValueValue.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

# cniSetInputBuffer

Use this function to supply a buffer. It is used only by input nodes. The address is specified by the source parameter as an input bit stream of the input message to the broker.

By supplying a buffer, an input node can read data into the bit stream that represents an input message from an external data source. The broker takes a copy of the data and the caller can free the storage on return.

## Syntax

```
int cniSetInputBuffer(
  void*       returnCode,
  CciMessage* message,
  Void*       source,
  CCiInt      length);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN

**message**
The message object that uses the buffer described by the *source* parameter to represent the input bit stream. (input)

**source**
The address of the buffer to be used as input. (input)

**length**
The length of the input buffer described by the *source* parameter. (input)

## Return values

None. If an error occurs, the*returnCode* parameter indicates the reason for the error.

## Example

```
cniSetInputBuffer(&rc, message, buffer, buflen);
```

# cniSqlCreateModifyablePathExpression

Use this function to create an SqlPathExpression object that represents the path that is specified by the path argument. When they are navigated, path elements are created if they do not already exist. This function returns a pointer to the PathExpression object, which is used as input to the functions that navigate the path, namely the cniSqlNavigatePath family.

Because an overhead is incurred in creating the expression, if the same path expression is to be used for every message, call this function once, and use the CciSqlPathExpression* that is returned in a call to cniSqlNavigate for each message. You can use the CciSqlPathExpression on threads other than the one on which it was created.

## Syntax

```
CciSqlPathExpression* cniSqlCreateModifiablePathExpression(
 int* returnCode,
 CciNode* nodeObject,
 CciChar* dataSourceName,
 CciChar* path);
```

## Parameters

**returnCode (output)**
A NULL pointer input signifies that the user-defined node does not handle errors. Any exceptions that are thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. Call cciGetLastExceptionData to retrieve details of the exception. If an invalid nodeObject parameter was passed in, *returnCode* is set to CCI_INV_NODE_OBJECT. If an invalid path parameter, such as NULL or an empty string, was passed in, *returnCode* is set to CCI_INV_ESQL_PATH_EXPR.

**nodeObject (input)**
Specifies the message flow processing node that the ESQL Path Expression is owned by. This pointer is passed to the cniCreateNodeContext implementation function. This parameter must not be NULL.

**dataSourceName (input)**
The ODBC data source name to be used if the statement references an external database. This parameter can be NULL.

**path (input)**
Pointer to a NULL terminated string of CciChars. This parameter specifies the ESQL path expression to be created as defined by the ESQL field reference syntax diagram, except that it cannot include local ESQL variables, ESQL reference variables, user-defined functions, or ESQL namespace constants, because they cannot be declared. This parameter must not be NULL.

## Return values

If successful, the address of the SQLPathExpression object is returned. If an error occurs, CCI_NULL_ADDR is returned, and the return code parameter indicates the reason for the error. When the SQLPathExpression is no longer needed, (typically when the node is deleted) call cniSqlDeletePathExpression to delete it.

## Example

If you add the following code to the Transform node sample, you can create an element, and all necessary ancestor elements, with one function call.

Create the CciSQLPathExpression in the _Transform_createNodeContext function:

```
{
      CciChar ucsPathExpressionString[32];
      char*  mbPathExpressionString =
                    "OutputRoot.XMLNS.Request.A.B.C.D.E";
      /* convert our path string to unicode*/
      cciMbsToUcs(NULL,
          mbPathExpressionString,
          ucsPathExpressionString,
          32,
          BIP_DEF_COMP_CCSID);

      p->pathExpression =
```

```
                         cniSqlCreateModifiablePathExpression(
                                 NULL,
                                 nodeObject,
                                 NULL,/* do not reference Database*/
                                 ucsPathExpressionString);
     }
```

Now use the CciSqlPathExpression later in the _Transform_evaluate function

```
{
     CciElement* newElement =
              cniSqlNavigatePath(
                  NULL,
                  ((NODE_CONTEXT_ST *)context)->pathExpression,
                  message,
                  localEnvironment,
                  exceptionList,
                  outMsg,
                  NULL,/* do not reference OutputLocalEnvironment*/
                  NULL/* do not reference OutputLExceptionList*/);
}
```

Therefore passing in the input message PluginSample.change.xml:

```
<Request
type="change">
  <CustomerAccount>01234567</CustomerAccount>
  <CustomerPhone>555-0000</CustomerPhone>
</Request>
```

The following output message is generated:

```
<Request
type="modify">
  <CustomerAccount>01234567</CustomerAccount>
  <CustomerPhone>555-0000</CustomerPhone>
  <A>
    <B>
      <C>
         <D/>
      </C>
    </B>
  </A>
</Request>
```

This approach, rather than using functions such as cniCreateElementAsLastChild, has the following advantages:

- The path is more dynamic: the path string could be determined at deploy time, for example based on a node property (you could create the CciSQLPathExpression in the cniSetAttribute implementation function).
- While navigating to and creating the element, only one function call is made. This technique is more apparent when the target element is deep within the tree structure.

## cniSqlCreateReadOnlyPathExpression

Use this function to create an SqlPathExpression object that represents the path that is specified by the path argument. The navigated path does not create path elements if they do not already exist. This function returns a pointer to the PathExpression object, which is used as input to the functions that navigate the path, namely the cniSqlNavigatePath family.

Because an overhead is incurred in creating the expression, if the same path expression is to be used for every message, call this function once, and use the

CciSqlPathExpression* that is returned in a call to cniSqlNavigate for each message. You can use the CciSqlPathExpression* on threads other than the one on which it was created.

## Syntax

```
CciSqlPathExpression* cniSqlCreateReadOnlyPathExpression(
    int*      returnCode,
    CciNode* nodeObject,
    CciChar* dataSourceName,
    CciChar* path );
```

## Parameters

**returnCode (output)**

A NULL pointer input signifies that the user-defined node does not handle errors. Any exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. A call to cciGetLastExceptionData provides details of the exception. If an invalid nodeObject parameter was passed in, returnCode is set to CCI_INV_NODE_OBJECT. If an invalid path parameter, such as a NULL or empty string, was passed in, returnCode is set to CCI_INV_ESQL_PATH_EXPR.

**nodeObject (input)**

Specifies the message flow processing node that owns the ESQL Path Expression. This pointer is passed to the cniCreateNodeContext implementation function. This parameter must not be NULL.

**dataSourceName (input)**

The ODBC data source name that is used if the statement references an external database. NULL is allowed.

**path (input)**

Pointer to a NULL terminated string of CciChars. This parameter specifies the ESQL path expression to be created, as defined by the ESQL field reference syntax diagram. It cannot include local ESQL variables, ESQL reference variables, user-defined functions, or ESQL namespace constants, because they cannot be declared. This parameter must not be NULL.

## Return values

If successful, the address of the SQLPathExpression object is returned. If an error occurs, CCI_NULL_ADDR is returned and the return code parameter indicates the reason for the error. When the SQLPathExpression is no longer needed (typically when the node is deleted), call cniSqlDeletePathExpression to delete it.

## Example

The switch node sample shows how to navigate to a syntax element using functions like cniFirstChild. The following code could be used to achieve the same result.

In _Switch_createNodeContext function, create the CciSqlPathExpression for use later.

```
{
    CciChar ucsPathExpressionString[32];
    char*   mbPathExpressionString = "InputBody.Request.type";
    /* convert our path string to unicode*/
```

```
            cciMbsToUcs(
                    NULL,
                    mbPathExpressionString,
                    ucsPathExpressionString,
                    32,
                    BIP_DEF_COMP_CCSID);

            p->pathExpression =
                    cniSqlCreateReadOnlyPathExpression(
                            NULL,
                            nodeObject,
                            NULL, /* do not reference Database*/
                            ucsPathExpressionString);
}
```

This code assumes the addition of the field CciSqlPathExpression* pathExpression
to the NODE_CONTEXT_ST struct.

Now use the CciSqlPathExpression in the _Switch_evaluate function.

```
CciElement* targetElement = cniSqlNavigatePath(
                            NULL,
                            ((NODE_CONTEXT_ST *)context)->pathExpression,
                            message,
                            localEnvironment,
                            exceptionList,
                            NULL, /* do not reference any output trees*/
                            NULL,
                            NULL);
```

This approach, rather than using functions such as cniFirstChild and
cniNextSibling, has the following advantages:

- The path is more dynamic: the path string could be determined at deploy time
  based on a node property (you could create the CciSqlPathExpression in the
  cniSetAttribute implementation function).
- While navigating to the element, only one function call is made. This technique
  is more apparent when the target element is deep within the tree structure.

## cniSqlCreateStatement

Use this function to create an ESQL expression object that represents the statement
specified by the statement argument, by using the syntax defined for the Compute
node.

You cannot use the following statements:
- CREATE PROCEDURE
- CREATE MODULE
- CREATE SCHEMA
- CREATE FUNCTION

This function returns a pointer to the SQL expression object, which is used as input
to the functions that execute the statement, which are cniSqlExecute and
cniSqlSelect. You can create multiple SQL expression objects in a single message
flow node. Although you can create these objects at any time, you would typically
create them when the message flow node is instantiated, within the
implementation function cniCreateNodeContext.

## Syntax

```
CciSqlExpression* cniSqlCreateStatement(
  int*              returnCode,
  CciNode*          nodeObject,
  CciChar*          dataSourceName,
  CciSqlTransaction transaction,
  CciChar*          statement);
```

## Parameters

**returnCode**
:   The return code from the function (output).

    Possible return codes are:
    - CCI_SUCCESS
    - CCI_EXCEPTION
    - CCI_INV_NODE_OBJECT
    - CCI_INV_TRANSACTION_TYPE
    - CCI_INV_STATEMENT

**nodeObject**
:   The message flow processing node that the SQL expression object is owned by (input). This pointer is passed to the cniCreateNodeContext implementation function.

**dataSourceName**
:   The ODBC data source name used if the statement references data in an external database (input).

**transaction**
:   Specifies whether a database commit is performed after the statement is executed (input). Valid values are:

    **CCI_SQL_TRANSACTION_AUTO**
    :   Specifies that a database commit is performed at the completion of the message flow (that is, as a fully globally coordinated or partially globally coordinated transaction). This value is the default.

    **CCI_SQL_TRANSACTION_COMMIT**
    :   Specifies that a commit is performed after execution of the statement, and in the cniSqlExecute or cniSqlSelect function (that is, the message flow is partially broker coordinated).

**statement**
:   The SQL expression to be created, using the syntax as defined for the Compute node (input).

## Return values

If successful, the address of the SQL expression object is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

# cniSqlDeletePathExpression

Use this function to delete the SQLPathExpression that you have created by calling the cniSqlCreateReadOnlyPathExpression or the cniSqlCreateModifiablePathExpression utility functions, as defined by the *sqlPathExpression* argument.

## Syntax

```
void cniSqlDeletePathExpression(
        int*                returnCode,
        CciSqlPathExpression* sqlPathExpression );
```

## Parameters

**returnCode (output)**
A NULL pointer input signifies that the user-defined node does not want to
deal with errors. All exceptions thrown during the execution of this call are
re-thrown to the next upstream node in the flow. If input is not NULL, output
signifies the success status of the call. If an exception occurs during execution,
*returnCode is set to CCI_EXCEPTION on output. Call
cciGetLastExceptionData to get details of the exception. If an invalid
*sqlPathExpression* parameter is passed in, *returnCode* is set to
CCI_INV_SQL_EXPR_OBJECT.

**sqlPathExpression (output)**
Specifies the *SQLPathExpression* object to be deleted as returned by either
cniCreateModifiablePathExpression or cniCreateReadOnlyPathExpression
functions. Tyhis parameter cannot be NULL.

## Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the
error..

## Example

Expanding on the example for cniSqlCreateReadOnlyPathExpression, include the
following code in _deleteNodeContext.

```
cniSqlDeletePathExpression(
                NULL,
                ((NODE_CONTEXT_ST *)context)->pathExpression);
```

# cniSqlDeleteStatement

Use this function to delete an SQL statement that you have created by using the
cniSqlCreateStatement utility function, as defined by the *sqlExpression* parameter.

## Syntax

```
void cniSqlDeleteStatement(
  int*            returnCode,
  CciSqlExpression*  sqlExpression);
```

## Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CC_INV_SQL_EXPR_OBJECT

**sqlExpression**
The SQL expression object to be deleted, as returned by the
cniSqlCreateStatement utility function (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniSqlExecute

Use this function to run an SQL statement that you have created by using the cniSqlCreateStatement utility function, as defined by the *sqlExpression* parameter. Use this function when the statement does not return data, for example, when a PASSTHRU function is used.

### Syntax

```
void cniSqlExecute(
  int*              returnCode,
  CciSqlExpression*  sqlExpression,
  CciMessage*        localEnvironment,
  CciMessage*        exceptionList,
  CciMessage*        message);
```

### Parameters

**returnCode**
The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_INV_SQL_EXPR_OBJECT
- CCI_INV_MESSAGE_OBJECT

**sqlExpression**
The SQL expression object to be executed, as returned by the cniSqlCreateStatement utility function (input).

**localEnvironment**
The message representing the input local environment (input).

For compatibility with earlier versions, you can refer to this parameter as destinationList.

**exceptionList**
The message representing the input exception list (input).

**message**
The message representing the input message (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniSqlNavigatePath

Use this function to run the SQLPathExpression that you have created by calling the cniSqlCreateReadOnlyPathExpression or the cniSqlCreateModifiablePathExpression utility functions, as defined by the *sqlPathExpression* argument.

## Syntax

```
CciElement* cniSqlNavigatePath(
 int*                returnCode,
 CciSqlPathExpression* sqlPathExpression,
 CciMessage*          inputMessageRoot,
 CciMessage*          inputLocalEnvironment,
 CciMessage*          inputExceptionList,
 CciMessage*          outputMessageRoot
 CciMessage*          outputLocalEnvironment,
 CciMessage*          outputExceptionList);
```

## Parameters

**returnCode (output)**
A NULL pointer input signifies that the user-defined node does not handle errors. Any exceptions that are thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. A call to cciGetLastExceptionData provides details of the exception. If an invalid *sqlPathExpression* parameter was passed in, *returnCode* is set to CCI_INV_SQL_EXPR_OBJECT. If an invalid CciMessage* value is passed in, *returnCode* is set to CCI_INV_MESSAGE_OBJECT. If the element could not be navigated to or created, *returnCode* is set to CCI_PATH_NOT_NAVIGABLE.

**sqlPathExpression (input)**
Specifies the SQLPathExpression object to be executed as returned by either the cniCreateReadOnlyPathExpression or the cniCreateModifyablePathExpression function. This parameter cannot be NULL.

**inputMessageRoot (input)**
The message representing the input message. This parameter cannot be NULL.

**inputLocalEnvironment (input)**
The message representing the input local environment. This parameter cannot be NULL.

**inputExceptionList (input)**
The message representing the input exception list. This parameter cannot be NULL.

**outputMessageRoot (input)**
The message representing the output message. This parameter can be NULL.

**outputLocalEnvironment (input)**
The message representing the output local environment. This parameter can be NULL.

**outputExceptionList (input)**
The message representing the output exception list. This parameter can be NULL.

The following table shows the mapping between the correlation names accepted in the ESQL path expression and the data that is accessed.

| Correlation name | Data accessed |
| --- | --- |
| Environment | The single Environment tree for the flow. This element is determined by the broker and it is not necessary to specify it with this API. |
| InputLocalEnvironment | inputLocalEnvironment parameter to cniSqlNavigatePath |

| Correlation name | Data accessed |
|---|---|
| OutputLocalEnvironment | outputLocalEnvironment parameter to cniSqlNavigatePath |
| InputRoot | inputMessageRoot parameter to cniSqlNavigatePath |
| InputBody | Last child of InputRoot |
| InputProperties | InputRoot.Properties (InputRoot.Properties is the first child of InputRoot, named "Properties") |
| OutputRoot | outputMessageRoot parameter to cniSqlNavigatePath |
| InputExceptionList | inputExceptionList parameter to cniSqlNavigatePath |
| OutputExceptionList | outputExceptionList parameter to cniSqlNavigatePath |
| Database | ODBC datasource identified by dataSourceName parameter to cniCreateReadOnlyPathExpression or cniCreateModifyablePathExpression |
| InputDestinationList | Synonym for InputLocalEnvironment that is compatible with earlier versions |
| OutputDestinationList | Synonym for OutputLocalEnvironment that is compatible with earlier versions |

All other rules regarding the actual navigability and validity of paths are defined in Correlation names.

## Return values

If the path is navigated successfully, the address of the syntax element is returned. However, if the path is not navigable, a value of zero (CCI_NULL_ADDR) is returned, and the *returnCode* parameter indicates the reason for the error.

## Example

Assuming that you have previously created a SQLPathExpression (see the example for cniSqlCreateReadOnlyPathExpression or cniSqlCreateModifiablePathExpression), you could use the following code to navigate to the target element.

```
CciElement* targetElement = cniSqlNavigatePath(
                    NULL,
                    ((NODE_CONTEXT_ST *)context)->pathExpression,
                    message,
                    localEnvironment,
                    exceptionList,
                    NULL, /* do not reference any output trees*/
                    NULL,
                    NULL);
```

# cniSqlSelect

Use this function to run an SQL statement that you have created by using the cniSqlCreateStatement utility function, as defined by the *sqlExpression* parameter. If the statement returns data, the data is written into the message specified by the *outputMessage* parameter.

## Syntax

```
void cniSqlSelect(
  int*               returnCode,
  CciSqlExpression*  sqlExpression,
```

```
CciMessage*        localEnvironment,
CciMessage*        exceptionList,
CciMessage*        message,
CciMessage*        outputMessage);
```

## Parameters

**returnCode**
>    The return code from the function (output).
>
>    Possible return codes are:
>    - CCI_SUCCESS
>    - CCI_EXCEPTION
>    - CCI_INV_SQL_EXPR_OBJECT
>    - CCI_INV_MESSAGE_OBJECT

**sqlExpression**
>    The SQL expression object to be executed, as returned by the
>    cniSqlCreateStatement utility function (input).

**localEnvironment**
>    The message representing the input local environment (input).
>
>    For compatibility with earlier versions, you can refer to this parameter as
>    destinationList.

**exceptionList**
>    The message representing the input exception list (input).

**message**
>    The message representing the input message (input).

**outputMessage**
>    The message into which any data returned by the statement is written
>    (output).

## Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the
error.

# cniWriteBuffer

Use this function to write the syntax element tree associated with the specified
message to the data buffer that is owned by the message object. This function is
typically used by output nodes.

This operation serializes the element tree into a bit stream that can then be
processed as a sequence of contiguous bytes. This function should be used when
writing the bit stream to a target that is outside the broker.

You must call cniFinalize before this call, or cniWriteBuffer fails.

## Syntax
```
void cniWriteBuffer(
  int*        returnCode,
  CciMessage*  message);
```

## Parameters

**returnCode**
>    The return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_MESSAGE_OBJECT

**message**
The address of the message object for which the element tree is to be serialized (input).

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

### Example
```
cniCopyElementTree(&rc, inLastChild, outLastChild);
cniFinalize(&rc, outMessage);
cniWriteBuffer(&rc, outMessage);
```

# C language user-defined parser API

The C language user-defined parser API consists of two complementary sets of functions that you can use to define the function of your parser.

The two sets of functions are:

1. A set of implementation functions that provide the functionality of the user-defined parser. These functions are called by the broker. Most implementation functions are mandatory and, if not supplied by the developer, cause an exception at run time.

2. A set of utility functions that create resources in the broker or request a service of the broker. These utility functions can be called by a user-defined parser.

These functions are defined in the `BipCpi.h` header file.

This section covers the following topics:
- "C parser implementation functions."
- "C parser utility functions" on page 198.

# C parser implementation functions

A user-defined parser implements its capability through a function interface which is called by the broker during runtime operation. This interface includes functions to create and delete all local context storage that is associated with a parser object and the parsing operations.

Some implementation functions are mandatory, and must be implemented when you develop your own parser. Other functions are optional, or conditional.

**Mandatory functions**
- "cpiCreateContext" on page 207
- "cpiParseNextSibling" on page 232
- "cpiParsePreviousSibling" on page 233
- "cpiParseFirstChild" on page 230
- "cpiParseLastChild" on page 231

**Optional and conditional functions**

## C parser utility functions

A user-defined parser can call functions provided by the broker to create or define objects, such as message parser factories.

These system-provided functions relate to different types of operation of the parser:

**Initialization and resource creation**

**Message buffer access**

**Syntax element navigation**

**Syntax element access**

# cpiAddAfter

This function adds a new (and currently unattached) syntax element to the syntax element tree after the specified target element. The newly added element becomes the *next sibling* of the target element.

## Syntax

```
void cpiAddAfter(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**newElement**
Specifies the address of the new syntax element object that is to be added to the tree structure (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

```
void cpiSetElementValue(
  CciParser*      parser,
  CciElement*     element,
  CciElementValue* value
){
  CciElement* newElement;
  int         rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE)) {
```

```
        cpiSetElementValueValue(&rc, element, value);
    }
    else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
        /* Create a new value element, add after the current value element,
     and set the value */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_VALUE);
        cpiSetElementValueValue(&rc, newElement, value);
        cpiAddAfter(&rc, element, newElement);
    }
    else {
    }

    return;
}
```

# cpiAddAsFirstChild

This function adds a new (and currently unattached) syntax element to the syntax
element tree as the first child of the specified target element.

## Syntax

```
void cpiAddAsFirstChild(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**newElement**
Specifies the address of the new syntax element object that is to be added to
the tree structure (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
void cpiSetElementValue(
  CciParser*      parser,
  CciElement*     element,
  CciElementValue* value
){
  CciElement* newElement;
  int         rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE)) {
    cpiSetElementValueValue(&rc, element, value);
```

```
    }
    else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
      /* Create a new value element, add as a first child, and set the value */
      newElement = cpiCreateElement(&rc, parser);
      cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_VALUE);
      cpiSetElementValueValue(&rc, newElement, value);
      cpiAddAsFirstChild(&rc, element, newElement);
    }
    else {
    }

    return;
}
```

# cpiAddAsLastChild

This function adds a new (and currently unattached) syntax element to the syntax element tree as the last child of the specified target element.

## Syntax

```
void cpiAddAsLastChild(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**newElement**
Specifies the address of the new syntax element object that is to be added to the tree structure (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetElementName(&rc, newElement, data);

        /* Free the memory created in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
```

# cpiAddBefore

This function adds a new (and currently unattached) syntax element to the syntax element tree before the specified target element. The newly added element becomes the *previous sibling* of the target element.

## Syntax

```
void cpiAddBefore(
  int*        returnCode,
  CciElement* targetElement,
  CciElement* newElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**newElement**
Specifies the address of the new syntax element object that is to be added to the tree structure (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

```
void cpiSetElementValue(
  CciParser*      parser,
  CciElement*     element,
  CciElementValue* value
){
  CciElement* newElement;
  int        rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE)) {
    cpiSetElementValueValue(&rc, element, value);
  }
  else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
    /* Create a new value element, add before the current value element,
   and set the value */
    newElement = cpiCreateElement(&rc, parser);
    cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_VALUE);
    cpiSetElementValueValue(&rc, newElement, value);
    cpiAddBefore(&rc, element, newElement);
  }
  else {
  }

  return;
}
```

# cpiAppendToBuffer

This function appends data to the buffer containing the bit stream representation of a message, for the specified parser object.

## Syntax

```
void cpiAppendToBuffer(
  int*       returnCode,
  CciParser* parser,
  CciByte*   data,
  CciSize    length);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_PARSER_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_LENGTH

**parser**
Specifies the address of the parser object (input).

**data**
The address of the data to be appended to the buffer (input).

**length**
The size in bytes of the data to be appended to the buffer (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);
```

# cpiBufferByte

This function gets a single byte from the buffer containing the bit stream representation of the input message, for the specified parser object. The value of the index argument indicates which byte in the byte array is to be returned.

## Syntax

```
CciByte cpiBufferByte(
  int*       returnCode,
  CciParser* parser,
  CciSize    index);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:

- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_PARSER_OBJECT
- CCI_NO_BUFFER_EXISTS

**parser**
Specifies the address of the parser object (input).

**index**
Specifies the offset to use as an index into the buffer (input).

### Return values

The requested byte is returned. If an error occurs, *returnCode* indicates the reason for the error.

### Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
void advance(
  PARSER_CONTEXT_ST* context,
  CciParser*         parser
){
  int rc = 0;

  /* Advance to the next character */
  context->iIndex++;

  /* Detect and handle the end condition */
  if (context->iIndex == context->iSize) return;

  /* Obtain the next character from the buffer */
  context->iCurrentCharacter = cpiBufferByte(&rc, parser, context->iIndex);
}
```

## cpiBufferPointer

This function gets a pointer to the buffer containing the bit stream representation of the input message, for the specified parser object.

### Syntax

```
const CciByte* cpiBufferPointer(
  int*        returnCode,
  CciParser*  parser);
```

### Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_PARSER_OBJECT
- CCI_NO_BUFFER_EXISTS

**parser**
Specifies the address of the parser object (input).

## Return values

If successful, the address of the buffer is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiParseBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int               rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc;, parser);
  pc->iIndex = 0;
```

# cpiBufferSize

This function gets the size of the buffer that contains the bit stream representation of the input message, for the specified parser object.

## Syntax

```
CciSize cpiBufferSize(
  int*        returnCode,
  CciParser*  parser);
```

## Parameters

**returnCode**
  Receives the return code from the function (output).

  Possible return codes are:
  - CCI_SUCCESS
  - CCI_EXCEPTION
  - CCI_INV_PARSER_OBJECT
  - CCI_NO_BUFFER_EXISTS

**parser**
  Specifies the address of the parser object (input).

## Return values

If successful, the size of the buffer, in bytes, is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample node file `BipSampPluginParser.c`:

```
int cpiParseBufferEncoded(
  CciParser*  parser,


  CciContext* context,
  int         encoding,
  int         ccsid
```

```
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int            rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
  pc->iIndex = 0;

  /* Save the format of the buffer */
  pc->iEncoding = encoding;
  pc->iCcsid = ccsid;

  /* Save size of the buffer */
  pc->iSize = cpiBufferSize(&rc, parser);
```

# cpiCreateAndInitializeElement

This function creates a syntax element, owned by the specified parser, that is not attached to a syntax tree. The element is partially initialized with the values of the *type*, *name*, *firstChildComplete*, and *lastChildComplete* parameters.

## Syntax

```
CciElement* cpiCreateAndInitializeElement(
  int*           returnCode,
  CciParser*     parser,
  CciElementType type,
  const CciChar* name,
  CciBool        firstChildComplete,
  CciBool        lastChildComplete);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_FAILURE
- CCI_INV_PARSER_OBJECT

**parser**
Specifies the address of the parser object (input). This address is passed to the parser as a parameter of the cpiCreateContext implementation function.

**type**
Specifies the type of the element being created (input).

**name**
Specifies a descriptive name for the element (input).

**firstChildComplete**
Specifies a value for the firstChildComplete flag of the syntax element (input).

**lastChildComplete**
Specifies a value for the lastChildComplete flag of the syntax element (input).

## Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason for the error.

### Sample

```
/* Advance to the end of the value */
        while (pc->iCurrentCharacter != quoteChar) {
          advance( (PARSER_CONTEXT_ST *)context, parser );
        }

        /* Get a pointer to the end of the tag */
        endMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Compute the size of the tag */
        markedSize = (size_t)endMarker-(int)startMarker;

        /* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

        /* Create a new name-value element for the attribute */
        newElement = cpiCreateAndInitializeElement(&rc, parser, type, name);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetElementName(&rc, newElement, data);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);
```

# cpiCreateContext

This function creates a user-defined extension context associated with a parser
object. It is called by the broker when an instance of a parser object is constructed
or allocated. This action occurs when a message flow causes the message data to
be parsed; the broker constructs or allocates a parser object to acquire the
appropriate section of the message data.

Before this function is called, the broker creates a name element as the effective
root element for the parser. However, this element is not named. The parser must
name this element in the cpiSetElementName function.

The responsibilities of the extension are to:

1. Allocate all parser-instance specific data areas (such as context) that might be
   required.
2. Perform all additional resource acquisition or initialization that might be
   required.
3. Return the address of the context to the calling function. Whenever an
   implementation function for this parser instance is called, the appropriate
   context is passed as an argument to that function. Therefore, a user-defined
   parser developed in C need not maintain its own static pointers to per-instance
   data areas.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Mandatory | iFpCreateContext |

## Syntax

```
void cpiCreateContext(
  CciParser*  parser);
```

## Parameters

**parser**
   The address of the parser object (input).

## Return values

If successful, the address of the user-defined extension context is returned. Otherwise, a value of zero is returned.

# cpiCreateElement

This function creates a default syntax element that is not attached to a syntax tree. The element is owned by the specified parser. The element is incomplete in that none of its attributes (such as type or name) are set.

## Syntax

```
CciElement* cpiCreateElement(
  int*        returnCode,
  CciParser*  parser);
```

## Parameters

**returnCode**
   Receives the return code from the function (output).

   Possible return codes are:
   - CCI_SUCCESS
   - CCI_EXCEPTION
   - CCI_FAILURE
   - CCI_INV_PARSER_OBJECT

**parser**
   Specifies the address of the parser object (input).

## Return values

If successful, the address of the new element object is returned. Otherwise, a value of zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
/* Advance to the end of the value */
        while (pc->iCurrentCharacter != quoteChar) {
          advance( (PARSER_CONTEXT_ST *)context, parser );
        }

        /* Get a pointer to the end of the tag */
        endMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Compute the size of the tag */
        markedSize = (size_t)endMarker-(int)startMarker;
```

```
                  /* Convert the attribute value into broker form */
                  data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

                  /* Create a new name-value element for the attribute */
                  newElement = cpiCreateElement(&rc, parser);
                  cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
                  cpiSetElementName(&rc, newElement, data);
                  if (pc->trace) {
                    const char * mbData = mbString(data, pc->iCcsid);
                    fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                            object=0x%x type=0x%x name=",
                            newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
                    fprintf(pc->tracefile, "%s\n", mbData);
                    fflush(pc->tracefile);
                    free((void *)mbData);
                  }
                  /* Free the memory created in CciNString() */
                  free((void *)data);
```

# cpiCreateParserFactory

This function creates a single instance of the named parser factory in the broker.

## Purpose

It must be called only in the initialization function bipGetParserFactory, which is
called when the LIL file is loaded by the broker. If cpiCreateParserFactory is called
at a later time, the results are unpredictable.

## Syntax

```
CciFactory* cpiCreateParserFactory(
  int*      returnCode,
  CciChar*  name);
```

## Parameters

**returnCode**
   Receives the return code from the function (output).

   Possible return codes are:
   - CCI_SUCCESS
   - CCI_EXCEPTION
   - CCI_FAILURE
   - CCI_INV_FACTORY_NAME
   - CCI_INV_OBJECT_NAME

**name**
   Specifies the name of the factory being created (input).

## Return values

If successful, the address of the parser factory object is returned. Otherwise, a
value of zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason
for the error.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
void LilFactoryExportPrefix * LilFactoryExportSuffix bipGetParserFactory()
{
  /* Declare variables */
  CciFactory*    factoryObject;
  int            rc;
  static CPI_VFT vftable = {CPI_VFT_DEFAULT};

  /* Before we proceed we need to initialise all the static constants */
  /* that may be used by the plug-in.                                 */
  initParserConstants();

  /* Setup function table with pointers to parser implementation functions */
  vftable.iFpCreateContext          = cpiCreateContext;
  vftable.iFpParseBufferEncoded     = cpiParseBufferEncoded;
  vftable.iFpParseFirstChild        = cpiParseFirstChild;
  vftable.iFpParseLastChild         = cpiParseLastChild;
  vftable.iFpParsePreviousSibling   = cpiParsePreviousSibling;
  vftable.iFpParseNextSibling       = cpiParseNextSibling;
  vftable.iFpWriteBufferEncoded     = cpiWriteBufferEncoded;
  vftable.iFpDeleteContext          = cpiDeleteContext;
  vftable.iFpSetElementValue        = cpiSetElementValue;
  vftable.iFpElementValue           = cpiElementValue;
  vftable.iFpNextParserClassName    = cpiNextParserClassName;
  vftable.iFpSetNextParserClassName = cpiSetNextParserClassName;
  vftable.iFpNextParserEncoding     = cpiNextParserEncoding;
  vftable.iFpNextParserCodedCharSetId = cpiNextParserCodedCharSetId;

  /* Create the parser factory for this plugin */
  factoryObject = cpiCreateParserFactory(&rc, constParserFactory);
  if (factoryObject) {
    /* Define the classes of message supported by the factory */
    cpiDefineParserClass(&rc, factoryObject, constPXML, &vftable);
  }
  else {
    /* Error: Unable to create parser factory */
  }

  /* Return address of this factory object to the broker */
  return(factoryObject);
}
```

## cpiDefineParserClass

This function defines the name of a parser class that is supported by a parser
factory.

*functbl* is a pointer to a virtual function table that contains pointers to the C
implementation functions; that is, those functions that provide the function of the
parser itself.

### Syntax

```
void cpiDefineParserClass(
  int*        returnCode,
  CciFactory* factoryObject,
  CciChar*    name,
  CPI_VFT*    functbl);
```

### Parameters

**returnCode**
> Receives the return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS

- CCI_EXCEPTION
- CCI_INV_FACTORY_OBJECT
- CCI_INV_PARSER_NAME
- CCI_PARSER_NAME_TOO_LONG
- CCI_INV_OBJECT_NAME
- CCI_INV_VFTP
- CCI_MISSING_IMPL_FUNCTION
- CCI_INV_IMPL_FUNCTION
- CCI_NAME_EXISTS

**factoryObject**
Specifies the address of the factory object that supports the named parser (input). The address is returned from **cpiCreateParserFactory**.

**name**
The name of the parser class to be defined (input). The maximum length of a parser class name is 8 characters.

**functbl**
The address of the CPI_VFT structure that contains pointers to the implementation functions (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
void LilFactoryExportPrefix * LilFactoryExportSuffix bipGetParserFactory()
{
  /* Declare variables */
  CciFactory*     factoryObject;
  int             rc;
  static CPI_VFT  vftable = {CPI_VFT_DEFAULT};

  /* Before we proceed we need to initialise all the static constants */
  /* that may be used by the plug-in.                                 */
  initParserConstants();

  /* Setup function table with pointers to parser implementation functions */
  vftable.iFpCreateContext          = cpiCreateContext;
  vftable.iFpParseBufferEncoded     = cpiParseBufferEncoded;
  vftable.iFpParseFirstChild        = cpiParseFirstChild;
  vftable.iFpParseLastChild         = cpiParseLastChild;
  vftable.iFpParsePreviousSibling   = cpiParsePreviousSibling;
  vftable.iFpParseNextSibling       = cpiParseNextSibling;
  vftable.iFpWriteBufferEncoded     = cpiWriteBufferEncoded;
  vftable.iFpDeleteContext          = cpiDeleteContext;
  vftable.iFpSetElementValue        = cpiSetElementValue;
  vftable.iFpElementValue           = cpiElementValue;
  vftable.iFpNextParserClassName    = cpiNextParserClassName;
  vftable.iFpSetNextParserClassName = cpiSetNextParserClassName;
  vftable.iFpNextParserEncoding     = cpiNextParserEncoding;
  vftable.iFpNextParserCodedCharSetId = cpiNextParserCodedCharSetId;

  /* Create the parser factory for this plugin */
  factoryObject = cpiCreateParserFactory(&rc, constParserFactory);
  if (factoryObject) {
    /* Define the classes of message supported by the factory */
```

```
            cpiDefineParserClass(&rc, factoryObject, constPXML, &vftable);
      }
      else {
        /* Error: Unable to create parser factory */
      }

      /* Return address of this factory object to the broker */
      return(factoryObject);
}
```

# cpiDeleteContext

This function deletes the context owned by the parser object. It is invoked by the message broker when an instance of a parser object is destroyed.

The responsibilities of the parser are to:

1. Release all parser-instance specific data areas (such as context) that were acquired at construction or during parser processing.
2. Release all additional resources that might have been acquired for the processing of the parser.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Optional | iFpDeleteContext |

## Syntax

```
void cpiDeleteContext(
  CciParser*  parser,
  CciContext* context);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

## Return values

None.

# cpiElementCompleteNext

This function gets the value of the 'next child complete' flag from the target syntax element. This attribute indicates whether the element tree is complete.

## Syntax

```
CciBool cpiElementCompleteNext(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**
    Receives the return code from the function (output).

    Possible return codes are:
    • CCI_SUCCESS
    • CCI_EXCEPTION

- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

### Return values

The value of the attribute is returned. If an error occurs, *returnCode* indicates the reason for the error.

### Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
if ((!cpiElementCompleteNext(&rc, element)) &&
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

   while ((!cpiElementCompleteNext(&rc, element))     &&
          (!cpiFirstChild(&rc, element)) &&
          (pc->iCurrentElement))
   {
     pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
   }
```

# cpiElementCompletePrevious

This function gets the value of the 'previous child complete' flag from the target syntax element. This attribute indicates whether the element tree is complete.

### Syntax

```
CciBool cpiElementCompletePrevious(
  int*        returnCode,
  CciElement*  targetElement);
```

### Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

### Return values

The value of the attribute is returned. If an error occurs, *returnCode* indicates the reason for the error.

### Sample

This example is based on code taken from the sample parser file `BipSampPluginParser.c`. In the sample file, the code uses cpiElementCompleteNext.

```
if ((!cpiElementCompletePrevious(&rc, element)) &&
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

   while ((!cpiElementCompletePrevious(&rc, element))     &&
          (!cpiFirstChild(&rc, element)) &&
```

```
                 (pc->iCurrentElement))
    {
      pc->iCurrentElement = parsePreviousItem(parser, context, pc->iCurrentElement);
    }
```

## cpiElementName

This function gets the name of the target syntax element. The syntax element name must be set previously by using cniSetElementName or cpiSetElementName.

### Syntax

```
Ccisize          cpiElementName(
  int*           returnCode,
  CciElement*    targetElement,
  const CciChar* value,
  CciSize        length);
```

### Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**targetElement**
Specifies the address of the target syntax element object (input).

**value**
Specifies the address of a buffer into which the element name is copied (input).

**length**
The length, in characters, specified by the *value* parameter (input).

### Return values

If successful, the element name is copied into the supplied buffer and the number of *CciChar* characters copied is returned. If the buffer is not large enough to contain the element name, *returnCode* is set to CCI_BUFFER_TOO_SMALL and the number of characters required is returned. For all other failures, CCI_FAILURE is returned and *returnCode* indicates the reason for the error.

### Sample

```
cpiElementName(&rc;, element, (CciChar*)&elementName;, sizeof(elementName));
```

## cpiElementNameSpace

This function retrieves the value of the *namespace* attribute for the specified syntax element.

| Defined In | Type | Member |
|------------|----------|---------------|
| CPI_VFT | Optional | iFpElementValue |

## Syntax

```
CciSize cpiElementNamespace(
   int*          returnCode,
   CciElement*   targetElement,
   const CciChar* value,
   CciSize        length);
```

## Parameters

**returnCode**

A NULL pointer input signifies that the user-defined node does not want to deal with errors. All exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. You can call CciGetLastExceptionData to get details of the exception. If the caller did not allocate enough memory to hold the namespace value, *returncode is set to CCI_BUFFER_TOO_SMALL.

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN
- CCI_INV_BUFFER_TOO_SMALL

**targetElement**

Specifies the address of the target syntax element object.

**value**

Specifies the address of a buffer into which the element namespace value is copied. A string of characters (including a NULL terminator) representing the namespace value is copied into this buffer. The buffer must be a portion of memory previously allocated by the caller.

**length**

The length in CciChars of the buffer specified by the value parameter.

## Return values

If successful, the number of CciChars copied into the buffer is returned.

If the buffer is not large enough to contain the attribute value, *returnCode* is set to CCI_BUFFER_TOO_SMALL, and the number of bytes CciChars required is returned.

## Sample

```
elementNamespace=(CciChar*)malloc(sizeof(CciChar) * elementNamespaceLength);
   elementNamespaceLength = cpiElementNamespace(&rc;,
                        element,
                        elementNamespace,
                        elementNamespaceLength);

   if (rc==CCI_BUFFER_TOO_SMALL){
     free(elementNamespace);
     elementNamespace=(CciChar*)malloc(sizeof(CciChar) * elementNamespaceLength);
     elementNamespaceLength = cpiElementNamespace(&rc;,
                        element,
```

```
                       elementNamespace,
                       elementNamespaceLength);
      }
    checkRC(rc);
```

# cpiElementType

This function gets the type of the target syntax element. The syntax element type must be set previously by using cniSetElementType or cpiSetElementType.

## Syntax

```
CciElementType cpiElementType(
  int*        returnCode,
  CciElement* targetElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

## Return values

The value of the element type is returned. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
if ((!cpiElementCompleteNext(&rc, element)) &&
    (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

  while ((!cpiElementCompleteNext(&rc, element))    &&
         (!cpiFirstChild(&rc, element)) &&
         (pc->iCurrentElement))
  {
    pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
  }
```

# cpiElementValue

This function gets the value of a specified element. It is called by the broker when the value of a syntax element is to be retrieved. It provides an opportunity for a user-defined parser to override the behavior for retrieving element values.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Optional | iFpElementValue |

## Syntax

```
const CciElementValue* cpiElementValue(
  CciParser*  parser,
  CciElement* currentElement);
```

## Parameters

**parser**
    The address of the parser object (input).

**currentElement**
    The address of the current syntax element (input).

## Return values

The value of the target syntax element object is returned. This value has been returned by the cpiElementValueValue function.

# cpiElementValue group

This group of functions retrieve the value of the specified syntax element.

Specify the appropriate function from this group that matches the type of data to be retrieved:
- cpiElementBitArrayValue
- cpiElementBooleanValue
- cpiElementByteArrayValue
- cpiElementCharacterValue
- cpiElementDateValue
- cpiElementDecimalValue
- cpiElementGmtTimestampValue
- cpiElementGmtTimeValue
- cpiElementIntegerValue
- cpiElementRealValue
- cpiElementTimestampValue
- cpiElementTimeValue

## Syntax

```
CciSize cpiElementBitArrayValue(
  int*        returnCode,
  CciElement* targetElement,
  const struct CciBitArray* value);

CciBool cpiElementBooleanValue(
  int*        returnCode,
  CciElement* targetElement);

CciSize cpiElementByteArrayValue(
  int*        returnCode,
  CciElement* targetElement,
  const struct CciByteArray* value);

CciSize cpiElementCharacterValue(
  int*           returnCode,
  CciElement*    targetElement,
  const CciChar* value,
  CciSize        length);

struct CciDate cpiElementDateValue(
  int*        returnCode,
  CciElement* targetElement);
```

```
CciSize cpiElementDecimalValue(
  int*          returnCode,
  CciElement*   targetElement,
  const CciChar* value,
  CciSize       length);
struct CciTimestamp cpiElementGmtTimestampValue(
  int*         returnCode,
  CciElement*  targetElement);
struct CciTime cpiElementGmtTimeValue(
  int*         returnCode,
  CciElement*  targetElement);
CciInt cpiElementIntegerValue(
  int*         returnCode,
  CciElement*  targetElement);
CciReal cpiElementRealValue(
  int*         returnCode,
  CciElement*  targetElement);
struct CciTimestamp cpiElementTimestampValue(
  int*         returnCode,
  CciElement*  targetElement);
struct CciTime cpiElementTimeValue(
  int*         returnCode,
  CciElement*  targetElement);
```

## Parameters

**returnCode**
> Receives the return code from the function (output).

> Possible return codes are:
> - CCI_SUCCESS
> - CCI_EXCEPTION
> - CCI_INV_ELEMENT_OBJECT
> - CCI_INV_DATA_POINTER
> - CCI_INV_DATA_BUFLEN
> - CCI_INV_BUFFER_TOO_SMALL

**targetElement**
> Specifies the address of the target syntax element object (input).

**value**
> The address of an output buffer into which the value of the syntax element is stored (input). Used on relevant function calls only.

**length**
> The length of the output buffer, in characters, specified by the *value* parameter (input). Used on relevant function calls only.

## Return values

The value of the element is returned.

In some cases, if the buffer is not large enough to receive the data, the data is not written into the buffer. The size of the required buffer is passed as the return value, and *returnCode* is set to CCI_BUFFER_TOO_SMALL. For example, cpiElementCharacterValue or cpiElementDecimalValue use this technique.

If an error occurs, *returnCode* indicates the reason for the error.

## cpiElementValueValue

This function gets the value object from the specified syntax element. This value object is opaque in that it cannot be interrogated. It can be used to set or derive the value of one element from another, without knowing its type, by using the cpiSetElementValueValue function.

This function can be used by parsers that override behavior by calling the implementation functions cpiElementValue and cpiSetElementValue.

### Syntax

```
const CciElementValue* cpiElementValueValue(
  int*        returnCode,
  CciElement*  targetElement);
```

### Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

### Return values

The address of the *CciElementValue* object stored in the specified target syntax element is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned and *returnCode* indicates the reason for the error.

### Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
const CciElementValue* cpiElementValue(
  CciParser*  parser,
  CciElement*  element
){
  CciElement* firstChild;
  const CciElementValue* value;
  int         rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE)) {
    value = cpiElementValueValue(&rc, element);
  }
  else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
    firstChild = cniFirstChild(&rc, element);
    value = cpiElementValueValue(&rc, firstChild);
  }
  else {
  }

  return(value);
}
```

# cpiFirstChild

This function returns the address of the syntax element object that is the first child of the specified target element.

## Syntax

```
CciElement* cpiFirstChild(
  int*              returnCode,
  const CciElement*  targetElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

## Return values

The address of the requested syntax element object is returned, unless there is no child in which case zero is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned and *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample node file `BipSampPluginParser.c`:

```
while ((!cpiElementCompleteNext(&rc, element))    &&
        (!cpiFirstChild(&rc, element)) &&
        (pc->iCurrentElement))
```

# cpiLastChild

This function returns the address of the syntax element object that is the last child of the specified target element.

## Syntax

```
CciElement* cpiLastChild(
  int*              returnCode,
  const CciElement*  targetElement);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

## Return values

The address of the requested syntax element object is returned, unless there is no child in which case zero is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned and *returnCode* indicates the reason for the error.

# cpiNextParserClassName

This function returns the name of the next parser class in the chain, if defined. Use this function to return to the broker the name of the parser class that handles the next section, or remainder, of the message content. Typically, for messages that have a simple format type, only one message content parser is defined.

For messages that have a more complex format type with multiple message parsers, each parser can identify the next one in the chain by returning its name in the *buffer* parameter. The last parser in the chain must return an empty string.

If you specify the name of a parser supplied with WebSphere Message Broker, you must use the correct class name of the parser.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Optional | iFpNextParserClassName |

## Syntax

```
void cpiNextParserClassName(
  CciParser*  parser,
  CciContext* context,
  CciChar*    buffer,
  int         size);
```

## Parameters

**parser**
　　The address of the parser object (input).

**context**
　　The address of the context owned by the parser object (input).

**buffer**
　　The address of a buffer into which the parser class name should be put (input).

**size**
　　The length, in bytes, of the buffer provided by the broker (input).

## Return values

None.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiNextParserClassName(
  CciParser*  parser,
  CciContext* context,
  CciChar*    buffer,
  int         size
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc = 0;
```

```
        if (pc->trace) {
          fprintf(pc->tracefile, "PLUGIN: -> cpiNextParserClassName() parser=0x%x context=0x%x\n",
                  parser, context);
          fflush(pc->tracefile);
        }

        /* Copy the name to the broker */
        CciCharNCpy(buffer, pc->iNextParserClassName, size);

        if (pc->trace) {
          fprintf(pc->tracefile, "PLUGIN: <- cpiNextParserClassName()\n");
          fflush(pc->tracefile);
        }

        return;
      }
```

## cpiNextParserCodedCharSetId

This function returns the coded character set ID (CCSID) of the data owned by the next parser class in the chain, if one is defined.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Optional | iFpNextParserCodedCharSetId |

### Syntax

```
int cpiNextParserCodedCharSetId(
  CciParser*   parser,
  CciContext*  context);
```

### Parameters

**parser**
   The address of the parser object (input).

**context**
   The address of the context owned by the parser object (input).

### Return values

The CCSID of the data is returned. If it is not known, zero might be returned, and a default CCSID is assumed.

### Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiNextParserCodedCharSetId(
  CciParser*   parser,
  CciContext*  context
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                ccsid = 0;

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: -> cpiNextParserCodedCharSetId() parser=0x%x
        context=0x%x\n", parser, context);
    fflush(pc->tracefile);
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiNextParserCodedCharSetId()\n");
```

```
      fflush(pc->tracefile);
  }

  return ccsid;
}
```

# cpiNextParserEncoding

This function returns the encoding of data owned by the next parser class in the chain, if defined.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Optional | iFpNextParserEncoding |

## Syntax

```
int cpiNextParserEncoding(
  CciParser*  parser,
  CciContext*  context);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

## Return values

The encoding of the data is returned. If it is not known, zero might be returned, and default encoding is assumed.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiNextParserEncoding(
  CciParser*  parser,
  CciContext* context
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int              encoding = 0;

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: -> cpiNextParserEncoding() parser=0x%x context=0x%x\n",
            parser, context);
    fflush(pc->tracefile);
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiNextParserEncoding()\n");
    fflush(pc->tracefile);
  }

  return encoding;
}
```

# cpiNextSibling

This function returns the address of the syntax element object that is the next (right) sibling of the specified target element.

### Syntax

```
CciElement* cpiNextSibling(
  int*             returnCode,
  const CciElement*  targetElement);
```

### Parameters

**returnCode**
   Receives the return code from the function (output).

   Possible return codes are:
   * CCI_SUCCESS
   * CCI_EXCEPTION
   * CCI_INV_ELEMENT_OBJECT

**targetElement**
   Specifies the address of the target syntax element object (input).

### Return values

The address of the requested syntax element object is returned, unless there is no next sibling in which case zero is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned and *returnCode* indicates the reason for the error.

### Sample

This example is taken from the sample node file `BipSampPluginParser.c`:

```
while ((!cpiElementCompleteNext(&rc, cpiParent(&rc, element))) &&
        (!cpiNextSibling(&rc, element))      &&
        (pc->iCurrentElement))
```

# cpiParent

This function returns the address of the syntax element object that is the parent of the specified target element.

### Syntax

```
CciElement* cpiParent(
  int*             returnCode,
  const CciElement*  targetElement);
```

### Parameters

**returnCode**
   Receives the return code from the function (output).

   Possible return codes are:
   * CCI_SUCCESS
   * CCI_EXCEPTION
   * CCI_INV_ELEMENT_OBJECT

**targetElement**
   Specifies the address of the target syntax element object (input).

### Return values

If successful, the address of the requested syntax element is returned. If there is no parent element, zero is returned. If an error occurs, zero (CCI_NULL_ADDR) is

returned and the *returnCode* parameter indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void* parseNextItem(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  void*              endMarker;
  void*              startMarker;
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context;
  CciElement*        returnElement = element;
  CciElement*        newElement;
  size_t             markedSize;
  const CciChar*     data;
  int                rc;

  if (pc->trace)

  /* Skip any white space */
  skipWhiteSpace( (PARSER_CONTEXT_ST *)context );

  /* Are we at the end of the buffer? */
  if (pc->iIndex == pc->iSize)
        return(0);
  }

  /* Are we within a tag? */
  if (pc->iInTag) {

    if (pc->iCurrentCharacter == chCloseAngle) {

      /* We have reached the end of a tag */
      pc->iInTag = 0;
      advance( (PARSER_CONTEXT_ST *)context, parser );
    }
    else if (pc->iCurrentCharacter == chForwardSlash) {

      /* We may have reached the end of an empty tag */
      advance( (PARSER_CONTEXT_ST *)context, parser );

      if (pc->iCurrentCharacter == chCloseAngle) {

        pc->iInTag = 0;
        advance( (PARSER_CONTEXT_ST *)context, parser );

        cpiSetElementCompleteNext(&rc, element, 1);

        returnElement = cpiParent(&rc, element);
      }
```

## cpiParseBuffer

This function prepares a parser to parse a new message object. It is called the first time (for each message) that the message flow causes the message content to be parsed.

This function is called for each user-defined parser that is used to parse a particular message format to complete the following actions:
- Perform all initialization that is required
- Return the length of the message content that it takes ownership for

The *offset* parameter indicates the offset within the message buffer where parsing is to commence. This indication is required because another parser might own a previous portion of the message (for example, an MQMD header is parsed by an internal parser owned by the broker). The offset must be positive and be less than the size of the buffer. Verify in this function that the offset is valid, to remove problems associated with previous parsers.

The parser must return the size of the remaining buffer for which it takes ownership. The size must be less than or equal to the size of the buffer, less the current offset.

A parser must not attempt to cause parsing of other portions of the syntax element tree, for example, by navigating to the root element and to another branch. This action can cause unpredictable results.

If this implementation function is provided in the CPI_VFT structure, you can call neither cpiParseBufferEncoded() nor cpiParseBufferFormatted(), because the cpiDefineParserClass() function fails with a return code of CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Conditional | iFpParseBuffer |

## Syntax

```
int cpiParseBuffer(
  CciParser*  parser,
  CciContext* context,
  int         offset);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

**offset**
    The offset into the message buffer at which parsing is to commence (input).

## Return values

The size (in bytes) of the remaining portion of the message buffer for which the parser takes ownership.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiParseBuffer(
  CciParser*  parser,
  CciContext* context,
  int         offset,
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
```

```
pc->iIndex = 0;

/* Save size of the buffer */
pc->iSize = cpiBufferSize(&rc, parser);

/* Prime the first byte in the stream */
pc->iCurrentCharacter = cpiBufferByte(&rc, parser, pc->iIndex);

/* Set the current element to the root element */
pc->iCurrentElement = cpiRootElement(&rc, parser);

/* Reset flag to ensure parsing is reset correctly */
pc->iInTag = 0;

if (pc->trace) {
  fprintf(pc->tracefile, "PLUGIN: <- cpiParseBuffer()
  retvalue=%d\n", pc->iSize);
  fflush(pc->tracefile);
}
```

# cpiParseBufferEncoded

This function extends the capability of the cpiParseBuffer() implementation function, and provides the encoding and coded character set that the input message is represented in.

If this implementation function is provided in the CPI_VFT structure, you cannot specify either cpiParseBuffer() or cpiParseBufferFormatted(); if you do, the cpiDefineParserClass() function fails with a return code of CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Conditional | iFpParseBufferEncoded |

## Syntax

```
int cpiParseBufferEncoded(
  CciParser*   parser,
  CciContext*  context,
  int          encoding,
  int          ccsid);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

**encoding**
    The encoding of the message buffer (input).

**ccsid**
    The CCSID of the message buffer (input).

## Return values

The size (in bytes) of the remaining portion of the message buffer for which the parser takes ownership.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiParseBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int               rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
  pc->iIndex = 0;

  /* Save the format of the buffer */
  pc->iEncoding = encoding;
  pc->iCcsid = ccsid;

  /* Save size of the buffer */
  pc->iSize = cpiBufferSize(&rc, parser);

  /* Prime the first byte in the stream */
  pc->iCurrentCharacter = cpiBufferByte(&rc, parser, pc->iIndex);

  /* Set the current element to the root element */
  pc->iCurrentElement = cpiRootElement(&rc, parser);

  /* Reset flag to ensure parsing is reset correctly */
  pc->iInTag = 0;

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiParseBufferEncoded()
    retvalue=%d\n", pc->iSize);
    fflush(pc->tracefile);
  }
```

# cpiParseBufferFormatted

This function extends the capability of the cpiParseBuffer() implementation function, and provides additional information about the input message.

The following additional information is available:

1. The encoding and coded character set that the input message is represented in.
2. The message set, type, and format for the message.

If this implementation function is provided in the CPI_VFT structure, you cannot specify either cpiParseBuffer() or cpiParseBufferEncoded(); if you do, the cpiDefineParserClass() function fails with a return code of CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Conditional | iFpParseBufferFormatted |

## Syntax

```
int cpiParseBufferFormatted(
  CciParser*     parser,
  CciContext*    context,
  int            encoding,
```

```
int          ccsid,
CciChar*     set,
CciChar*     type,
CciChar*     format);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

**encoding**
    The encoding of the message buffer (input).

**ccsid**
    The CCSID of the message buffer (input).

**set**
    The message set to which the message belongs (input).

**type**
    The message type (input).

**format**
    The message format (input).

## Return values

The size (in bytes) of the remaining portion of the message buffer for which the parser takes ownership.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
int cpiParseBufferFormatted(
  CciParser*     parser,
  CciContext*    context,
  int            encoding,
  int            ccsid,
  CciChar*       set,
  CciChar*       type,
  CciChar*       format
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int               rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
  pc->iIndex = 0;

  /* Save the format of the buffer */
  pc->iEncoding = encoding;
  pc->iCcsid = ccsid;

  /* Save size of the buffer */
  pc->iSize = cpiBufferSize(&rc, parser);

  /* Prime the first byte in the stream */
  pc->iCurrentCharacter = cpiBufferByte(&rc, parser, pc->iIndex);

  /* Set the current element to the root element */
  pc->iCurrentElement = cpiRootElement(&rc, parser);
```

```
                /* Reset flag to ensure parsing is reset correctly */
                pc->iInTag = 0;

                if (pc->trace) {
                  fprintf(pc->tracefile, "PLUGIN: <- cpiParseBufferFormatted()
                  retvalue=%d\n", pc->iSize);
                  fflush(pc->tracefile);
                }
```

# cpiParseFirstChild

This function parses the first child of a specified syntax element. It is called by the broker when the first child element of the current syntax element is required.

| Defined In | Type | Member |
|------------|------|--------|
| CPI_VFT | Mandatory | iFpParseFirstChild |

## Syntax

```
void cpiParseFirstChild(
  CciParser*  parser,
  CciContext* context,
  CciElement* currentElement);
```

## Parameters

**parser**
   The address of the parser object (input).

**context**
   The address of the context owned by the parser object (input).

**currentElement**
   The address of the current syntax element (input).

## Return values

None.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiParseFirstChild(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int             rc;

  if ((!cpiElementCompleteNext(&rc, element)) &&
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

    while ((!cpiElementCompleteNext(&rc, element))     &&
           (!cpiFirstChild(&rc, element)) &&
           (pc->iCurrentElement))
    {
      pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
    }
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiParseFirstChild()\n");
```

```
      fflush(pc->tracefile);
  }

  return;
}
```

# cpiParseLastChild

This function parses the last child of a specified syntax element. It is called by the broker when the last child element of the current syntax element is required.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Mandatory | iFpParseLastChild |

## Syntax

```
void cpiParseLastChild(
  CciParser*  parser,
  CciContext* context,
  CciElement* currentElement);
```

## Parameters

**parser**
  The address of the parser object (input).

**context**
  The address of the context owned by the parser object (input).

**currentElement**
  The address of the current syntax element (input).

## Return values

None.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c:

```
void cpiParseLastChild(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME)) {

    while ((!cpiElementCompleteNext(&rc, element))    &&
           (pc->iCurrentElement))
    {
      pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
    }
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiParseLastChild()\n");
    fflush(pc->tracefile);
  }

  return;
}
```

The purpose of this code is to parse children of an element until the last child is reached. You can use this kind of structure in a parser that does not already know the exact offset in the bit stream of the last child of an element.

# cpiParseNextSibling

This function parses the next (right) sibling of a specified syntax element. It is called by the broker when the next (right) sibling element of the current syntax element is required.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Mandatory | iFpParseNextSibling |

## Syntax

```
void cpiParseNextSibling(
  CciParser*   parser,
  CciContext*  context,
  CciElement*  currentElement);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

**currentElement**
    The address of the current syntax element (input).

## Return values

None.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiParseNextSibling(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int              rc;

    while ((!cpiElementCompleteNext(&rc, cpiParent(&rc, element))) &&
        (!cpiNextSibling(&rc, element))        &&
        (pc->iCurrentElement))
  {
    pc->iCurrentElement = parseNextItem(parser, context, pc->iCurrentElement);
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiParseNextSibling()\n");
    fflush(pc->tracefile);
  }

  return;
}
```

# cpiParsePreviousSibling

This function parses the previous (left) sibling of a specified syntax element. It is called by the broker when the previous (left) sibling element of the current syntax element is required.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Mandatory | iFpParsePreviousSibling |

## Syntax

```
void cpiParsePreviousSibling(
  CciParser*  parser,
  CciContext* context,
  CciElement* currentElement);
```

## Parameters

**parser**
  The address of the parser object (input).

**context**
  The address of the context owned by the parser object (input).

**currentElement**
  The address of the current syntax element (input).

## Return values

None.

## Sample

```
void cpiParsePreviousSibling(
  CciParser*  parser,
  CciContext* context,
  CciElement* element
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int               rc;

  while ((!cpiElementCompletePrevious(&rc, cpiParent(&rc, element))) &&
         (!cpiPreviousSibling(&rc, element))       &&
         (pc->iCurrentElement))
  {
    pc->iCurrentElement = parsePreviousItem(parser, context, pc->iCurrentElement);
  }

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiParsePreviousSibling()\n");
    fflush(pc->tracefile);
  }

  return;
}
```

The code sample is similar to that used for cpiParseNextSibling. Use cpiParsePreviousSibling as shown in the example when you are parsing the bit stream right to left.

## cpiParserType

This optional function returns whether the parser is an implementation of a *standard* parser.

A standard parser expects the Format field of the preceding header to contain the name of the parser class that follows. *Non-standard* parsers expects the Domain field to contain the parser class name. If the cpiParserType implementation function is not provided, the broker assumes that the parser is a standard parser.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Optional | iFpParserType |

### Syntax

```
CciBool cpiParserType(
  CciParser*   parser,
  CciContext*  context);
```

### Parameters

**parser**
   The address of the parser object (input).

**context**
   The address of the context owned by the parser object (input).

### Return values

If the implementation is of a standard parser, zero is returned. Otherwise, the implementation is assumed to be that of a non-standard parser, nd a non-zero value is returned.

## cpiRootElement

This function gets the address of the root syntax element of the specified parser object.

### Syntax

```
CciElement* cpiRootElement(
  int*         returnCode,
  CciParser*  parser);
```

### Parameters

**returnCode**
   Receives the return code from the function (output).

   Possible return codes are:
   - CCI_SUCCESS
   - CCI_EXCEPTION
   - CCI_INV_PARSER_OBJECT

**parser**
   Specifies the address of the parser object (input).

### Return values

The address of the root syntax element is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned, and *returnCode* indicates the reason for the error.

### Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiParseBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc;

  /* Get a pointer to the message buffer and set the offset */
  pc->iBuffer = (void *)cpiBufferPointer(&rc, parser);
  pc->iIndex = 0;

  /* Save the format of the buffer */
  pc->iEncoding = encoding;
  pc->iCcsid = ccsid;

  /* Save size of the buffer */
  pc->iSize = cpiBufferSize(&rc, parser);

  /* Prime the first byte in the stream */
  pc->iCurrentCharacter = cpiBufferByte(&rc, parser, pc->iIndex);

  /* Set the current element to the root element */
  pc->iCurrentElement = cpiRootElement(&rc, parser);

  /* Reset flag to ensure parsing is reset correctly */
  pc->iInTag = 0;

  /* We will assume ownership of the remainder of the buffer */
  return(pc->iSize);
}
```

# cpiSetCharacterValueFromBuffer

This function sets the value of the specified syntax element.

### Syntax

```
void cpiSetCharacterValueFromBuffer(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  value,
  CciSize         length);
```

### Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

- CCI_INV_DATA_BUFLEN

**targetElement**
Specifies the address of the target syntax element object (input).

**value**
The value to be set in the target element (input).

**length**
The length of the character string, expressed as the number of `CciChar` characters, specified by the *value* parameter (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetCharacterValueFromBuffer(&rc, newElement, data, length);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
```

# cpiSetElementCompleteNext

This function sets the 'next child complete' flag in the target syntax element to the specified value.

## Syntax

```
void cpiSetElementCompleteNext(
  int*        returnCode,
  CciElement* targetElement,
  CciBool     value);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**value**
    The value to be set in the flag (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
/* Get a pointer to the start of the tag */
        startMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Skip over the tag */
        goToNameEnd( (PARSER_CONTEXT_ST *)context, parser );

        /* Get a pointer to the end of the tag */
        endMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Compute the size of the tag */
        markedSize = (size_t)endMarker-(int)startMarker;

        /* Convert the tag into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

        /* Create a new name element for the tag */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME);
        cpiSetElementName(&rc, newElement, data);
        cpiSetElementCompletePrevious(&rc, newElement, 0);
        cpiSetElementCompleteNext(&rc, newElement, 0);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: New tag found\n");
          fprintf(pc->tracefile, "PLUGIN: Created new NAME element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory allocated in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
        cpiSetElementCompletePrevious(&rc, element, 1);
```

# cpiSetElementCompletePrevious

This function sets the 'previous child complete' flag in the target syntax element to the specified value.

## Syntax

```
void cpiSetElementCompletePrevious(
  int*        returnCode,
  CciElement* targetElement,
  CciBool     value);
```

## Parameters

**returnCode**
    Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**value**
The value to be set in the flag (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
/* Get a pointer to the start of the tag */
        startMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Skip over the tag */
        goToNameEnd( (PARSER_CONTEXT_ST *)context, parser );

        /* Get a pointer to the end of the tag */
        endMarker = (char*)pc->iBuffer+(int)pc->iIndex;

        /* Compute the size of the tag */
        markedSize = (size_t)endMarker-(int)startMarker;

        /* Convert the tag into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

        /* Create a new name element for the tag */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME);
        cpiSetElementName(&rc, newElement, data);
        cpiSetElementCompletePrevious(&rc, newElement, 0);
        cpiSetElementCompleteNext(&rc, newElement, 0);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: New tag found\n");
          fprintf(pc->tracefile, "PLUGIN: Created new NAME element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory allocated in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
        cpiSetElementCompletePrevious(&rc, element, 1);
```

# cpiSetElementName

This function sets the name of the specified syntax element.

## Syntax

```
void cpiSetElementName(
  int*         returnCode,
  CciElement*  targetElement,
  const CciChar*  name);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

**targetElement**
Specifies the address of the target syntax element object (input).

**name**
The name to be set in the target element (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetElementName(&rc, newElement, data);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
```

# cpiSetElementNamespace

Use this function to set the namespace attribute for the specified syntax element.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Optional | iFpSetElementValue |

## Syntax

```
void            cpiSetElementNamespace(
  int*             returnCode,
  CciElement*      targetElement,
  const CciChar*  nameSpace);
```

## Parameters

**returnCode**
A NULL pointer input value indicates that the user-defined node does not want to deal with errors. All exceptions thrown during the execution of this call are rethrown to the next upstream node in the flow. If the input value is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. Call CciGetLastExceptionData to obtain details of the exception. (input).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

**currentElement**
The address of the current syntax element (input).

**targetElement**
Specifies the address of the target syntax element object.

**value**
Specifies the address of a null terminated string of `CciChars` that represents the namespace value. An empty string is a valid value for namespace; elements are created in the empty string namespace by default, therefore if you specify an empty string as the namespace by using this call, the call is effective only if the element was previously in another namespace, and the required effect is to change the namespace value to empty string.

## Return values

None.

## Sample

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetElementName(&rc, newElement, data);
        cpiSetElementNamespace(&rc, newElement, data);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMESPACEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);
```

```
                 /* Add the element */
                 cpiAddAsLastChild(&rc, element, newElement);
```

# cpiSetElementType

This function sets the type of the specified syntax element.

## Syntax

```
void cpiSetElementType(
  int*           returnCode,
  CciElement*    targetElement,
  CciElementType type);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT

**targetElement**
Specifies the address of the target syntax element object (input).

**type**
The type to be set in the target element (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetElementName(&rc, newElement, data);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
```

# cpiSetElementValue

This optional function sets the value of a specified element.

It called by the broker when the value of a syntax element is to be set. It provides an opportunity for a user-defined parser to override the behavior for setting element values.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Optional | iFpSetElementValue |

## Syntax

```
void cpiSetElementValue(
  CciParser*      parser,
  CciElement*     currentElement,
  CciElementValue*  value);
```

## Parameters

**parser**
    The address of the parser object (input).

**currentElement**
    The address of the current syntax element (input).

**value**
    The value (input).

## Return values

None.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiSetElementValue(
  CciParser*      parser,
  CciElement*     element,
  CciElementValue* value
){
  CciElement* newElement;
  int         rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE))  {
    cpiSetElementValueValue(&rc, element, value);
  }
  else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
    /* Create a new value element, add as a first child, and set the value */
    newElement = cpiCreateElement(&rc, parser);
    cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_VALUE);
    cpiSetElementValueValue(&rc, newElement, value);
    cpiAddAsFirstChild(&rc, element, newElement);
  }
  else {
  }

  return;
}
```

# cpiSetElementValue group

This group of functions set the value of the specified syntax element.

Specify the appropriate function from this group that matches the type of data to
be set:

- cpiSetElementBitArrayValue
- cpiSetElementByteArrayValue
- cpiSetElementBooleanValue
- cpiSetElementCharacterValue
- cpiSetElementDateValue
- cpiSetElementDecimalValue
- cpiSetElementGmtTimestampValue
- cpiSetElementGmtTimeValue
- cpiSetElementIntegerValue
- cpiSetElementRealValue
- cpiSetElementTimestampValue
- cpiSetElementTimeValue

## Syntax

```
void cpiSetElementBitArrayValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciBitArray*  value);

void cpiSetElementByteArrayValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciByteArray*  value);

void cpiSetElementBooleanValue(
  int*        returnCode,
  CciElement*  targetElement,
  CciBool      value);

void cpiSetElementCharacterValue(
  int*         returnCode,
  CciElement*  targetElement,
  const CciChar*  value,
  CciSize      length);

void cpiSetElementDateValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciDate*  value);

void cpiSetElementDecimalValue(
  int*          returnCode,
  CciElement*  targetElement,
  const CciChar*  value);

void cpiSetElementGmtTimestampValue(
  int*                    returnCode,
  CciElement*             targetElement,  const struct CciTimestamp*  value);

void cpiSetElementGmtTimeValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTime*  value);

void cpiSetElementIntegerValue(
  int*        returnCode,
  CciElement*  targetElement,
  CciInt      value);

void cpiSetElementRealValue(
  int*        returnCode,
  CciElement*  targetElement,
  CciReal      value);
```

```
void cpiSetElementTimestampValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTimestamp*  value);
```

```
void cpiSetElementTimeValue(
  int*                    returnCode,
  CciElement*             targetElement,
  const struct CciTime*  value);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN

**targetElement**
Specifies the address of the target syntax element object (input).

**value**
The value to be set in the target element (input).

**length**
The length of the data value, expressed as the number of `CciChar` characters.
This parameter is used on relevant function calls only.

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

# cpiSetElementValueValue

This function sets the value of the specified syntax element.

## Syntax

```
void cpiSetElementValueValue(
  int*            returnCode,
  CciElement*     targetElement,
  CciElementValue*  value);
```

## Parameters

**returnCode**
Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER

**targetElement**
Specifies the address of the target syntax element object (input).

**value**

Specifies the address of the *CciElementValue* object that contains the value to be stored in the specified target element (input).

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiSetElementValue(
  CciParser*       parser,
  CciElement*      element,
  CciElementValue* value
){
  CciElement* newElement;
  int         rc;

  if ((cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_VALUE) ||
      (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME_VALUE))  {
    cpiSetElementValueValue(&rc, element, value);
  }
  else if (cpiElementType(&rc, element) == CCI_ELEMENT_TYPE_NAME) {
    /* Create a new value element, add as a first child, and set the value */
    newElement = cpiCreateElement(&rc, parser);
    cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_VALUE);
    cpiSetElementValueValue(&rc, newElement, value);
    cpiAddAsFirstChild(&rc, element, newElement);
  }
  else {
  }

  return;
}
```

# cpiSetNameFromBuffer

This function sets the name attribute of the target syntax element using the data supplied in the buffer pointed to by the **name** parameter. The size of the name is specified using the **length** parameter.

## Syntax

```
void cpiSetNameFromBuffer(
  int*            returnCode,
  CciElement*     targetElement,
  const CciChar*  name,
  CciSize         length);
```

## Parameters

**returnCode**

Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_EXCEPTION
- CCI_INV_ELEMENT_OBJECT
- CCI_INV_DATA_POINTER
- CCI_INV_DATA_BUFLEN

**targetElement**
Specifies the address of the target syntax element object (input).

**name**
The address of a buffer containing the name (input).

**length**
The length of the character string, expressed as the number of `CciChar` characters, specified by the name parameter.

## Return values

None. If an error occurs, *returnCode* indicates the reason for the error.

## Sample

```
/* Convert the attribute value into broker form */
        data = CciNString((char *)startMarker, markedSize, pc->iCcsid);

/* Create a new name-value element for the attribute */
        newElement = cpiCreateElement(&rc, parser);
        cpiSetElementType(&rc, newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
        cpiSetNameFromBuffer(&rc, newElement, data, length);
        if (pc->trace) {
          const char * mbData = mbString(data, pc->iCcsid);
          fprintf(pc->tracefile, "PLUGIN: Created new NAMEVALUE element;
                  object=0x%x type=0x%x name=",
                  newElement, CCI_ELEMENT_TYPE_NAME_VALUE);
          fprintf(pc->tracefile, "%s\n", mbData);
          fflush(pc->tracefile);
          free((void *)mbData);
        }
        /* Free the memory created in CciNString() */
        free((void *)data);

        /* Add the element */
        cpiAddAsLastChild(&rc, element, newElement);
```

# cpiSetNextParserClassName

This optional function returns the name of the next parser class in the chain.

It is called during finalize processing, and returns to the caller a string that contains the name of the next parser class in the chain. Using this information, a user-defined parser can, during the finalize phase, modify the syntax element tree before the phase that causes serialization of the bit stream.

If you specify the name of a parser supplied with WebSphere Message Broker, you must use the correct class name of the parser.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Optional | iFpSetNextParserClassName |

## Syntax

```
void cpiSetNextParserClassName(
  CciParser*   parser,
  CciContext*  context,
  CciChar*     name,
  CciBool      parserType);
```

## Parameters

**parser**
> The address of the parser object (input).

**context**
> The address of the context owned by the parser object (input).

**name**
> The name of the next parser as a string of **CciChar** characters.

**parserType**
> Indicates whether the referenced parser is *standard* (`parserType=0`) or *non-standard* (`parserType=non-zero`) (input). A standard parser expects the Format field of the preceding header in the chain to contain the name of the parser class that follows. Non-standard parsers expects the Domain field to contain the parser class name.

## Return values

None.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
void cpiSetNextParserClassName(
  CciParser*  parser,
  CciContext* context,
  CciChar*    name,
  CciBool     isHeaderParser
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                rc = 0;

  /* Save the name in my context */
  CciCharNCpy(pc->iNextParserClassName, name, CciCharLen(name));

  if (pc->trace) {
    fprintf(pc->tracefile, "PLUGIN: <- cpiSetNextParserClassName()\n");
    fflush(pc->tracefile);
  }

  return;
}
```

# cpiWriteBuffer

This function writes a syntax element tree to the message buffer associated with a parser.

It appends data to the bit stream in the message buffer associated with the parser object, using the current syntax element tree as a source. The element tree cannot be modified during the execution of this implementation function. The cpiAppendToBuffer utility function can be used to append the message buffer (bit stream) with data from the element tree.

If this implementation function is provided in the CPI_VFT structure, you cannot specify either cpiWriteBufferEncoded() or cpiWriteBufferFormatted(); if you do, the cpiDefineParserClass() function fails with a return code of CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
| --- | --- | --- |
| CPI_VFT | Conditional | iFpWriteBuffer |

## Syntax

```
int cpiWriteBuffer(
  CciParser*  parser,
  CciContext*  context);
```

## Parameters

**parser**
   The address of the parser object (input).

**context**
   The address of the context owned by the parser object (input).

## Return values

The size in bytes of the data appended to the bit stream in the buffer.

## Sample

```
int cpiWriteBuffer(
  CciParser*  parser,
  CciContext* context
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int             initialSize = 0;
  int             rc = 0;
  const void* a;
  CciByte b;


  initialSize = cpiBufferSize(&rc, parser);
  a = cpiBufferPointer(&rc, parser);
  b = cpiBufferByte(&rc, parser, 0);

  cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);

  return cpiBufferSize(0, parser) - initialSize;
}
```

# cpiWriteBufferEncoded

This function is an extension of the capability provided by the existing
**cpiWriteBuffer()** implementation function that provides the encoding and coded
character set that the output message should be represented in when the parser
serializes its element tree to an output bit stream.

If serialization is not required, for example when the output is based on an input
bit stream, and the tree has not been modified, this implementation function is not
called by the broker. If this implementation function is provided in the CPI_VFT
structure, you cannot specify either cpiWriteBuffer() or cpiWriteBufferFormatted();
if you do, the cpiDefineParserClass() function fails with a return code of
CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
| --- | --- | --- |
| CPI_VFT | Conditional | iFpWriteBufferEncoded |

## Syntax

```
int cpiWriteBufferEncoded(
  CciParser*    parser,
  CciContext*   context,
  int           encoding,
  int           ccsid);
```

## Parameters

**parser**
    The address of the parser object (input).

**context**
    The address of the context owned by the parser object (input).

**encoding**
    The encoding of the message buffer (input).

**ccsid**
    The CCSID of the message buffer (input).

## Return values

The size in bytes of the data appended to the bit stream in the buffer.

## Sample

This example is taken from the sample parser file `BipSampPluginParser.c`:

```
int cpiWriteBufferEncoded(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                initialSize = 0;
  int                rc = 0;
  const void* a;
  CciByte b;


  initialSize = cpiBufferSize(&rc, parser);
  a = cpiBufferPointer(&rc, parser);
  b = cpiBufferByte(&rc, parser, 0);

  cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);

  return cpiBufferSize(0, parser) - initialSize;
}
```

# cpiWriteBufferFormatted

This function extends the capability of the cpiWriteBuffer() implementation
function by providing additional information about the output message.

The following additional information is provided:
1. The encoding and coded character set that the output message is represented in
   when the parser serializes its element tree to an output bit stream.
2. The message set, type, and format for the output message for those parsers
   which require such information to correctly serialize its element tree to an
   output bit stream.

If serialization is not required, for example when the output is based on an input bit stream, and the tree has not been modified, this implementation function will not be invoked by the broker.

If this implementation function is provided in the CPI_VFT structure, you cannot specify either cpiWriteBuffer() or cpiWriteBufferEncoded(); if you do, the cpiDefineParserClass() function fails with a return code of CCI_INVALID_IMPL_FUNCTION.

| Defined In | Type | Member |
|---|---|---|
| CPI_VFT | Conditional | iFpWriteBufferFormatted |

## Syntax

```
int cpiWriteBufferFormatted(
  CciParser*    parser,
  CciContext*   context,
  int           encoding,
  int           ccsid,
  CciChar*      set,
  CciChar*      type,
  CciChar*      format);
```

## Parameters

**parser**
The address of the parser object (input).

**context**
The address of the context owned by the parser object (input).

**encoding**
The encoding of the message buffer (input).

**ccsid**
The CCSID of the message buffer (input).

**set**
The message set to which the message belongs (input).

**type**
The message type (input).

**format**
The message format (input).

## Return values

The size in bytes of the data appended to the bit stream in the buffer.

## Sample

```
int cpiWriteBufferFormatted(
  CciParser*  parser,
  CciContext* context,
  int         encoding,
  int         ccsid
  CciChar*      set,
  CciChar*      type,
  CciChar*      format
){
  PARSER_CONTEXT_ST* pc = (PARSER_CONTEXT_ST *)context ;
  int                initialSize = 0;
```

```
int              rc = 0;
const void* a;
CciByte b;


initialSize = cpiBufferSize(&rc, parser);
a = cpiBufferPointer(&rc, parser);
b = cpiBufferByte(&rc, parser, 0);

cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);

return cpiBufferSize(0, parser) - initialSize;
}
```

# C user exit API

The user exit API defines implementation and utility functions.

- A set of implementation functions provides the functionality of the user exits. Some of these implementation functions are mandatory and others are optional.

  These functions are defined in the **BipCci.h** header file. They are described in "C user exit implementation functions."

- A set of utility functions that are invoked by user exits.

  These functions are defined in the **BipCpi.h** header file. They are described in "C user exit utility functions" on page 259.

## C user exit implementation functions

A set of implementation functions provide the functionality of the user exits.

Some implementation functions are mandatory, and must be implemented by the developer; others are optional:

**Mandatory functions**

- "bipInitializeUserExits"
- "bipTerminateUserExits" on page 252

**Optional functions**

- "cciInputMessageCallback" on page 253
- "cciPropagatedMessageCallback" on page 255
- "cciOutputMessageCallback" on page 258
- "cciNodeCompletionCallback" on page 254
- "cciTransactionEventCallback" on page 257

### bipInitializeUserExits

bipInitializeUserExits is an implementation function that is exported by the User Exit library (.lel file).

This function is called when the execution group starts, just after loading the .lel file. When this function runs, your exit code must call cciRegisterUserExit to register each user exit provided by that .lel file.

**Syntax:**

```
void bipInitializeUserExits()
```

**Parameters:**

None.

**Return values:**

None.

**Example:**

```
extern "C"{

void bipInitializeUserExits(){

  int rc = CCI_SUCCESS;
  CCI_UE_VFT myVft = {CCI_UE_VFT_DEFAULT};
  myVft.iFpInputMessageCallback       = myInputMessageCallback;
  myVft.iFpTransactionEventCallback   = myTransactionEventCallback;
  myVft.iFpPropagatedMessageCallback  = myPropagatedMessageCallback;
  myVft.iFpNodeCompletionCallback     = myNodeCompletionCallback;

  cciRegisterUserExit(&rc,
                      MyConstants::myUserExitName,
                      0,
                      myVft);

  /*we should now check the rc for unexpected values*/

  return;
}

}/*end of extern "C" */
```

## bipTerminateUserExits

bipInitializeUserExits is an implementation function exported by the User Exit library (.lel file).

This function is called just before unloading the .lel file, which typically happens when the execution group process is stopping. When this function runs, your exit code must clean up all resources that were allocated during the bipInitializeUserExits function.

If this function is not exported, the .lel file fails to load. You cannot call other utility functions while bipTerminateUserExits is running. This function is started on the same thread as the bipInitializeUserExits function.

**Syntax:**

```
void bipTerminateUserExits()
```

**Parameters:**

None.

**Return values:**

None.

**Example:**

```
extern "C"{

void bipTerminateUserExits(){
  /*Here, we clean up any resources, e.g.
        spawned threads, file handles, sockets  */
```

```
    freeResources();
}

}/*end of extern "C" */
```

## cciInputMessageCallback

The cciInputMessageCallback function can be registered as a callback and is called every time a message is read by an input node, and before that message is propagated down the message flow.

The cciInputMessageCallback function is called for every input message that is read in the execution group where the callback is registered, if the user exit state is active. The callback is registered by providing a pointer to the function as the **iFpInputMessageCallback** field of the CCI_UE_VFT struct that is passed to **cciRegisterUserExit**.

**Syntax:**

```
typedef void (*cciInputMessageCallback)    (
                             CciDataContext*  userContext,
                             CciMessage*      message,
                             CciMessage*      localEnvironment,
                             CciMessage*      exceptionList,
                             CciMessage*      environment,
                             CciMessageOrigin messageOrigin,
                             CciNode*         inputNode);
```

**Parameters:**

**userContext (input)**
The value that is passed to the **cciRegisterUserExit** function.

**message**
A handle to the message object. The user exit code must not update transport headers or Properties elements in this tree. Updating the message can affect performance, particularly if the input message would not otherwise be changed in the message flow.

**localEnvironment**
A handle to the local environment object.

**exceptionList**
A handle to the exception list object.

**environment**
A handle to the environment object for the current message flow.

**messageOrigin**
Depending on the type of input node, the message might have originated from a bit stream (CCI_MESSAGE_ORIGIN_BITSTREAM) or from a tree (CCI_MESSAGE_ORIGIN_TREE). The user exit can access one of these sources without causing processing by the parser. For example, in the case of the MQInput node, you can access the bitstream safely whereas, in the case of the JMSInput node, you can access the tree safely. You can access the bit stream by calling **cniBufferPointer**, **cniBufferSize**, or **cniBufferByte**. You can access the tree by calling **cniRootElement** and using the usual syntax element navigation functions (for example, **cniFirstChild** ). Although this parameter tells the user exit what it can access safely without causing processing by the parser, the user exit code might ignore this advice and effectively alter the parse timing.

**inputNode**
> A handle to the input node that reads this input message. The handle can be used to make calls to functions such as **cciGetNodeName**, **cciGetNodeType**, and **cniGetBrokerInfo**.

**Return values:**

None.

**Example:**

```
void myInputMessageCallback(
                CciDataContext*  userContext,
                CciMessage*       message,
                CciMessage*       localEnvironment,
                CciMessage*       exceptionList,
                CciMessage*       environment,
                CciMessageOrigin messageOrigin,
                CciNode*          inputNode){
...
...
}
```

## cciNodeCompletionCallback

The cciNodeCompletionCallback function can be registered as a callback and is called whenever a node has completed processing of a message and is returning control to its upstream node.

The cciNodeCompletionCallback function is called for every message that is propagated in the execution group where the callback was registered, if the user exit state is active. The callback is registered by providing a pointer to the function as the **iFpNodeCompletionCallback** field of the CCI_UE_VFT struct that is passed to **cciRegisterUserExit**.

If the node completes due to an unhandled exception, it returns with a **reasonCode** of CCI_EXCEPTION, and that exception's details can be obtained by calling **cciGetLastExceptionData**.

If the node completes normally (including handling an exception on the catch or failure terminal), it returns with a **reasonCode** of CCI_SUCCESS. In this case, calling **cciGetLastExceptionData** returns unpredictable results.

**Syntax:**

```
typedef void (*cciNodeCompletionCallback)  (
                            CciDataContext* userContext,
                            CciMessage*     message,
                            CciMessage*      localEnvironment,
                            CciMessage*      exceptionList,
                            CciMessage*      environment,
                            CciConnection*  connection,
                            int             reasonCode);
```

**Parameters:**

**userContext (input)**
> The value that is passed to the **cciRegisterUserExit** function.

**message**
> A handle to the current message object. The current message is the message that is propagated to the node plus one or more of the following modifications:
> - Modifications that are applied to the input root in the node

- Modifications that are made from a user exit during the preceding propagate callback
- Modifications that are made from a user exit during the downstream node complete callback if the node does not create a new message; for example, output nodes, Compute nodes with a Compute Mode that is set to LocalEnvironment, Exception, or Exception And LocalEnvironment, or request nodes with an Output data location that is set to LocalEnvironment.

The user exit code must not update transport headers or Properties elements in the message tree. Updating the message can affect performance. Changes that are made during this callback are visible only if the upstream node does not cause a new Message to be created.

**localEnvironment**
A handle to the local environment object that is being propagated.

**exceptionList**
A handle to the exception list object that is being propagated.

**environment**
A handle to the environment object for the current message flow.

**connection**
A handle to the connection object between the two nodes. The handle can be used, for example, in calls to **cciGetSourceNode**, **cciGetTargetNode**, **cciGetSourceTerminalName**, and **cciGetTargetTerminalName**. This handle is valid only for the duration of this instance of the user exit function.

**reasonCode**
A reason code that indicates whether the node completes normally (CCI_SUCCESS) or the node completes as the result of an unhandled exception (CCI_EXCEPTION). If the node completes due to an unhandled exception, you can obtain that exception's details by calling **cciGetLastExceptionData**. If the node completes normally (including handling an exception on the catch or failure terminal), the effect of calling **cciGetLastExceptionData** is undetermined.

**Return values:**

None.

**Example:**

```
void myNodeCompletionCallback(
                CciDataContext* userContext,
                CciMessage*     message,
                CciMessage*     localEnvironment,
                CciMessage*     exceptionList,
                CciMessage*     environment,
                CciConnection*  connection
                int             reasonCode){
...
...
}
```

## cciPropagatedMessageCallback

The cciPropagatedMessageCallback function can be registered as a callback and is called whenever a message is propagated from one node to another.

The cciPropagatedMessageCallback function is called for every message that is propagated in the execution group where the callback was registered, if the user

exit state is active. The callback is registered by providing a pointer to the function as the **iFpPropagatedMessageCallback** field of the CCI_UE_VFT struct that is passed to **cciRegisterUserExit**.

**Syntax:**

```
typedef void (*cciPropagatedMessageCallback)(
                            CciDataContext* userContext,
                            CciMessage*     message,
                            CciMessage*     localEnvironment,
                            CciMessage*     exceptionList,
                            CciMessage*     environment,
                            CciConnection*  connection);
```

**Parameters:**

**userContext (input)**
> The value that is passed to the **cciRegisterUserExit** function.

**message**
> A handle to the message object that is being propagated. The user exit code must not update transport headers or Properties elements in this tree. Updating the message can affect performance, particularly if the input message would not otherwise be changed in the message flow.

**localEnvironment**
> A handle to the local environment object that is being propagated.

**exceptionList**
> A handle to the exception list object that is being propagated.

**environment**
> A handle to the environment object for the current message flow.

**connection**
> A handle to the connection object between the two nodes. The handle can be used, for example, in calls to **cciGetSourceNode**, **cciGetTargetNode**, **cciGetSourceTerminalName**, and **cciGetTargetTerminalName**. This handle is valid only for the duration of this instance of the user exit function.

**Return values:**

None.

**Example:**

```
void myPropagatedMessageCallback(
                        CciMessage*     message,
                        CciMessage*     localEnvironment,
                        CciMessage*     exceptionList,
                        CciMessage*     environment,
                        CciConnection* connection){


    int rc = CCI_SUCCESS;
    CciNode* targetNode = cciGetTargetNode(amp rc,
                                    connection);



    CciChar targetNodeName [initialStringBufferLength];
    targetNodeNameLength = cciGetNodeName(amp rc,
                                    targetNode,
                                    targetNodeName,
                                    initialStringBufferLength);
```

```
    /*you should now check the rc for unexpected values*/
    /*if rc is CCI_BUFFER_TOO_SMALL, resize and retry*/

}
```

## cciTransactionEventCallback

The cciTransactionEventCallback function can be registered as a callback and is called every time a message flow transaction ends.

This function is called for every message flow transaction within the execution group where the callback was registered, if the user exit state is active. The callback is registered by providing a pointer to the function in the **iFpTransactionEventCallback** field of the CCI_UE_VFT struct passed to **cciRegisterUserExit**.

**Syntax:**

```
typedef void (*cciTransactionEventCallback) (
                                CciDataContext*        userContext,
                                CciTransactionEventType type,
                                CciMessage*            environment,
                                CciNode*               inputNode);
```

**Parameters:**

**userContext (input)**
    This is the value that was passed to the **cciRegisterUserExit** function.

**type**
    This describes the event that occurred. Possible values are:
    - CCI_TRANSACTION_EVENT_COMMIT

        A transaction has been successfully committed.
    - CCI_TRANSACTION_EVENT_ROLLBACK

        A transaction has been rolled back.

    If the transaction was rolled back due to an unhandled exception, you can obtain details of that exception by calling **cciGetLastExceptionData**.

**environment**
    This is a handle to the environment object for the current message flow. Although the user exit can update this tree, it is cleared after returning from this function, so any updates are lost.

**inputNode**
    This is a handle to the input node which reads the input message that triggered the transaction. It can be used to make calls to functions such as **cciGetNodeName**, **cciGetNodeType**, and **cniGetBrokerInfo**.

**Return values:**
None

**Example:**

```
void myTransactionEventCallback(
                CciDataContext*        userContext,
                CciTransactionEventType type,
                CciMessage*            environment,
                CciNode*               inputNode){
...
...
}
```

## cciOutputMessageCallback

The cciOutputMessageCallback function can be registered as a callback and is called whenever a message is sent by an output node.

The cciOutputMessageCallback function is called for every output message that is sent successfully in the execution group or message flow where the callback was registered if the user exit state is active. If the node provides WrittenDestination information in the LocalEnvironment tree, the callback is called after this information is created.

Calls are made after the following operations are completed:
- Sending a message from an output or reply node (for WebSphere MQ, JMS, TCPIP, HTTP, or SOAP nodes).
- Sending a message from a request node (TCPIP, HTTP, or SOAP nodes). The callback is made after the reply has been received.
- Writing to a file (FileOutput node).
- Sending an e-mail (EmailOutput node).
- Completing an adapter request (WebSphere Adapters request nodes).

The callback is registered by providing a pointer to the function as the iFpOutputMessageCallback field of the CCI_UE_VFT struct that is passed to cciRegisterUserExit. The iFpOutputMessageCallback field was added in CCI_UE_VFT struct version 2.

When you implement this callback, check the node type by using cciGetNodeType before you perform any node-specific operations.

**Syntax:**

```
typedef void (*cciOutputMessageCallback)     (
                               CciDataContext*  userContext,
                               CciMessage*      message,
                               CciMessage*      localEnvironment,
                               CciMessage*      exceptionList,
                               CciMessage*      environment,
                               CciNode*         node);
```

**Parameters:**

**userContext (input)**
   The value that is passed to the cciRegisterUserExit function.

**message (input)**
   A handle to the message object. You must not update the transport headers or Properties elements in this tree.

   You can update the message body. However, because this callback is called after the message has been sent to the transport, any changes do not appear in that message. Changes are visible only to nodes that are connected downstream of the output node. Updating the message can affect performance, particularly if the message tree would not otherwise be changed in the message flow.

**localEnvironment (input)**
   A handle to the local environment object. The handle can contain information about the destination of the message that is written in the WrittenDestination subtree. See each node's documentation for more details.

**exceptionList (input)**
   A handle to the exception list object.

**environment (input)**

A handle to the environment object for the current message flow.

**node (input)**

A handle to the node that has sent the output message. You can use the handle to make calls to functions such as cciGetNodeName, cciGetNodeType, and cniGetBrokerInfo.

**Return values:**

None.

**Example:**

```
void myOutputMessageCallback(
                  CciDataContext*  userContext,
                  CciMessage*      message,
                  CciMessage*      localEnvironment,
                  CciMessage*      exceptionList,
                  CciMessage*      environment,
                  CciNode*         node){
}
```

## C user exit utility functions

The broker provides several utility functions that you can call from your user exits.

- "cciGetNodeAttribute"
- "cciGetNodeName" on page 260
- "cciGetNodeType" on page 261
- "cciGetSourceNode" on page 264
- "cciGetSourceTerminalName" on page 265
- "cciGetTargetNode" on page 266
- "cciGetTargetTerminalName" on page 266
- "cciRegisterUserExit" on page 267

### cciGetNodeAttribute

The cciGetNodeAttribute function returns the value of the specified attribute.

**Syntax:**

```
CciSize  cciGetNodeAttribute (int*             returnCode,
                        CciNode*      node,
                        CciChar*      name,
                        CciChar*      value,
                        CciSize       length);
```

**Parameters:**

**returnCode (output)**

Receives the return code from the function (output).

- CCI_INV_BUFFER_TOO_SMALL

  The provided buffer was not large enough to hold the value of node's type.

**node (input)**

This is a handle to a node.

**name (input)**

This is a pointer to a NULL-terminated string of CciChar specifying the name of the node attribute being queried.

**value (output)**
    Address of a buffer, allocated by the caller to hold the value of the attribute.

**length**
    The length, in CciChars, of the buffer allocated by the caller.

**Return values:**
- If successful, the attribute value is copied into the supplied buffer and the number of CciChar characters copied is returned.
- If the buffer is not large enough to contain the attribute value, **returnCode** is set to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is returned.
- If **name** specifies an attribute name that is not appropriate for the given node, **returnCode** is set to CCI_ATTRIBUTE_UNKOWN.

**Example:**

```
void myPropagatedMessageCallback(
                     CciMessage*    message,
                     CciMessage*    localEnvironment,
                     CciMessage*    exceptionList,
                     CciMessage*    environment,
                     CciConnection* connection){
   int rc = CCI_SUCCESS;
   CciNode* sourceNode = cciGetSourceNode(&rc,
                                     connection);
   /*you should now check the rc for unexpected values*/
   CciChar   queueNameAttribute[16];
   cciMbsToUcs(&rc,
            "queueName",
            queueNameAttribute,
            16,
            BIP_DEF_COMP_CCSID);
 /*you should now check the rc for unexpected values*/

   CciChar queueName [512];
   sourceNodeQueueNameLength = cciGetNodeType(&rc,
                                     sourceNode,
                                     queueName,
                                     512);
   /*you should now check the rc for unexpected values*/
   /*if rc is CCI_BUFFER_TOO_SMALL, resize and retry*/
 /*sourceNodeQueueNameLength will hold the actual or required size */
```

## cciGetNodeName

This function returns the name of the specified node.

The name is assigned internally by the Message Broker Toolkit and is unique in the message flow. The label that is assigned to a node by the message flow designer in the Message Flow Editor can be obtained by calling "cciGetNodeAttribute" on page 259 to read the label attribute.

**Syntax:**

```
CciSize  getNodeName (int*                 returnCode,
                      CciNode*         node,
                      CciChar*         value,
                      CciSize          length);
```

**Parameters:**

**returnCode (output)**
    Receives the return code from the function (output)

- CCI_INV_BUFFER_TOO_SMALL

    The provided buffer was not large enough to hold the value of node's name.

**node (input)**
    This is a handle to a node.

**value (output)**
    Address of a buffer, allocated by the caller to hold the value of the node's name.

**length**
    The length, in CciChars, of the buffer allocated by the caller.

**Return values:**

- If successful, the node name is copied into the supplied buffer and the number of CciChar characters copied is returned.
- If the buffer is not large enough to contain the node name, **returnCode** is set to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is returned.

**Example:**

```
void myPropagatedMessageCallback(
                    CciMessage*    message,
                    CciMessage*    localEnvironment,
                    CciMessage*    exceptionList,
                    CciMessage*    environment,
                    CciConnection* connection){

  int rc = CCI_SUCCESS;
  CciNode* targetNode = cciGetTargetNode(&rc,
                                   connection);



  CciChar targetNodeName [initialStringBufferLength];
  targetNodeNameLength = cciGetNodeName(&rc,
                                   targetNode,
                                   targetNodeName,
                                   initialStringBufferLength);
  /*you should now check the rc for unexpected values*/
  /*if rc is CCI_BUFFER_TOO_SMALL, resize and retry*/

}
```

## cciGetNodeType
This function returns the type of the specified node.

**Syntax:**

```
CciSize  cciGetNodeType (int*                     returnCode,
                         CciNode*        node,
                         CciChar*        value,
                         CciSize         length);
```

**Parameters:**

**returnCode (output)**
    Receives the return code from the function (output).

- CCI_INV_BUFFER_TOO_SMALL

    The provided buffer was not large enough to hold the value of node's type.

**node (input)**
    This is a handle to a node.

**value (output)**
Address of a buffer, allocated by the caller to hold the value of the node type.

**length**
The length, in CciChars, of the buffer allocated by the caller.

**Return values:**
- If the function is successful, the node type is copied into the supplied buffer and the number of CciChar characters copied is returned.
- If the buffer is not large enough to contain the node type, **returnCode** is set to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is returned.

**Example:**

```
void myPropagatedMessageCallback(
                   CciMessage*    message,
                   CciMessage*    localEnvironment,
                   CciMessage*    exceptionList,
                   CciMessage*    environment,
                   CciConnection* connection){

  int rc = CCI_SUCCESS;
  CciNode* sourceNode = cciGetSourceNode(&rc,
                                    connection);
  /*you should now check the rc for unexpected values*/

  CciChar sourceNodeType[initialStringBufferLength];
  sourceNodeTypeLength = cciGetNodeType(&rc,
                                 sourceNode,
                                 sourceNodeType,
                                 initialStringBufferLength);
  /*you should now check the rc for unexpected values*/
  /*if rc is CCI_BUFFER_TOO_SMALL then you should resize and retry*/
```

**Node types:**

Use the cciGetNodeType function to find out the node type of any node.

| Node | Node Type |
|---|---|
| **WebSphere MQ** | |
| MQInput | ComIbmMQInputNode |
| MQOutput | ComIbmMQOutputNode |
| MQReply | ComIbmMQOutputNode |
| MQGet | ComIbmMQGetNode |
| MQOptimizedFlow | ComIbmMQInputNode |
| **JMS** | |
| JMSInput | ComIbmJMSClientInputNode |
| JMSOutput | ComIbmJMSClientOutputNode |
| JMSReply | ComIbmJMSClientReplyNode |
| **HTTP** | |
| HTTPInput | ComIbmWSInputNode |
| HTTPReply | ComIbmWSReplyNode |
| HTTPRequest | ComIbmWSRequestNode |
| **Web Services** | |
| SOAPInput | ComIbmSOAPInputNode |

| Node | Node Type |
|---|---|
| SOAPReply | ComIbmSOAPReplyNode |
| SOAPRequest | ComIbmSOAPRequestNode |
| SOAPAsyncRequest | ComIbmSOAPAsyncRequestNode |
| SOAPAsyncResponse | ComIbmSOAPAsyncResponseNode |
| SOAPEnvelope | ComIbmSOAPWrapperNode |
| SOAPExtract | ComIbmSOAPExtractNode |
| RegistryLookup | SRRetrieveEntityNode |
| EndpointLookup | SRRetrieveITServiceNode |
| **WebSphere Adapters** | |
| PeopleSoftInput | ComIbmPeopleSoftInputNode |
| PeopleSoftRequest | ComIbmPeopleSoftRequestNode |
| SAPInput | ComIbmSAPInputNode |
| SAPRequest | ComIbmSAPRequestNode |
| SiebelInput | ComIbmSiebelInputNode |
| SiebelRequest | ComIbmSiebelRequestNode |
| TwineballInput | ComIbmTwineBallInputnode |
| TwineballRequest | ComIbmTwineBallRequestNode |
| **Routing** | |
| Filter | ComIbmFilterNode |
| Label | ComIbmLabelNode |
| Publication | The subflow is composed of:<br><br>ComIbmPSServiceNode + ComIbmMQOutputNode + ComIbmMQeOutputNode + ComIbmSCADAOutputNode<br><br>ComIbmPSServiceNode + ComIbmMQOutputNode |
| RouteToLabel | ComIbmRouteToLabelNode |
| Route | ComIbmRouteNode |
| AggregateControl | ComIbmAggregateControlNode |
| AggregateReply | ComIbmAggregateReplyNode |
| AggregateRequest | ComIbmAggregateRequestNode |
| Collector | ComIbmCollectorNode |
| **Transformation** | |
| Mapping | ComIbmComputeNode |
| XSLTransform | ComIbmXslMqsiNode |
| Compute | ComIbmComputeNode |
| JavaCompute | ComIbmJavaComputeNode |
| PHPCompute | ComIbmPhpCompute |
| **Construction** | |
| Throw | ComIbmThrowNode |

| Node | Node Type |
|------|-----------|
| Trace | ComIbmTraceNode |
| TryCatch | ComIbmTryCatchNode |
| FlowOrder | ComIbmFlowOrderNode |
| Passthrough | ComIbmPassthruNode |
| ResetContentDescriptor | ComIbmResetContentDescriptorNode |
| **Database** | |
| Database | ComIbmDatabaseNode |
| DataDelete | ComIbmDatabaseNode |
| DataInsert | ComIbmDatabaseNode |
| DataUpdate | ComIbmDatabaseNode |
| Warehouse | ComIbmDatabaseNode |
| DatabaseRetrieve | ComIbmDatabaseRetrieveNode |
| DatabaseRoute | ComIbmDatabaseRouteNode |
| **File** | |
| FileInput | ComIbmFileInputNode |
| FileOutput | ComIbmFileOutputNode |
| **Email** | |
| EmailOutput | ComIbmEmailOutputNode |
| **TCPIP** | |
| TCPIPClientInput | ComIbmTCPIPClientInputNode |
| TCPIPClientOutput | ComIbmTCPIPClientOutputNode |
| TCPIPClientReceive | ComIbmTCPIPClientRequestNode |
| TCPIPServerInput | ComIbmTCPIPServerInputNode |
| TCPIPServerOutput | ComIbmTCPIPServerOutputNode |
| TCPIPServerReceive | ComIbmTCPIPServerRequestNode |
| **IMS**™ | |
| IMSRequest | ComIbmIMSRequestNode |
| **Validation** | |
| Validate | ComIbmValidateNode |
| **Timer** | |
| TimeoutControl | ComIbmTimeoutControlNode |
| TimeoutNotification | ComIbmTimeoutNotificationNode |
| **Additional protocols** | |
| SCADAInput | ComIbmSCADAInputNode |
| SCADAOutput | ComIbmSCADAOutputNode |

## cciGetSourceNode

This function returns a handle to the upstream node of a given connection.

**Syntax:**

```
CciNode*  cciGetSourceNode(int*                      returnCode,
                           CciConnection * connection);
```

**Parameters:**

**returnCode (output)**
> Receives the return code from the function.

**connection**
> This parameter is a handle to a connection on the output terminal of the requested node.

**Return values:**
A handle to the node that is on the source side of the connection.

**Example:**

```
void myPropagatedMessageCallback(
                    CciMessage*    message,
                    CciMessage*    localEnvironment,
                    CciMessage*    exceptionList,
                    CciMessage*    environment,
                    CciConnection* connection){
   ...
   ...

   int rc = CCI_SUCCESS;
   CciNode* sourceNode = cciGetSourceNode(&rc,
                                connection);
   /*you should now check the rc for unexpected values*/
```

## cciGetSourceTerminalName

This function returns the name of the output terminal of the source node for the specified connection.

**Syntax:**

```
CciSize  cciGetSourceTerminalName (int*           returnCode,
                                   CciConnection* connection,
                                   CciChar*       value,
                                   CciSize        length);
```

**Parameters:**

**returnCode (output)**
> Receives the return code from the function (output).
>
> - CCI_BUFFER_TOO_SMALL
>
>   The provided buffer was not large enough to hold the value of node's name.

**connection (input)**
> This parameter is a handle to a connection between two nodes.

**value (output)**
> Address of a buffer, allocated by the caller to hold the value of the terminal's name.

**length**
> The length, in `CciChars`, of the buffer allocated by the caller.

**Return values:**
- If successful, the terminal name is copied into the supplied buffer, and the number of `CciChar` characters copied is returned.
- If the buffer is not large enough to contain the attribute value, **returnCode** is set to CCI_BUFFER_TOO_SMALL, and the number of `CciChars` required is returned.

**Example:**

```
void myPropagatedMessageCallback(
                      CciDataContext* userContext,
                      CciMessage*     message,
                      CciMessage*     localEnvironment,
                      CciMessage*     exceptionList,
                      CciMessage*     environment,
                      CciConnection*  connection){
  int rc = CCI_SUCCESS;
  CciChar sourceTerminalName[initialStringBufferLength];
  cciGetSourceTerminalName(&rc,
                      connection,
                      sourceTerminalName,
                      initialStringBufferLength);
}
```

## cciGetTargetNode

This function returns a handle to the downstream node of a given connection.

**Syntax:**

```
CciNode*  cciGetTargetNode(int*                    returnCode,
                           CciConnection * connection);
```

**Parameters:**

**returnCode (output)**
> Receives the return code from the function (output).

**connection**
> This parameter is a handle to a connection on an input terminal of the
> requested node.

**Return values:**
A handle to the node that is on the target side of the connection.

**Example:**

```
void myPropagatedMessageCallback(
                      CciMessage*     message,
                      CciMessage*     localEnvironment,
                      CciMessage*     exceptionList,
                      CciMessage*     environment,
                      CciConnection* connection){
  ...
  ...

  CciNode* targetNode = cciGetTargetNode(&rc,
                                connection);
```

## cciGetTargetTerminalName

This function returns the name of the input terminal of the target node for the
specified connection.

**Syntax:**

```
CciSize  cciGetTargetTerminalName (int*          returnCode,
                                   CciConnection* connection,
                                   CciChar*       value,
                                   CciSize        length);
```

**Parameters:**

**returnCode (output)**
> Receives the return code from the function (output).

- CCI_BUFFER_TOO_SMALL

    The provided buffer was not large enough to hold the value of node name.

**connection (input)**

    This parameter is a handle to a connection between two nodes.

**value (output)**

    Address of a buffer, allocated by the caller to hold the value of the terminal's name.

**length**

    The length, in CciChars, of the buffer allocated by the caller.

**Return values:**

- If successful, the terminal name is copied into the supplied buffer, and the number of CciChar characters copied is returned.
- If the buffer is not large enough to contain the terminal name, **returnCode** is set to CCI_BUFFER_TOO_SMALL, and the number of CciChars required is returned.

**Example:**

```
void myPropagatedMessageCallback(
                    CciDataContext* userContext,
                    CciMessage*     message,
                    CciMessage*     localEnvironment,
                    CciMessage*     exceptionList,
                    CciMessage*     environment,
                    CciConnection*  connection){
  int rc = CCI_SUCCESS;
  CciChar targetTerminalName[initialStringBufferLength];
  cciGetTargetTerminalName(&rc,
                    connection,
                    targetTerminalName,
                    initialStringBufferLength);
  /*you should now check the rc for unexpected values*/
  /*if rc is CCI_BUFFER_TOO_SMALL then you should resize and retry*/
```

### cciRegisterUserExit

cciRegisterUserExit is a utility function that can be called by the user's code when bipInitializeUserExits is called.

This function is called by the user's code if the user wants to register functions to be called every time certain events occur.

**Syntax:**

```
typedef struct cci_UEVft {
    int     reserved;
    char    StrucId[4];
    int     Version;
    cciInputMessageCallback       iFpInputMessageCallback;
    cciTransactionEventCallback   iFpTransactionEventCallback;
    cciPropagatedMessageCallback  iFpPropagatedMessageCallback;
    cciNodeCompletionCallback     iFpNodeCompletionCallback;
    cciOutputMessageCallback      iFpOutputMessageCallback;

} CCI_UE_VFT;

void cciRegisterUserExit (
  int*                           returnCode,
  CciChar*                       name,
  CciDataContext*                userContext,
  CCI_UE_VFT*                    functionTable);
```

**Parameters:**

**returnCode (output)**

Requires the return code from the function. Possible values are:

- CCI_DUP_USER_EXIT_NAME

  The specified name matches the name of a user exit previously registered in the current execution group.

- CCI_INV_USER_EXIT_NAME

  The specified name was invalid. This can be caused if a NULL pointer, empty string or a string containing non-alphanumeric characters was specified.

**Name (input)**

This parameter must contain a pointer to a NULL-terminated string of CciChars specifying a name for the user exit. The name must be unique across all user exits that can be installed on the same broker. This name is used to identify the user exit in, for example:

- User Trace messages
- Exceptions or syslog messages
- Administration commands (for example, mqsichangeflowuserexits)

The name has the following restrictions:

- It must consist of alphanumeric characters only.
- It must be no greater than 255 characters.
- The name must be unique across all user exits that can be installed on the same broker.

**userContext (input)**

This parameter allows the caller to provide a context pointer that is passed to the callback function when it is called. This parameter can be NULL.

**functionTable (input)**

This parameter is a pointer to a struct whose fields must contain either pointers to functions matching the correct signatures or contain NULL. A NULL value for any of these fields indicates that the user exit must not be called for that event.

Initialize the structure by using the define CCI_UE_VFT_DEFAULT, which sets the version as CCI_UE_VFT_CURRENT_VERSION. The cciOutputMessageCallback was added at version 2, CCI_UE_VFT_VERSION_2.

**Return values:**

None. If an error occurs, the **returnCode** parameter indicates the reason for the error.

**Example:**

```
extern "C"{

void bipInitializeUserExits(){

  int rc = CCI_SUCCESS;
  CCI_UE_VFT myVft = {CCI_UE_VFT_DEFAULT};
  myVft.iFpInputMessageCallback       = myInputMessageCallback;
  myVft.iFpTransactionEventCallback   = myTransactionEventCallback;
  myVft.iFpPropagatedMessageCallback  = myPropagatedMessageCallback;
  myVft.iFpNodeCompletionCallback     = myNodeCompletionCallback;
  myVft.iFpOutputMessageCallback      = myOutputMessageCallback;
```

```
cciRegisterUserExit(&rc,
                    MyConstants::myUserExitName,
                    0,
                    &myVft);

/*you should now check the rc for unexpected values*/

return;
}

}/*end of extern "C" */
```

# C common API

The C language common API consists of implementation and utility functions that you can use in user-defined nodes, parsers, and user exits.

All these functions are defined in the `BipCpi.h` header file, see "C Header files" on page 123.

- "C common implementation functions."
- "C common utility functions" on page 271.

## C common implementation functions

You can use the common implementation functions in user-defined nodes and user-defined parsers. All the functions are called by the broker on occurrence of certain events.

These functions are defined in the `BipCci.h` header file.

### Optional functions

- cciRegCallback

### cciRegCallback

This function can be registered as a callback function, and is called when the registered event occurs. The function is registered by providing a function pointer which matches a particular typedef.

**Syntax:**

```
typedef int (*CciRegCallback)(CciDataContext *, cciCallbackType);
```

**Parameters:**

**type CciDataContext***
   This parameter is the pointer that is provided by the caller to the registration function.

**type CciCallbackType**
   This parameter indicates the reason for the callback. The reason is always one of the CciCallbackType values that is specified on the registration call corresponding to this callback.

**Return values:**

**type CciRegCallbackStatus** (defined in BipCci.h)

- CCI_THREAD_STATE_REGISTRATION_RETAIN: This return code is used for a callback that is to remain registered as a callback function on a particular thread.

- CCI_THREAD_STATE_REGISTRATION_REMOVE: This return code is used to signify that the callback is to be de-registered, and that it must not be called again on this thread unless it is reregistered.
- If another value is returned, a warning is written to a log, and CCI_THREAD_STATE_REGISTRATION_RETAIN is assumed.

During execution of this function, it is possible that the node or parser that has registered the function has already been deleted. Therefore, you must not call a node or parser utility function that depends on the existence of a node or parser. The only utility functions that can be called from this callback are:
- cciLog
- cciUserTrace
- cciServiceTrace
- cciUserDebugTrace
- cciServiceDebugTrace
- cciIsTraceActive

For each of these five trace utility functions, the CciObject parameter must be NULL.

**Example:**

Declare the following struct and function:

```
typedef struct {
    int        id;
}MyContext;

static int registered=0;

CciRegCallbackStatus switchThreadStateChange(CciDataContext *context, CciCallbackType type)
{
  char traceText[256];
  char* typeStr=0;
  MyContext* myContext = (MyContext*)context;

  if (type==CCI_THREAD_STATE_IDLE){
      typeStr = "idle";
  }else if(type==CCI_THREAD_STATE_INSTANCE_END){
      typeStr = "instance end";
  }else if (type==CCI_THREAD_STATE_TERMINATION){
      typeStr = "termination";
  }else{
      typeStr = "unknown";
  }

  memset(traceText,0,256);
  sprintf(traceText,"switchThreadStateChange: context id = %d, thread state %s",myContext->id,typeStr);
  cciServiceTrace(NULL,
                  NULL,
                  traceText);
  return CCI_THREAD_STATE_REGISTRATION_RETAIN;

}
```

Place the following code into the _Switch_evaluate function in the samples to enable you to read service trace, and see when the message processing thread changes state:

```
/*register for thread state change*/
  CciMessageContext* messageContext = cniGetMessageContext(NULL,message);
  CciThreadContext*  threadContext  = cniGetThreadContext(NULL,messageContext);

  static MyContext myContext={1};

  if(registered==0){
    cciRegisterForThreadStateChange(
                  NULL,
                  threadContext,
                  & myContext,
                  switchThreadStateChange,
                  CCI_THREAD_STATE_IDLE |
                  CCI_THREAD_STATE_INSTANCE_END |
                  CCI_THREAD_STATE_TERMINATION);

    registered=1;

  }
```

This example registers only on the first thread that receives a message. If it is necessary to register every thread that receives a message, the user-defined extensions must remember on which threads they have registered.

By using the userContext parameter you can see how data is passed from the code where the callback is registered to the actual callback function.

When registering the callback, a pointer to an instance of the **MyContext** struct is passed in. This pointer is the pointer that is passed back to the callback. To ensure that the pointer is still valid when it is passed back to the callback, an instance of the struct is declared as static. Another technique to ensure that the pointer is valid is to allocate storage on the heap.

In the callback function, the userContext parameter can be cast to a **(MyContext*)**. The original MyContext struct can be referenced through this address. This technique permits the passing of data from the code where the callback is registered to the callback function.

## C common utility functions

WebSphere Message Broker provides some additional utilities that user-defined nodes and parsers can use.

These utilities belong to the following categories:
- Exception handling and logging
- Character representation handling

These functions are defined in the BipCci.h header file.

The following exception handling and logging functions are provided for use by a user-defined node or parser:
- "cciGetLastExceptionData" on page 274
- "cciGetLastExceptionDataW" on page 276
- "cciLog" on page 277
- "cciLogW" on page 278
- "cciRethrowLastException" on page 283
- "cciThrowException" on page 287

- "cciThrowExceptionW" on page 289

The following utilities help you convert between the processing code (in UCS-2) that is used internally by WebSphere Message Broker, and file codes (for example, ASCII).
- "cciMbsToUcs" on page 280
- "cciUcsToMbs" on page 292

The following utility functions enable you to determine whether trace is active, and write entries that are appropriate for the trace settings.
- "cciIsTraceActive" on page 290
- "cciUserTrace" on page 296
- "cciUserTraceW" on page 298
- "cciUserDebugTrace" on page 293
- "cciUserDebugTraceW" on page 295
- "cciServiceTrace" on page 285
- "cciServiceTraceW" on page 286
- "cciServiceDebugTrace" on page 283
- "cciServiceDebugTraceW" on page 284

The following utility function is used to register a function that is to be called when the current thread enters a particular state:
- "cciRegisterForThreadStateChange" on page 281

The following utility functions are available for use with user exits:
- "cciGetBrokerInfo"
- "cciGetNodeAttribute" on page 259
- "cciGetNodeName" on page 260
- "cciGetNodeType" on page 261
- "cciGetSourceNode" on page 264
- "cciGetSourceTerminalName" on page 265
- "cciGetTargetNode" on page 266
- "cciGetTargetTerminalName" on page 266
- "cciInputMessageCallback" on page 253
- "cciNodeCompletionCallback" on page 254
- "cciPropagatedMessageCallback" on page 255
- "cciRegisterUserExit" on page 267
- "cciTransactionEventCallback" on page 257

## cciGetBrokerInfo

Use the cciGetBrokerInfo function to query the current broker environment (for example, for information about broker name, execution group name, queue manager name). The information is returned in a structure of type CCI_BROKER_INFO_ST.

This function differs from cniGetBrokerInfo in that you do not have to specify a CciNode* handle, and that it does not return information about a message flow. Therefore, you can call cciGetBrokerInfo from initialization functions; for example, bipInitializeUserExits, bipGetMessageParserFactory, and bipGetMessageFlowNodeFactory.

**Syntax:**

```
void cciGetBrokerInfo(
                     int*               returnCode,
                     CCI_BROKER_INFO_ST* broker_info_st);
```

**Parameters:**

**returnCode (output)**

Receives the return code from the function (output).

Possible return codes are:
- CCI_SUCCESS
- CCI_INV_BROKER_INFO_ST
- CCI_EXCEPTION

**broker_info_st (output)**

The address of a CCI_BROKER_INFO_ST structure to be populated with the relevant values on successful completion:

```
typedef struct cci_broker_info_st {
int versionId;    /*Structure version identification*/
CCI_STRING_ST brokerName;   /*The label of the broker*/
CCI_STRING_ST executionGroupName; /*The label of the current execution group*/
CCI_STRING_ST queueManagerName;  /*The name of the MQ Queue Manager for the broker*/
CCI_STRING_ST dataSourceUserId;  /*The userid broker connects to datasource as*/
} CCI_BROKER_INFO_ST;
```

**Return values:**

None. If an error occurs, the **returnCode** parameter indicates the reason for the error.

**Example:**

```
int rc = CCI_SUCCESS;

CCI_BROKER_INFO_ST brokerInfo = {CCI_BROKER_INFO_ST_DEFAULT};

#define INTITIAL_STR_LEN 256
CciChar brokerNameStr[INTITIAL_STR_LEN];
CciChar executionGroupNameStr[INTITIAL_STR_LEN];
CciChar queueManagerNameStr[INTITIAL_STR_LEN];

brokerInfo.brokerName.bufferLength = INTITIAL_STR_LEN;
brokerInfo.brokerName.buffer       = brokerNameStr;

brokerInfo.executionGroupName.bufferLength = INTITIAL_STR_LEN;
brokerInfo.executionGroupName.buffer = executionGroupNameStr;

brokerInfo.queueManagerName.bufferLength = INTITIAL_STR_LEN;
brokerInfo.queueManagerName.buffer = queueManagerNameStr;

cciGetBrokerInfo(&rc,&brokerInfo);

/* just in case any of the buffers were too short*/
if ((brokerInfo.brokerName.bytesOutput        < brokerInfo.brokerName.dataLength)  ||
    (brokerInfo.executionGroupName.bytesOutput < brokerInfo.executionGroupName.dataLength) ||
    (brokerInfo.queueManagerName.bytesOutput   < brokerInfo.queueManagerName.dataLength)) {

  /*at least one of the buffer were too short, need to rerty*/
  /* NOTE this is unlikely given that the initial sizes were reasonably large*/

  brokerInfo.brokerName.bufferLength =
     brokerInfo.brokerName.dataLength;
  brokerInfo.brokerName.buffer       =
```

```
    (CciChar*)malloc (brokerInfo.brokerName.bufferLength * sizeof(CciChar));

  brokerInfo.executionGroupName.bufferLength =
    brokerInfo.executionGroupName.dataLength;
  brokerInfo.executionGroupName.buffer         =
    (CciChar*)malloc (brokerInfo.executionGroupName.bufferLength * sizeof(CciChar));

  brokerInfo.queueManagerName.bufferLength =
    brokerInfo.queueManagerName.dataLength;
  brokerInfo.queueManagerName.buffer        =
    (CciChar*)malloc (brokerInfo.queueManagerName.bufferLength * sizeof(CciChar));

  cciGetBrokerInfo(&rc,&brokerInfo);

  /*now do something sensible with these strings before the buffers go out of scope*/
  /* for example call a user written function to copy them away*/
  copyBrokerInfo(brokerInfo.brokerName.buffer,
                 brokerInfo.executionGroupName.buffer,
                 brokerInfo.queueManagerName.buffer);

  free((void*)brokerInfo.brokerName.buffer);
  free((void*)brokerInfo.executionGroupName.buffer);
  free((void*)brokerInfo.queueManagerName.buffer);

}else{
  /*now do something sensible with these strings before the buffers go out of scope*/
  /* for example call a user written function to copy them away*/
  copyBrokerInfo(brokerInfo.brokerName.buffer,
                 brokerInfo.executionGroupName.buffer,
                 brokerInfo.queueManagerName.buffer);
}
```

## cciGetLastExceptionData

Gets diagnostic information about the last exception generated. Information about the last exception generated on the current thread is returned in a CCI_EXCEPTION_ST output structure. The user-defined extension can use this function to determine whether any recovery is required when a utility function returns an error code.

You can call this function when a utility function or user exit callback indicates that an exception has occurred, by setting *returnCode* to CCI_EXCEPTION.

You must call cciGetLastExceptionData() only when CCI_EXCEPTION is indicated; at other times, the function returns unpredictable results.

The *traceText* that is associated with the exception converts to a char* if the char* is US-ASCII. If the *traceText* is in another language, use cciGetLastExceptionDataW and its associated CCI_EXCEPTION_WIDE_ST structure, which stores the *traceText* as UTF-16.

If the exception has been raised by the broker, or by cciThrowExceptionW, the *traceText* element of the CCI_EXCEPTION_ST structure is an empty string.

**Syntax:**

```
void* cciGetLastExceptionData(
  int*              returnCode,
  CCI_EXCEPTION_ST*  exception_st);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). Possible return codes are:

- CCI_INV_DATA_POINTER
- CCI_NO_EXCEPTION_EXISTS
- CCI_EXCEPTION
- CCI_EXCEPTION_UNKNOWN
- CCI_EXCEPTION_FATAL
- CCI_EXCEPTION_RECOVERABLE
- CCI_EXCEPTION_CONFIGURATION
- CCI_EXCEPTION_PARSER
- CCI_EXCEPTION_CONVERSION
- CCI_EXCEPTION_DATABASE
- CCI_EXCEPTION_USER

**exception_st**

Specifies the address of a CCI_EXCEPTION_ST structure to receive data about the last exception (output). The type value returned in the lower four bits of the `exception_st.type` field is one of the following type values:

- CCI_EXCEPTION_ST_TYPE_EXCEPTION_BASE
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_TERMINATION
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_FATAL
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_RECOVERABLE
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_CONFIGURATION
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_PARSER
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_CONVERSION
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_DATABASE
- CCI_EXCEPTION_ST_TYPE_EXCEPTION_USER

The value returned in the `exception_st.messageNumber` field, for exceptions resulting in a BIP catalogued exception message, contains the message level in the high order bytes and the BIP message number in the lower four bytes.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
typedef struct exception_st {
  int               versionId;     /* Structure version identification */
  int               type;          /* Type of exception */
  int               messageNumber; /* Message number */
  int               insertCount;   /* Number of message inserts */
  CCI_STRING_ST     inserts[CCI_MAX_EXCEPTION_INSERTS];
                                   /* Array of message insert areas */
  const char*       fileName;      /* Source: file name */
  int               lineNumber;    /* Source: line number in file */
  const char*       functionName;  /* Source: function name */
  const char*       traceText;     /* Trace text associated with exception */
  CCI_STRING_ST     objectName;    /* Object name */
  CCI_STRING_ST     objectType;    /* Object type */
} CCI_EXCEPTION_ST;
char msgnumTypeStr[64];


CCI_EXCEPTION_ST exception_st = malloc(sizeof(CCI_EXCEPTION_ST));
int rc = 0;
```

```
memset(&exception_st,0,sizeof(exception_st));
cciGetLastExceptionData(&rc, &exception_st);
sprintf(msgnumTypeStr, "MsgNum: %d Type: %d",
(exception_st.messageNumber & 0x0ffff),
(exception_st.type & 0x0f));
```

## cciGetLastExceptionDataW

Gets diagnostic information about the last exception generated. Information about the last exception generated on the current thread is returned in a CCI_EXCEPTION_WIDE_ST output structure. The user-defined extension uses this function to determine whether any recovery is required when a utility function returns an error code.

You can call this function, when a utility function or user exit callback indicates that an exception has occurred, by setting *returnCode* to CCI_EXCEPTION.

Unless CCI_EXCEPTION is indicated you must not call cciGetLastExceptionDataW() because it returns unpredictable results.

**Syntax:**

```
void* cciGetLastExceptionDataW(
  int*                    returnCode,
  CCI_EXCEPTION_WIDE_ST*  exception_st);
```

**Parameters:**

**returnCode**

   Receives the return code from the function (output). Possible return codes are:
   - CCI_INV_DATA_POINTER
   - CCI_NO_EXCEPTION_EXISTS
   - CCI_EXCEPTION
   - CCI_EXCEPTION_UNKNOWN
   - CCI_EXCEPTION_FATAL
   - CCI_EXCEPTION_RECOVERABLE
   - CCI_EXCEPTION_CONFIGURATION
   - CCI_EXCEPTION_PARSER
   - CCI_EXCEPTION_CONVERSION
   - CCI_EXCEPTION_DATABASE
   - CCI_EXCEPTION_USER

**exception_st**

   Specifies the address of a CCI_EXCEPTION_WIDE_ST structure to receive data about the last exception (output). The type value returned in the lower four bits of the **exception_st.type** field is one of the following type values:
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_BASE
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_TERMINATION
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_FATAL
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_RECOVERABLE
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_CONFIGURATION
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_PARSER
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_CONVERSION
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_DATABASE
   - CCI_EXCEPTION_ST_TYPE_EXCEPTION_USER

The value returned in the **exception_st.messageNumber** field, for exceptions resulting in a BIP catalogued exception message, contains the message level in the high order bytes and the BIP message number in the lower four bytes.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
typedef struct exception_wide_st {
  int               versionId;     /* Structure version identification */
  int               type;          /* Type of exception */
  int               messageNumber; /* Message number */
  int               insertCount;   /* Number of message inserts */
  CCI_STRING_ST     inserts[CCI_MAX_EXCEPTION_INSERTS];
                                   /* Array of message insert areas */
  const char*       fileName;      /* Source: file name */
  int               lineNumber;     /* Source: line number in file */
  const char*       functionName;  /* Source: function name */
  CCI_STRING_ST     traceText;     /* Trace text associated with exception */
  CCI_STRING_ST     objectName;    /* Object name */
  CCI_STRING_ST     objectType;    /* Object type */
} CCI_EXCEPTION_WIDE_ST;
char msgnumTypeStr[64];


CCI_EXCEPTION_WIDE_ST exception_st = malloc(sizeof(CCI_EXCEPTION_WIDE_ST));
int rc = 0;
memset(&exception_st,0,sizeof(exception_st));
cciGetLastExceptionDataW(&rc, &exception_st);
sprintf(msgnumTypeStr, "MsgNum: %d Type: %d",
(exception_st.messageNumber & 0x0ffff),
(exception_st.type & 0x0f));
```

## cciLog

Use cciLog to write an error, warning, or informational event.

The event is logged by the broker interface, and includes the specified arguments as log data.

**Syntax:**

```
void cciLog(
  int*          returnCode,
  CCI_LOG_TYPE  type,
  char*         file,
  int           line,
  char*         function,
  CciChar*      messageSource,
  int           messageNumber,
  char*         traceText,
                ...);
```

**Parameters:**

**returnCode**

> The return code from the function (output). Possible return codes are:
> - CCI_SUCCESS
> - CCI_INV_DATA_POINTER
> - CCI_INV_LOG_TYPE

**type**   The type of event, as defined by CCI_LOG_TYPE (input). Valid values are:

- CCI_LOG_ERROR
- CCI_LOG_WARNING
- CCI_LOG_INFORMATION

**file**    The source file name where the function was invoked (input). The value is optional, but it is useful for debugging purposes.

**line**    The line number in the source file where the function was invoked (input). The value is optional, but it is useful for debugging purposes.

**function**
The function name that invoked the log function (input). The value is optional, but it is useful for debugging purposes.

**messageSource**
The fully-qualified location and name of the Windows message source or the Linux, UNIX, or z/OS message catalog.

To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

**messageNumber**
The message number identifying the event (input). If *messageNumber* is specified as zero, it is assumed that a message is not available. If *messageNumber* is non-zero, the specified message is written into the broker event log with any inserts provided in the variable argument list.

**traceText**
Trace information that is written into the broker service trace log (input). The information is optional, but it is useful for debugging purposes.

**...**    A C variable argument list containing any message inserts that accompany the message (input). These inserts are treated as character strings, and the variable arguments are assumed to be of type pointer to char.

char* characters must be strings in either ASCII (Latin) or EBCDIC (1047).

The last argument in this list must be (char*)0.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cciLogW
cciLogW logs an error, warning, or informational event. The event is logged by the broker interface and uses the specified arguments as log data.

**Syntax:**
```
void cciLogW(
  int*              returnCode,
  CCI_LOG_TYPE      type,
  const char*       file,
  int               line,
  const char*       function,
  const CciChar*    messageSource,
  int               messageNumber,
  const CciChar*    traceText,
                    ...
);
```

**Parameters:**

**returnCode**

The return code from the function (output). If the *messageSource* parameter is null, the *returnCode* is set to CCI_INV_DATA_POINTER.

**type**   The type of event, as defined by CCI_LOG_TYPE (input). Valid values are:
- CCI_LOG_ERROR
- CCI_LOG_WARNING
- CCI_LOG_INFORMATION

**file**   The source file name where the function was invoked (input). The value is optional, but it is useful for debugging purposes.

**line**   The line number in the source file where the function was invoked (input). The value is optional, but it is useful for debugging purposes.

**function**

The function name that invoked the log function (input). The value is optional, but it is useful for debugging purposes.

**messageSource**

The fully-qualified location and name of the Windows message source or the Linux, UNIX, or z/OS message catalog.

To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

**messageNumber**

The message number identifying the event (input). If *messageNumber* is specified as zero, it is assumed that a message is not available. If *messageNumber* is non-zero, the specified message is written into the broker event log with any inserts provided in the variable argument list (see example).

**traceText**

Trace information that is written into the broker service trace log (input). The information is optional, but it is useful for debugging purposes.

**...**   A C variable argument list containing any message inserts that accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to CciChar`.

The last argument in this list *must* be `(CciChar*)0`.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
void logSomethingWithBroker(CciChar* helpfulText,
                            char* file,
                            int line,
                            char* func
                            ){
  int rc = CCI_SUCCESS;
  /* set up the message catalog name */
  const CciChar* catalog = CciString("BIPv610", BIP_DEF_COMP_CCSID);

  cciLogW(&rc,
          CCI_LOG_INFORMATION
          file, line, func,
          catalog, BIP2111,
```

```
          helpfulText,
          helpfulText,
          (CciChar*)0
          );

  if(CCI_SUCCESS != rc){
    const CciChar* message = CciString("Failed to log message",
                                       BIP_DEF_COMP_CCSID);
    raiseExceptionWithBroker(message,
                             __FILE__,
                             __LINE__,
                             "logSomethingWithBroker");
  }
}
```

## cciMbsToUcs

Use this function to convert multibyte string data to Universal Character Set (UCS).

**Syntax:**

```
int cciMbsToUcs(
  int*         returnCode,
  const char*  mbString,
  CciChar*     ucsString,
  int          ucsStringLength,
  int          codePage);
```

**Parameters:**

**returnCode**
>   The return code from the function (output). Possible return codes are:
>   - CCI_SUCCESS
>   - CCI_BUFFER_TOO_SMALL
>   - CCI_INV_CHARACTER
>   - CCI_FAILURE
>   - CCI_INV_CODEPAGE

**mbString**
>   The string to be converted, expressed as 'file code' (input).

**ucsString**
>   The location of the resulting UCS-2 Unicode string (input). This string has
>   a trailing CciChar of 0, just as the *mbString* has a trailing byte of 0.

**ucsStringLength**
>   The length (in CciChar) of the buffer that you have provided (input). Each
>   byte in *mbString* expands to not more than one CciChar character, and this
>   defines an upper limit for the buffer size required.

**codePage**
>   The code page of the source string (input). The value of the code page
>   must be suitable for the compiler that you are using to compile the
>   user-defined node.
>
>   For an ASCII system, a value of 1208 (meaning code page ibm-1208, which
>   is UTF-8 Unicode) is a good choice if you are using cciMbsToUcs to
>   convert string constants for processing by WebSphere Message Broker. 1208
>   is appropriate for Linux, UNIX, and Windows systems.
>
>   On Linux, UNIX systems, nl_langinfo(CODEPAGE) gives you the code page
>   that has been selected by setlocale.

On z/OS, the default code page for WebSphere MQ, which is 500, must
not be used. Instead, use a code page value of 1047.

**Return values:**

The *returnCode* parameter is set to the converted length in half-words (UCS-2
characters).

## cciRegisterForThreadStateChange

This function registers a function to be called when the current thread enters a
particular state.

**Syntax:**

```
void cciRegisterForThreadStateChange(
        int               *returnCode,
        CciThreadContext  *threadContext,
        CciDataContext    *userContext,
        CciRegCallback     callback,
        CciCallbackType    type);
```

**Parameters:**

**returnCode**

The return code from the function (output). An input value of NULL signifies
that errors are silently handled, or are ignored by the broker. If the input value
is not NULL, the output value signifies the success status of the call. If the
threadContext parameter is not valid, *returnCode is set to
CCI_INV_THREAD_CONTEXT, and the callback is not registered.

**threadContext**

This parameter provides the thread context in which to register the callback
function and associated data. It is assumed that this parameter is obtained by
calling cniGetThreadContext() on the current thread. If NULL is supplied as
threadContext, the thread context is determined by the framework. This
method is less efficient than calling cniGetThreadContext.

**userContext**

This parameter allows the caller to provide a context pointer that is passed to
the callback function when it is called. This parameter can be NULL.

**callback**

This parameter is a pointer to the callback function that is to be called. This
function must be of the type CciRegCallback.

**type**

This parameter specifies whether the callback is to be called at the time when
the thread is ending, or when the thread is in one of the idle states. The idle
states can be one of the following values:

- CCI_THREAD_STATE_IDLE:

  The input node for the current thread is actively polling for data from the
  input source, but no data is available. Messages are not propagated through
  the message flow until data becomes available for the input node.

- CCI_THREAD_STATE_INSTANCE_END

  The input node for the current thread has stopped polling for data and the
  thread has been released. The thread is dispatched again either by the same
  input node or by another input node in the same message flow. This state is
  entered when additional instances, which have been deployed for a message
  flow, have been configured to cope with an influx of input data that has

now ceased. The input node continues to poll for input data on a single thread, and the other threads are released.

- CCI_THREAD_STATE_TERMINATION

    The current thread is ending. This action can happen when the broker is shut down, the execution group process is ending in a controlled manner, or when the message flow is being deleted. This state can occur after all nodes and parsers in the flow are deleted.

    Alternatively, the type parameter can be the result of a bit-wise OR operation on two or more of these values. In this case, the specified function is called when the thread enters the relevant state for each individual type value.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

Declaring the struct and function:

```
typedef struct {
    int        id;
}MyContext;

static int registered=0;

CciRegCallbackStatus switchThreadStateChange(
          CciDataContext *context, CciCallbackType type)
{
  char    traceText[256];
  char*   typeStr=0;
  MyContext* myContext = (MyContext*)context;

  if (type==CCI_THREAD_STATE_IDLE){
      typeStr = "idle";
  }else if(type==CCI_THREAD_STATE_INSTANCE_END){
      typeStr = "instance end";
  }else if (type==CCI_THREAD_STATE_TERMINATION){
      typeStr = "termination";
  }else{
      typeStr = "unknown";
  }

  memset(traceText,0,256);
  sprintf(traceText,"switchThreadStateChange: context id = %d, thread state %s",myContext->id,typeStr);
  cciServiceTrace(NULL,
                  NULL,
                  traceText);
  return CCI_THREAD_STATE_REGISTRATION_RETAIN;

}
```

Place the following code into the _Switch_evaluate function in the samples to enable you to read service trace, and to see when the message processing thread changes state:

```
/*register for thread state change*/
  CciMessageContext* messageContext = cniGetMessageContext(NULL,message);
  CciThreadContext*  threadContext  = cniGetThreadContext(NULL,messageContext);

  static MyContext myContext={1};

  if(registered==0){
```

```
  cciRegisterForThreadStateChange(
                  NULL,
                  threadContext,
                  & myContext,
                  switchThreadStateChange,
                  CCI_THREAD_STATE_IDLE |
                  CCI_THREAD_STATE_INSTANCE_END |
                  CCI_THREAD_STATE_TERMINATION);
registered=1;

}
```

This example registers only on the first thread that receives a message. If it is necessary to register every thread that receives a message, the user-defined extensions must remember on which threads they have registered.

By using the `userContext` parameter, you can see how data is passed from the code where the callback is registered to the actual callback function.

When registering the callback, a pointer to an instance of the **MyContext** struct is passed in. This pointer is the same pointer as is passed back to the callback. To ensure that the pointer is still valid when it is passed back to the callback, an instance of the struct is declared as static. Another technique to ensure that the pointer is valid is to allocate storage on the heap.

In the callback function, the **userContext** parameter can be cast to a **(MyContext\*)**. The original **MyContext** struct can be referenced through this address. This technique permits the passing of data from the code where the callback is registered to the callback function.

## cciRethrowLastException

This function re-throws the last exception that has been generated on the current thread. It is used to pass the exception back to the broker for further handling. The function, like a C exit call, does not return to the caller.

**Syntax:**

```
void cciRethrowLastException(int* returnCode);
```

**Parameters:**

**returnCode**
>   The return code from the function (output). The possible return code is CCI_NO_EXCEPTION_EXISTS

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
 if (rc == CCI_EXCEPTION) {
     cciRethrowLastException(&rc);
   }
```

## cciServiceDebugTrace

This function is very similar to cciServiceTrace with the only difference being that the entry is written to service trace only when service trace is active at debug level.

**Syntax:**

```
void cciServiceDebugTrace(
  int*         returnCode,
  CciObject*   object,
  const char*  traceText
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call will be re-thrown to the next upstream node in the flow. If input is not NULL, output will signify the success status of the call. If an exception occurs during execution, *returnCode* will be set to CCI_EXCEPTION on output. A call to **CciGetLastExceptionData** will provide details of the exception.

**object (input)**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**traceText (input)**

A string of characters that ends with NULL (input). This string will be written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, there could be several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string will be a static literal string in the source and therefore the same string will be in both the source code file and the formatted trace file.

This string must be in ISO-8859-1 (ibm-819) code page for user-defined extensions running on distributed platforms and must be in EBCDIC (1047) for user-defined extensions running on Z/OS See NLS section.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**
```
CciNode*         thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;

cciServiceTrace(&rc,(CciObject*)thisNode,">>_Switch_evaluate()");
checkRC(rc);
```

## cciServiceDebugTraceW

The function is very similar to cciServiceTraceW with the only difference being that the entry is written to service trace only when service trace is active at debug level.

**Syntax:**

```
void cciServiceDebugTraceW(
  int*          returnCode,
  CciObject*    object,
  const CciChar* traceText
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call will be re-thrown to the next upstream node in the flow. If input is not NULL, output will signify the success status of the call. If an exception occurs during execution, *returnCode* will be set to CCI_EXCEPTION on output. A call to **CciGetLastExceptionData** will provide details of the exception.

**object (input)**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**traceText (input)**

A string of characters that ends with NULL (input). This string will be written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, there could be several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string will be a static literal string in the source and therefore the same string will be in both the source code file and the formatted trace file.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
CciNode* thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;
CciChar* traceText = CciString(">>_Switch_evaluate()",BIP_DEF_COMP_CCSID");
cciServiceTraceW(&rc,(CciObject*)thisNode,traceText);
checkRC(rc);
```

## cciServiceTrace

Writes a message to service trace, if service trace is active.

The message that is written to service trace has the following format:

*<date-time stamp>* *<threadNumber>* +cciServiceTrace <nodeName> <nodeType> <traceText>, <nodeLabel>

**Syntax:**

```
void cciServiceTrace(
  int*          returnCode,
  CciObject*    object,
  const char*   traceText
);
```

**Parameters:**

**returnCode**
Receives the return code from the function (output). A NULL pointer input parameter signifies that the user-defined node does not handle errors. Exceptions thrown during the execution of this call are thrown again to the next upstream node in the flow. If the input parameter is not NULL, output signifies the success status of the call. If an exception occurs during execution, *returnCode* is set to CCI_EXCEPTION on output. Use CciGetLastExceptionData to get details of the exception.

**object (input)**
The address of the object that is to be associated with the trace entry (input). This object can be the address of a CciNode or a CciParser. If you specify a CciNode, the name of that node is written to trace. If you specify a CciParser, the name of the node that created the parser is written to trace. This object is also used to determine if the entry is written to trace. The entry is written only if trace is active for the node. Currently, nodes inherit their trace setting from the message flow.

If this parameter is NULL, the following actions are taken:
- <nodeName>, <nodeType>, <nodeLabel>, and <messageFlowLabel> are omitted from the trace entry.
- The entry is written based on the trace setting of the execution group.

**traceText (input)**
A string of characters that ends with NULL (input). This string is written to service trace, and you can use it to correlate trace entries with paths through the source code. For example, you might have several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string is a static literal string in the source, and therefore the same string is in both the source code file and the formatted trace file.

This string must be in ISO-8859-1 (ibm-819) code page for user-defined extensions running on distributed platforms, and must be in EBCDIC (1047) for user-defined extensions running on z/OS.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**
```
CciNode*         thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;

cciServiceTrace(&rc,(CciObject*)thisNode,">>_Switch_evaluate()");
checkRC(rc);
```

## cciServiceTraceW
cciServiceTraceW writes a message to service trace, if service trace is active.

The message that is written to service trace has the following format:
```
<date-time stamp> <threadNumber> +cciServiceTrace <nodeName> <nodeType> <traceText>, <nodeLabel>
```

**Syntax:**

```
void cciServiceTraceW(
  int*          returnCode,
  CciObject*    object,
  const CciChar* traceText
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call will be re-thrown to the next upstream node in the flow. If input is not NULL, output will signify the success status of the call. If an exception occurs during execution, *returnCode* will be set to CCI_EXCEPTION on output. A call to **CciGetLastExceptionData** will provide details of the exception.

**object (input)**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**traceText (input)**

A string of characters that ends with NULL (input). This string will be written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, there could be several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string will be a static literal string in the source and therefore the same string will be in both the source code file and the formatted trace file.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
CciNode*       thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;
const CciChar* traceText = CciString(">>_Switch_evaluate()",
                                     BIP_DEF_COMP_CCSID);
cciServiceTraceW(&rc,(CciObject*)thisNode,traceText);
checkRC(rc);
```

## cciThrowException

Use this function to throw an exception. The exception is thrown by the broker interface, and includes the specified arguments as exception data.

**Syntax:**

```
void cciThrowException(
  int*                returnCode,
  CCI_EXCEPTION_TYPE  type,
  char*               file,
  int                 line,
  char*               function,
```

```
CciChar*          messageSource,
int               messageNumber,
char*             traceText,
                  ...);
```

**Parameters:**

**returnCode**
> The return code from the function (output). The possible return code is CCI_INV_DATA_POINTER.

**type**  The type of exception (input). Valid values are:
- CCI_FATAL_EXCEPTION
- CCI_RECOVERABLE_EXCEPTION
- CCI_CONFIGURATION_EXCEPTION
- CCI_PARSER_EXCEPTION
- CCI_CONVERSION_EXCEPTION
- CCI_DATABASE_EXCEPTION
- CCI_USER_EXCEPTION

**file**  The source file name where the exception was generated (input). The value is optional, but it is useful for debugging purposes.

**line**  The line number in the source file where the exception was generated (input). The value is optional, but it is useful for debugging purposes.

**function**
> The function name which generated the exception (input). The value is optional, but it is useful for debugging purposes.

**messageSource**
> The fully-qualified location and name of the Windows message source or the Linux, UNIX, or z/OS message catalog.
>
> To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

**messageNumber**
> The message number identifying the exception (input). If *messageNumber* is specified as zero, it is assumed that a message is not available. If *messageNumber* is non-zero, the specified message is written into the broker event log with any inserts provided in the variable argument list.

**traceText**
> Trace information that is written into the broker service trace log (input). The information is optional, but it is useful in debugging problems.

**...**  A C variable argument list that contains any message inserts that accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to char`.
>
> char* characters must be strings in either ASCII (Latin) or EBCDIC (1047).
>
> The last argument in this list must be `(char*)0`.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cciThrowExceptionW

The cciThrowExceptionW exception is thrown by the broker interface and uses the specified arguments as exception data.

**Syntax:**

```
void cciThrowExceptionW(
  int*               returnCode,
  CCI_EXCEPTION_TYPE type,
  const char*        file,
  int                line,
  const char*        function,
  const CciChar*     messageSource,
  int                messageNumber,
  const CciChar*     traceText,
                     ...
);
```

**Parameters:**

**returnCode**

> The return code from the function (output). If the *messageSource* parameter is null, the *returnCode* is set to CCI_INV_DATA_POINTER.

**type**    The type of exception (input). Valid values are:

- CCI_FATAL_EXCEPTION
- CCI_RECOVERABLE_EXCEPTION
- CCI_CONFIGURATION_EXCEPTION
- CCI_PARSER_EXCEPTION
- CCI_CONVERSION_EXCEPTION
- CCI_DATABASE_EXCEPTION
- CCI_USER_EXCEPTION

**file**    The source file name where the exception was generated (input). The value is optional, but it is useful for debugging purposes.

**line**    The line number in the source file where the exception was generated (input). The value is optional, but it is useful for debugging purposes.

**function**

> The function name which generated the exception (input). The value is optional, but it is useful for debugging purposes.

**messageSource**

> A string that identifies the Windows message source or the Linux and UNIX message catalog. To use the current broker message catalog, specify BIPv610 on all operating systems.

**messageNumber**

> The message number identifying the exception (input). If *messageNumber* is specified as zero, it is assumed that a message is not available. If *messageNumber* is non-zero, the specified message is written into the broker event log with any inserts provided in the variable argument list.

**traceText**

> Trace information that is written into the service trace log (input). The information is optional, but it is useful in debugging problems.

**...**    A C variable argument list that contains any message inserts that

accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to CciChar`.

The last argument in this list *must* be `(Ccichar*)0`.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
void raiseExceptionWithBroker(CciChar* helpfulText,
                             char* file, /* which source file is broken */
                             int line,   /* line in above file */
                             char* func  /* function in above file */
                             ){
  int rc = CCI_SUCCESS;

  /* Set up the message catalog name */
  const char* catalog = "BIPv610";

  /* Convert the catalog name to wide characters.
   * BIP_DEF_COMP_CCSID is UTF-8 on distributed and LATIN1 on z/OS
   */
  int maxChars =  strlen(catalog)+1;
  CciChar* wCatalog =(CciChar*)malloc(maxChars*sizeof(CciChar));
  cciMbsToUcs(&rc, catalog, wCatalog, maxChars, BIP_DEF_COMP_CCSID);

  /* The above might have failed, but we are already throwing an exception,
   * so rc is now set to type success. */
  rc = CCI_SUCCESS;

  /* Throw the exception. The explanation will be added as the traceText and
   *  as an insert to the message
   */
  cciThrowExceptionW(&rc,
                    CCI_USER_EXCEPTION,
                    file, line, func,
                    wCatalog, BIP2111,
                    helpfulText,
                    helpfulText,
                    (CciChar*)0
                    );
  /* The above might have failed, but we are already throwing an exception,
   * so the value of rc is not important. */
}
```

## cciIsTraceActive

cciIsTraceActive reports whether trace is active and the level at which trace is active.

**Syntax:**

```
CCI_TRACE_TYPE cciIsTraceActive(
  int*        returnCode,
  CciObject*  object);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call will be re-thrown to the

next upstream node in the flow. If input is not NULL, output will signify the success status of the call. If an exception occurs during execution, *returnCode will be set to CCI_EXCEPTION on output. A call to **CciGetLastExceptionData** will provide details of the exception.

**object**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**Return values:**

A CCI_TRACE_TYPE value indicating the level of trace that is currently active. The CCI_TRACE_TYPE type has the following possible values:
- CCI_USER_NORMAL_TRACE
- CCI_USER_DEBUG_TRACE
- CCI_ SERVICE_NORMAL_TRACE
- CCI_SERVICE_DEBUG_TRACE
- CCI_TRACE_NONE

These return values are bitwise values. Combinations of these values are also possible, for example:
- CCI_USER_NORMAL_TRACE + CCI_ SERVICE_NORMAL_TRACE
- CCI_USER_NORMAL_TRACE + CCI_SERVICE_DEBUG_TRACE
- CCI_USER_DEBUG_TRACE + CCI_ SERVICE_NORMAL_TRACE
- CCI_USER_DEBUG_TRACE + CCI_SERVICE_DEBUG_TRACE

CCI_TRACE_NONE is a zero value and all other values are non zero.

Two further values can be used as bitmasks when querying the active level of trace. These are:
- CCI_USER_TRACE
- CCI_SERVICE_TRACE

For example, the expression (`traceLevel & CCI_USER_TRACE`) will evaluate to a non zero value for *traceLevel* for the following return values:
- CCI_USER_NORMAL_TRACE + CCI_ SERVICE_NORMAL_TRACE
- CCI_USER_NORMAL_TRACE + CCI_SERVICE_DEBUG_TRACE
- CCI_USER_DEBUG_TRACE + CCI_ SERVICE_NORMAL_TRACE
- CCI_USER_DEBUG_TRACE + CCI_SERVICE_DEBUG_TRACE
- CCI_USER_NORMAL_TRACE
- CCI_USER_DEBUG_TRACE

The expression (`traceLevel & CCI_USER_TRACE`) will evaluate to zero for *traceLevel* for the following return values:
- CCI_SERVICE_NORMAL_TRACE
- CCI_SERVICE_DEBUG_TRACE
- CCI_TRACE_NONE

**Example:**

```
CciNode*        thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;

const CCI_TRACE_TYPE  traceActive = cciIsTraceActive(&rc, (CciObject*)thisNode);
checkRC(rc);
```

## cciUcsToMbs

Use this function to convert Universal Character Set (UCS) data to multibyte string data. This function is, typically, used only for formatting diagnostic messages. Normal processing is best done in UCS-2, which can represent all characters from all languages.

The sample code in `BipSampPluginUtil.c` contains more utilities for processing UCS-2 characters in a portable way.

**Syntax:**

```
int cciUcsToMbs(
  int*         returnCode,
  const CciChar* ucsString,
  char*        mbString,
  int          mbStringLength,
  int          codePage);
```

**Parameters:**

**returnCode**

> The return code from the function (output).
>
> Possible return codes are:
> - CCI_SUCCESS
> - CCI_BUFFER_TOO_SMALL
> - CCI_INV_CHARACTER
> - CCI_FAILURE
> - CCI_INV_CODEPAGE

**ucsString**

> The string to be converted, expressed as UCS-2 Unicode (input).

**mbString**

> The location of the resulting string (input). The string has a trailing byte of 0, just as the Unicode has a trailing `CciChar` of 0.

**mbStringLength**

> The length (in bytes) of the buffer that you have provided (input). Each `CciChar` in the source string expands to one byte (for SBCS code pages), or up to not more than the code page's MB_CUR_MAX value (typically less than five bytes), which defines an upper limit of the buffer size required.

**codePage**

> The code page of the resulting string (input). The value of the code page must be suitable for the compiler that you are using to compile the user-defined node.
>
> For an ASCII system, a value of 1208 (meaning code page ibm-1208, which is UTF-8 Unicode) is a good choice if you are using cciUcsToMbs to convert string constants for processing by WebSphere Message Broker. 1208 is appropriate for Linux, UNIX, and Windows.
>
> On Linux and UNIX systems, `nl_langinfo(CODEPAGE)` gives you the code page that has been selected by the setlocale command.

On z/OS, the default code page for WebSphere MQ, which is 500, must not be used. Instead, use a code page value of 1047.

**Return values:**

The *returnCode* parameter is set to the converted length in bytes.

## cciUserDebugTrace

Use cciUserDebugTrace to write a message from a message catalog (with inserts) to user trace when user trace is active at debug level.

If user trace is not active at debug level, an entry is written to service trace when service trace is active.

**Syntax:**

```
void cciUserDebugTrace(
  int*           returnCode,
  CciObject*     object,
  const CciChar* messageSource,
  int            messageNumber,
  const char*    traceText,
  ...
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input parameter indicates that the user-defined node does not handle errors. All exceptions that are thrown during the execution of this call are thrown again to the next upstream node in the flow. If the input parameter is not NULL, the output signifies the success status of the call. If an exception occurs during execution, *returnCode* is set to CCI_EXCEPTION on output. Call CciGetLastExceptionData to obtain details of the exception.

**object**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If you specify a CciNode*, the name of that node is written to trace. If you specify a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry is written to trace. The entry is written only if trace is active for the node. Nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**messageSource**

The fully qualified location and name of the Windows message source or the Linux, UNIX, or z/OS message catalog.

To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

When trace is formatted, a message from the NLS version of this catalog is written.

The locale used is that of the environment where the trace is formatted. You can run the broker on one operating system, read the log on that operating system, then format the log on a different operating system. For example, if the broker is running on Linux, UNIX, or z/OS but no `.cat` file is available, you could read the log, then transfer it to Windows where the log can be formatted by using the `.properties` file.

If this parameter is NULL, the effect is the same as specifying an empty string. That is, all other information is written to the log, and the catalog field has an empty string value. Therefore, the log formatter cannot find the message source and fails to format this entry.

**messageNumber**
The number that identifies the message within the specified *messageSource* (input). If the *messageSource* does not contain a message that corresponds to this *messageNumber*, the log formatter fails to format this entry.

**traceText**
A string of characters that ends with NULL (input). This string is written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, you might have several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string is a static literal string in the source and therefore the same string is in both the source code file and the formatted trace file.

**...** A C variable argument list that contains other message inserts that accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to char`. The last argument in this list *must* be `(char*)0`.

- For user-defined extensions that are running on distributed platforms, the `char*` arguments must be in ISO-8859-1 (ibm-918) code page.
- For user-defined extensions that are running on z/OS platforms, the `char*` arguments must be in EBCIDIC (1047).

These requirements apply to all char* arguments in **traceText** and the variable argument list of inserts (**...**).

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**
```
const CciChar*  myMessageSource=CciString("SwitchMSG",BIP_DEF_COMP_CCSID);
CciNode*        thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;

const char* mbElementName  = mbString((CciChar*)&elementName,BIP_DEF_COMP_CCSID);
const char* mbElementValue = mbString((CciChar*)&elementValue,BIP_DEF_COMP_CCSID);
const char* traceTextFormat = "Switch Element: name=%s, value=%s";
char* traceText = (char*)malloc(strlen(traceTextFormat) +
                                strlen(mbElementName) +
                                strlen(mbElementValue));
sprintf(traceText,traceTextFormat,mbElementName,mbElementValue);

cciUserDebugTrace(&rc,
                  (CciObject*)thisNode,
                  myMessageSource,
                  2,
                  traceText,
                  mbElementName,
                  mbElementValue,
                  (char*)0);
free((void*)mbElementName);
free((void*)mbElementValue);
free((void*)traceText);
```

## cciUserDebugTraceW

Use cciUserDebugTraceW to write a message from a message catalog (with inserts) to user trace when user trace is active at debug level. A message is also written to service trace, if service trace is active.

If user trace is not active at debug level, an entry is written to service trace when service trace is active at any level.

**Syntax:**

```
void cciUserDebugTraceW(
  int*            returnCode,
  CciObject*      object,
  const CciChar*  messageSource,
  int             messageNumber,
  const CciChar*  traceText,
  ...
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call will be re-thrown to the next upstream node in the flow. If input is not NULL, output will signify the success status of the call. If an exception occurs during execution, *returnCode will be set to CCI_EXCEPTION on output. A call to **CciGetLastExceptionData** will provide details of the exception.

**object**

The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**messageSource**

A string that identifies the Windows message source or the Linux and UNIX message catalog (input). When trace is formatted, a message from the NLS version of this catalog is written. The locale used is that of the environment where the trace is formatted. It is possible to run the broker on one type of operating system, read the log on that operating system, then format the log on a different operating system. For example, if the broker is running on Linux or UNIX but no .cat file is available, the user could read the log, then transfer it to Windows where the log can be formatted by using the .properties file.

If this parameter is NULL, the effect is the same as specifying an empty string. That is, all other information will be written to the log, and the catalog field will have an empty string value. Therefore, the log formatter will not be able to find the message source. Consequently, the log formatter will fail to format this entry.

**messageNumber**

The number that identifies the message within the specified *messageSource* (input). If the *messageSource* does not contain a message that corresponds to this *messageNumber*, the log formatter fails to format this entry.

**traceText**

A string of characters that ends with NULL (input). This string will be written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, there could be several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. *traceText* can be used to distinguish between these different paths. That is, the *traceText* string will be a static literal string in the source and therefore the same string will be in both the source code file and the formatted trace file.

**...** A C variable argument list that contains any message inserts that accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to CciChar`.

The last argument in this list *must* be `(CciChar*)0`.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**
```
const CciChar*  myMessageSource=CciString("SwitchMSG",BIP_DEF_COMP_CCSID);
CciNode*        thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;
const CciChar*  traceText = CciString("Found an element name and value",
                                      BIP_DEF_COMP_CCSID);

cciUserDebugTraceW(&rc,
                   (CciObject*)thisNode,
                   myMessageSource,
                   2,
                   traceText,
                   elementName,
                   elementValue,
                   (CciChar*)0);
```

## cciUserTrace

Use cciUserTrace to write a message from a message catalog (with inserts) to user trace. A message is also written to service trace, if service trace is active.

The message written to user trace has the following format:

*<date-time stamp> <threadNumber>* UserTrace *<Message text with inserts> <Message Explanation>*

**Syntax:**
```
void cciUserTrace(
  int*            returnCode,
  CciObject*      object,
  const CciChar*  messageSource,
  int             messageNumber,
  const char*     traceText,
                  ...
);
```

**Parameters:**

**returnCode**

Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output signifies the success

status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. A call to CciGetLastExceptionData provides details of the exception.

**object**
The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If it is a CciNode*, the name of that node is written to trace. If it is a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is only written if trace is active for the node. Currently, nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**messageSource**
The fully-qualified location and name of the Windows message source, or the Linux, UNIX, or z/OS message catalog (input).

To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

When trace is formatted, a message from the NLS version of this catalog is written. The locale used is that of the environment where the trace is formatted.

You can run the broker on one operating system, read the log on that operating system, then format it on a different operating system. For example, if the broker is running on Linux, but no `.cat` file is available, you could read the log, then transfer it to Windows, where the log can be formatted by using the `.properties` file.

If this parameter is NULL, the effect is the same as specifying an empty string. That is, all other information is written to the log, and the catalog field has an empty string value. Therefore, the log formatter cannot find the message source. Consequently, the log formatter fails to format this entry.

**messageNumber**
The number that identifies the message within the specified *messageSource* (input). If the *messageSource* does not contain a message that corresponds to this *messageNumber*, the log formatter fails to format this entry.

**traceText**
A string of characters that ends with NULL (input). This string is written to service trace, and provides an easy way to correlate trace entries with paths through the source code. For example, if several paths through the code result in the same message (*messageSource* and *messageNumber*) being written to trace, you can specify *traceText* to distinguish between these different paths. That is, the *traceText* string is a static literal string in the source, and therefore the same string appears in both the source code file and the formatted trace file.

**...** A C variable argument list that contains any message inserts that accompany the message (input). These inserts are treated as character strings, and the variable arguments are assumed to be of type `pointer to char`.

The last argument in this list must be `(char*)0`.

- For user-defined extensions that are running on distributed platforms, the `char*` arguments must be in ISO-8859-1 (ibm-918) code page.
- For user-defined extensions that are running on z/OS platforms, the `char*` arguments must be in EBCIDIC (1047).

These requirements include all char* arguments in **traceText** and the variable argument list of inserts (**...**).

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**
```
const CciChar*   myMessageSource=CciString("SwitchMSG",BIP_DEF_COMP_CCSID);
CciNode*          thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;

cciUserTrace(&rc,
             (CciObject*)thisNode,
              myMessageSource,
              1,
              "propagating to add terminal",
              "add",
              (char*)0);
 checkRC(rc);
```

# cciUserTraceW

Use cciUserTraceW to write a message from a message catalog (with inserts) to user trace. A message is also written to service trace, if service trace is active.

The message written to user trace has the following format:

*<date-time stamp> <threadNumber>* UserTrace *<Message text with inserts> <Message Explanation>*

**Syntax:**
```
void cciUserTraceW(
  int*              returnCode,
  CciObject*        object,
  const CciChar*    messageSource,
  int               messageNumber,
  const CciChar*    traceText,
                    ...
);
```

**Parameters:**

**returnCode**
Receives the return code from the function (output). A NULL pointer input signifies that the user-defined node does not wish to deal with errors. Any exceptions that are thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output indicates the success status of the call. If an exception occurs during execution, *returnCode is set to CCI_EXCEPTION on output. Call CciGetLastExceptionData to obtain details of the exception.

**object**
The address of the object that is to be associated with the trace entry (input). This object can be a CciNode* or a CciParser*. If you specify a CciNode*, the name of that node is written to trace. If you specify a CciParser*, the name of the node that created the parser is written to trace. This object is also used to determine if the entry should be written to trace. The entry is written only if trace is active for the node. Nodes inherit their trace setting from the message flow.

If this parameter is NULL, the trace level for the execution group is returned.

**messageSource**

The fully-qualified location and name of the Windows message source or the Linux, UNIX, or z/OS message catalog.

To use the current broker message catalog, specify BIPv610 on all operating systems. Alternatively, you can create your own message catalog.

When trace is formatted, a message from the NLS version of this catalog is written.

The locale used is that of the environment where the trace is formatted. You can run the broker on one operating system, read the log on that operating system, then format the log on a different operating system. For example, if the broker is running on Linux, UNIX, or z/OS but no `.cat` file is available, you could read the log, then transfer it to Windows where the log can be formatted by using the `.properties` file.

If this parameter is NULL, the effect is the same as specifying an empty string. That is, all other information is written to the log, and the catalog field has an empty string value. Therefore, the log formatter cannot find the message source and fails to format this entry.

**messageNumber**

The number that identifies the message within the specified *messageSource* (input). If the *messageSource* does not contain a message that corresponds to this *messageNumber*, the log formatter fails to format this entry.

**traceText**

A string of characters that ends with NULL (input). This string is written to service trace and provides an easy way to correlate trace entries with paths through the source code. For example, there could be several paths through the code that result in the same message (*messageSource* and *messageNumber*) being written to trace. Use *traceText* to distinguish between these different paths. That is, the *traceText* string is a static literal string in the source, and therefore the same string is in both the source code file and the formatted trace file.

**...** A C variable argument list that contains any message inserts that accompany the message (input). These inserts are treated as character strings and the variable arguments are assumed to be of type `pointer to CciChar`.

The last argument in this list must be `(CciChar*)0`.

**Return values:**

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

**Example:**

```
const CciChar*   myMessageSource=CciString("SwitchMSG",BIP_DEF_COMP_CCSID);
const CciChar*   text = CciString("propagating to add terminal",
                                  BIP_DEF_COMP_CCSID);
const CciChar*   insert = CciString("add", BIP_DEF_COMP_CCSID);
CciNode*         thisNode = ((NODE_CONTEXT_ST*)context)->nodeObject;
int              rc = CCI_SUCCESS;

cciUserTrace(&rc,
             (CciObject*)thisNode,
             myMessageSource,
             1,
```

```
                                           text,
                                           insert,
                                           (CciChar*)0);
                        checkRC(rc);
```

## C skeleton code

Use the skeleton code that is supplied as guidance for your C user-defined node.
The code has the minimum content that is required to compile a user-defined node
successfully.

```c
#ifdef __WIN32
#include <windows.h>
#endif
#include <BipCos.h>
#include <BipCci.h>
#include <BipCni.h>
#include <malloc.h>

#define BIP_DEF_COMP_CCSID 437
CciChar* constNodeFactory = 0;
CciChar* constNodeName     = 0;
CciChar* constTerminalName    = 0;
CciChar* constOutTerminalName    = 0;

CciChar* CciString(
  const char* source,
  int         codepage
){
  /* Maximum number of characters in Unicode representation */
  int maxChars = strlen(source) + 1 ;
  CciChar* buffer = (CciChar*)malloc(maxChars * sizeof(CciChar)) ;
  int rc ;
  cciMbsToUcs(&rc, source, buffer, maxChars, codepage) ;
  return buffer ;
}
void initNodeConstants(){
  constNodeFactory       = CciString("myNodeFactory", BIP_DEF_COMP_CCSID);
  constNodeName          = CciString("myNode",BIP_DEF_COMP_CCSID);
  constTerminalName      = CciString("in",BIP_DEF_COMP_CCSID);
  constOutTerminalName   = CciString("out",BIP_DEF_COMP_CCSID);
}

typedef struct {
  CciTerminal* iOutTerminal;
}MyNodeContext;

CciContext* createNodeContext(
  CciFactory* factoryObject,
  CciChar*    nodeName,
  CciNode*    nodeObject
){

  MyNodeContext * p = (MyNodeContext *)malloc(sizeof(MyNodeContext));

  /*here we would create an instance of some data structure
  where we could store context about this node instance.
  We would return a pointer to this struct and that pointer
  will be passed to our other implementation functions */

  /* now we create an input terminal for the node*/
  cniCreateInputTerminal(NULL, nodeObject, (CciChar*)constTerminalName);
  p->iOutTerminal = cniCreateOutputTerminal(NULL, nodeObject, (CciChar*)constOutTerminalName);
  return((CciContext*)p);
}
```

```
/****************************************************************/
/*                                                              */
/* Plugin Node Implementation Function:         cniEvaluate() */
/*                                                              */
/****************************************************************/
void evaluate(
  CciContext* context,
  CciMessage* localEnvironment,
  CciMessage* exceptionList,
  CciMessage* message
){
  /* we would place our node's processing logic in here*/
  return;
}

int run(
  CciContext* context,
  CciMessage* localEnvironment,
  CciMessage* exceptionList,
  CciMessage* message
)
{
  char* buffer="<doc><test>hello</test></doc>";
  CciChar* wBuffer=CciString(buffer,BIP_DEF_COMP_CCSID);
  //cniSetInputBuffer(NULL,message,(void*)wBuffer,strlen(buffer) * sizeof(CciChar));
  cniSetInputBuffer(NULL,message,(void*)buffer,strlen(buffer));
  cniFinalize(NULL,message,0);

  cniPropagate(NULL,((MyNodeContext*)context)->iOutTerminal,localEnvironment,exceptionList,message);
  return CCI_SUCCESS_CONTINUE;
}




#ifdef __cplusplus
extern "C"{
#endif
CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix bipGetMessageflowNodeFactory()
{
  CciFactory*      factoryObject;

  /* Before we proceed, we need to initialize all the static constants */
  /* that might be used by the plug-in.                                */
  initNodeConstants();

  /* Create the Node Factory for this plug-in */
  /* If any errors/exceptions    */
  /* occur during the execution of this utility function, then as we have not */
  /* supplied the returnCode argument, the exception will bypass the plugin   */
  /* and be directly handled by the broker.                                   */
  factoryObject = cniCreateNodeFactory(0, (unsigned short *)constNodeFactory);
  if (factoryObject == CCI_NULL_ADDR) {
    /* Any further local error handling can go here */
  }
  else {
    /* Define the node supported by this factory */
    static CNI_VFT vftable = {CNI_VFT_DEFAULT};
    /* Setup function table with pointers to node implementation functions */
    vftable.iFpCreateNodeContext = createNodeContext;
    vftable.iFpEvaluate          = evaluate;
    vftable.iFpRun               = run;

    /* Define a node type supported by our factory. If any errors/exceptions    */
    /* occur during the execution of this utility function, then as we have not */
    /* supplied the returnCode argument, the exception will bypass the plugin   */
    /* and be directly handled by the broker.                                   */
    cniDefineNodeClass(NULL, factoryObject, (CciChar*)constNodeName, &vftable);
```

```
  }

  /* Return address of this factory object to the broker */
  return(factoryObject);
}
#ifdef __cplusplus
}
#endif
```

## GNU makefile

The following file is a makefile that lists the files, dependencies, and rules by which the C user-defined node is compiled.

```
.SUFFIXES : .so .a .o .c

R1INC  = .
R1LIB  = .

# WMQI
MQSIDIR  = /cmvc/back/inst.images/x86_linux_2/shipdata/opt/mqsi
MQSIINC  = $(MQSIDIR)/include
MQSILIB  = $(MQSIDIR)/lib

# WMQ
MQIDIR  = /usr/mqm

CC   = /usr/bin/g++
LD   = ${CC}


OBJ = .o
LIL = .lil
THINGSTOCLEAN = *${OBJ}
CFLAGS  = -fpic -c #-pedantic -x c -Wall
CFLAGSADD = -I${R1INC} -I${MQSIINC} -I${MQSIINC}/plugin ${DEFINES}
DEFINES  = -DLINUX

LIBADD  = -L${MQSILIB} -limbdfplg
LDFLAG  = -shared ${LIBADD}

#CC   = /usr/bin/gcc
#LD   = ${CC}

OBJECTS = skeleton${OBJ}
.c.o : ; ${CC} ${CFLAGS} ${CFLAGSADD} $<

ALL : ${OBJECTS} Samples${LIL}
clean:
 rm *${OBJ} *${LIL}


skeleton${OBJ}: skeleton.c


Samples${LIL}: ${OBJECTS}
 ${LD} -o $@ ${OBJECTS} ${LDFLAG}
```

# Utility function return codes and values

By convention, the return code output parameter of all utility functions is set to indicate successful completion, or an error. The table lists all return codes with their meanings.

These return codes are defined in the `BipCci.h` header file.

| Return code | Explanation |
|---|---|
| CCI_BUFFER_TOO_SMALL | The output buffer is not large enough to store the requested data. |
| CCI_EXCEPTION | An exception occurred. |
| CCI_EXCEPTION_CONFIGURATION | A configuration exception was detected when invoking the function. [1] |
| CCI_EXCEPTION_CONVERSION | A conversion exception was detected when invoking the function. [1] |
| CCI_EXCEPTION_DATABASE | A database exception was detected when invoking the function. |
| CCI_EXCEPTION_FATAL | A fatal exception was detected when invoking the function. [1] |
| CCI_EXCEPTION_PARSER | A parser exception was detected when invoking the function. [1] |
| CCI_EXCEPTION_RECOVERABLE | A recoverable exception was detected when invoking the function. [1] |
| CCI_EXCEPTION_UNKNOWN | An unknown exception was specified or encountered. |
| CCI_EXCEPTION_USER | A user exception was detected when invoking the function. [1] |
| CCI_FAILURE | A function was unsuccessful. |
| CCI_FAILURE_CONTINUE | cniRun() return value: rollback message processing and continue thread execution |
| CCI_FAILURE_RETURN | cniRun() return value: rollback message processing and return thread to pool |
| CCI_INV_CODEPAGE | An invalid code page number was specified. |
| CCI_INV_CHARACTER | An invalid character was detected in the buffer to be converted. |
| CCI_INV_DATA_BUFLEN | A data buffer length of zero was specified. |
| CCI_INV_DATA_POINTER | A null pointer was specified for the address of an output data area. |
| CCI_INV_ELEMENT_OBJECT | A null pointer was specified for the element object. |
| CCI_INV_FACTORY_NAME | A factory name that is not valid (blank) was specified. |
| CCI_INV_FACTORY_OBJECT | A null pointer was specified for the factory object. |
| CCI_INV_IMPL_FUNCTION | An invalid combination of conditional implementation functions was specified |
| CCI_INV_LENGTH | A length of zero was specified. |
| CCI_INV_LOG_TYPE | The specified log type is not valid. |

| Return code | Explanation |
|---|---|
| CCI_INV_MESSAGE_CONTEXT | A null pointer was specified for the message context. |
| CCI_INV_MESSAGE_OBJECT | A null pointer was specified for the message object. |
| CCI_INV_NODE_ENV | Attempt to dispatch a thread from a non-input node. |
| CCI_INV_NODE_NAME | A node name that is not valid (blank) was specified. |
| CCI_INV_NODE_OBJECT | A null pointer was specified for the node object. |
| CCI_INV_OBJECT_NAME | Characters specified in the object name were not valid. |
| CCI_INV_PARSER_NAME | A parser class name that is not valid (blank) was specified. |
| CCI_INV_PARSER_OBJECT | A null pointer was specified for the parser object. |
| CCI_INV_SQL_EXPR_OBJECT | A null pointer was specified for an SQL expression value. |
| CCI_INV_STATEMENT | A statement was not specified. |
| CCI_INV_TERMINAL_NAME | A terminal name that is not valid (blank) was specified. |
| CCI_INV_TERMINAL_OBJECT | A null pointer was specified for the terminal object. |
| CCI_INV_TRANSACTION_TYPE | An invalid value was specified for the transaction type. |
| CCI_INV_VFTP | A null pointer was specified for the address of the user-defined extension virtual function pointer table. |
| CCI_MISSING_IMPL_FUNCTION | A mandatory implementation function was not defined in the function pointer table. |
| CCI_NAME_EXISTS | A parser with the same class name already exists. |
| CCI_NO_BUFFER_EXISTS | No buffer exists for the specified parser object. |
| CCI_NO_EXCEPTION_EXISTS | No previous exception was found for this thread. |
| CCI_NO_THREADS_AVAILABLE | No threads were available to be dispatched. |
| CCI_NULL_ADDR | A function that should return an address was unsuccessful; zero is returned instead. |
| CCI_PARSER_NAME_TOO_LONG | The name of the parser class is too long. |
| CCI_SUCCESS | Successful completion. |
| CCI_SUCCESS_CONTINUE | cniRun() return value: commit message processing and continue thread execution |
| CCI_SUCCESS_RETURN | cniRun() return value: commit message processing and return thread to pool |
| CCI_TIMEOUT | cniRun() return value: no message processing but continue thread execution |

**Note:**

1. This return code is returned only by cniGetLastExceptionData to indicate the type of the last exception.

# Available parsers

A parser is called by the broker only when that parser is required. The parser that is called depends upon the parser that has been specified.

For certain implementation functions, it might be necessary to specify the name of a parser supplied with WebSphere Message Broker. For example, functions include:

- cniCreateElementAfterUsingParser
- cniCreateElementAsFirsthChildUsingParser
- cniCreateElementAsLastChildUsingParser
- cniCreateElementAsLastChildFromBitstream
- cniCreateElementBeforeUsingParser

When using these functions, you must specify the correct class name of the parser. The following tables provide a summary of the parsers, root element names, and class names for different headers.

The following table shows the **Body** parsers.

| Parser | Root element name | Class name |
|--------|-------------------|------------|
| BLOB | BLOB | NONE |
| DataObject | DataObject | DataObject |
| IDOC (deprecated) | IDOC | IDOC |
| JMSMap | JMSMap | JMS_MAP |
| JMSStream | JMSStream | JMS_STREAM |
| MIME | MIME | MIME |
| MRM | MRM | MRM |
| SOAP | SOAP | SOAP |
| XML (deprecated) | XML | xml |
| XMLNS | XMLNS | xmlns |
| XMLNSC | XMLNSC | xmlnsC |

The following table shows the **Header** parsers.

| Parser | Root element name | Class name |
|--------|-------------------|------------|
| EmailOutputHeader | EmailOutputHeader | EMAILHDR |
| HTTPInputHeader | HTTPInputHeader | WSINPHDR |
| HTTPReplyHeader | HTTPReplyHeader | WSREPHDR |
| HTTPRequestHeader | HTTPRequestHeader | WSREQHDR |
| HTTPResponseHeader | HTTPResponseHeader | WSRSPHDR |
| JMS header | JMSTransport | jms_transport |
| MQCFH | MQPCF | MQPCF |
| MQCIH | MQCIH | MQCICS |

| Parser | Root element name | Class name |
|---|---|---|
| MQDLH | MQDLH | MQDEAD |
| MQIIH | MQIIH | MQIMS |
| MQMD | MQMD | MQHMD |
| MQMDE | MQMDE | MQHMDE |
| MQRFH | MQRFH | MQHRF |
| MQRFH2 | MQRFH2 | MQHRF2 |
| MQRFH2C | MQRFH2C | MQHRF2C |
| MQRMH | MQRMH | MQHREF |
| MQSAPH | MQSAPH | MQHSAP |
| MQWIH | MQWIH | MQHWIH |
| SMQ_BMH | SMQ_BMH | SMQBAD |

When using the MQMD parser, the MQMD is assumed to be a V2 MQMD.

The following table shows the **Properties** parser.

| Parser | Root element name | Class name |
|---|---|---|
| Properties | Properties | PropertyParser |

You can also create your own user-defined parsers, or you can make use of user-defined parsers that have been supplied by independent software vendors.

# XML, MRM, and XMLNSC parser constants

The names of the XML and MRM parser constants, together with their corresponding values, and a link to the XMLNSC constants.

When you are writing user-defined extensions you might need to know the value of various constants.

## XML parser constants

| Name | Value |
|---|---|
| Element | 0x01000000 |
| tag | 0x01000000 |
| ParserRoot | 0x01000010 |
| Content | 0x02000000 |
| pcdata | 0x02000000 |
| attr | 0x03000000 |
| Attribute | 0x03000000 |
| UnparsedEntityDecl | 0x05000004 |
| NotationDecl | 0x05000008 |
| EntityDecl | 0x05000011 |
| ParameterEntityDecl | 0x05000012 |
| ExternalEntityDecl | 0x05000014 |

| | |
|---|---|
| XmlDecl | 0x05000018 |
| DocTypeDecl | 0x05000020 |
| IntSubset | 0x05000021 |
| ExtSubset | 0x05000022 |
| AttributeList | 0x05000024 |
| AttributeDef | 0x05000028 |
| ExternalParameterEntityDecl | 0x05000040 |
| WhiteSpace | 0x06000002 |
| PublicId | 0x06000004 |
| SystemId | 0x06000008 |
| NotationReference | 0x06000010 |
| Version | 0x06000011 |
| Encoding | 0x06000012 |
| Standalone | 0x06000014 |
| Comment | 0x06000018 |
| EntityReferenceStart | 0x06000020 |
| EntityReferenceEnd | 0x06000021 |
| DocTypeComment | 0x06000022 |
| AsisElementContent | 0x06000028 |
| CDataSection | 0x06000040 |
| EntityDeclValue | 0x06000041 |
| AttributeDefValue | 0x06000042 |
| AttributeDefDefaultType | 0x06000044 |
| DocTypeWhiteSpace | 0x06000080 |
| ProcessingInstruction | 0x07000002 |
| ElementDef | 0x07000004 |
| DocTypePI | 0x07000008 |
| AttributeDefType | 0x07000010 |
| RequestedDomain | 0x07000011 |

## MRM parser constants

| Name | Value |
|---|---|
| PreDefStructureFav | 0x01000000 |
| PreDefStructure | 0x01000001 |
| SelfDefStructure | 0x01000002 |
| StructureInstance | 0x01000004 |
| MrmRoot | 0x01000008 |
| mtiSelfDefMessage | 0x01000010 |
| mtiPreDefMessage | 0x01000012 |
| mtiSelfDefIdentifier | 0x02000001 |
| mtiSdfFieldType | 0x02000002 |

| | |
|---|---|
| mtiSdfCharsCodepage | 0x02000008 |
| mtiSdfCharsEcho | 0x02000010 |
| mtiSdfCharsScale | 0x02000011 |
| mtiSdfCharsDateFmt | 0x02000012 |
| mtiSdfCharsTimeFmt | 0x02000014 |
| mtiSdfCharsTimeStampFmt | 0x02000018 |
| mtiSdfCharsBinaryFmt | 0x02000020 |
| mtiSdfCharsBinaryFmtContextLen | 0x02000021 |
| mtiSdfCharsBinaryFmtContext | 0x02000022 |
| mtiMixedContent | 0x02000024 |
| PreDefFieldFav | 0x03000000 |
| PreDefField | 0x03000001 |
| mtiSelfDefField | 0x03000002 |
| PreDefFieldInstance | 0x03000004 |
| SelfDefFieldInstance | 0x03000008 |
| Namespace | 0x03000010 |
| mtiPreDefStructureV | 0x03000012 |
| mtiSelfDefStructureV | 0x03000014 |
| mtiStructureInstanceV | 0x03000016 |
| mtiSelfDefMessageV | 0x03000018 |
| mtiPreDefMessageV | 0x03000020 |
| mtiUnresolvedChoice | 0x04000001 |

## XMLNSC parser constants

See XMLNSC: Using field types for a list of the XMLNSC parser constants.

# Trace logging from a user-defined C extension

Message processing nodes and parsers that are written to the C programming language API can write entries to trace.

You can use two types of trace:
- **Service Trace**: entries usually describe what is happening within the code and are only useful to the owner of the code, such as the user-defined extension developer.
- **User Trace**: entries usually describe what is happening at an external level and are useful to the user of the code. Users of the code include message flow designers, and broker domain administrators.

For each trace type, there are three levels:
- None
- Normal
- Debug

For C user-defined extensions, the following utility functions are available for each trace type:

- **cciServiceTrace** and **cciUserTrace**: these functions write an entry to the respective trace type only when trace has been activated, that is, trace is at normal or debug level.
- **cciServiceDebugTrace** and **cciUserDebugTrace**: these functions write an entry to the respective trace type only when trace is active at debug level.

To help avoid making function calls in the case where no trace is written, the **cciIsTraceActive** utility function is provided. **cciIsTraceActive** reports whether trace is active and the level at which trace is active.

The **cci\*Trace** functions can be used by a user-defined extension regardless of the trace settings. The functions determine if trace is active and only write entries which are appropriate for the trace settings. When calling the **cci\*Trace** functions, some additional processing can be required. The **cciIsTraceActive** function is provided to allow the user-defined extension to query the trace settings and avoid this extra processing when trace is inactive.

In many cases, it is sufficient to treat the value returned from the **cciIsTraceActive** function as a Boolean value. If the returned value is non zero, trace is active at some level and it is appropriate to call any of the **cci\*Trace** functions. The returned value can also be inspected closely in the cases when details of the trace settings are required.

Trace settings can be changed at any time so it is advisable to query them regularly. For example, use **cciIsTraceActive** to query the trace settings when an implementation function is entered.

Trace entries can be associated with certain objects, which allows for further refinement of control for writing trace. A trace entry can be associated with a node or parser and trace is written according to the trace setting for that object. The object's trace setting is inherited from the message flow to which the node or parser belongs. If no object is specified, the trace is associated with the execution group.

# Multicultural support considerations for message catalogs

WebSphere Message Broker converts any message that is loaded from the listed supported code pages into the local code page of the running processes (brokers) before output to the syslog.

You must provide symbolic links to your primary message catalogs for all locales that you intend to support. WebSphere Message Broker uses the LC_MESSAGES variable when opening message catalogs.

## Multicultural support considerations on Windows

Windows When you build a message file for Windows that contains multiple locales, ensure that the locale on the computer is set to a western European locale (for example, English (United Kingdom)) before building the message catalogs. Use the chcp (Change Code Page) command to ensure that the code page is 850.

Write or convert all your message files (those with file type .mc) to the following code pages; each message file should be compiled separately by the message compiler with the additional flag that is specified in the following table.

DBCS message files do not need to be in Unicode (no **-U** flag). Use the RC command to 'resource compile' all the files, then use the link command to build a single message DLL.

| Locale | Code page | Additional Flags |
|---|---|---|
| English (United States) | 437 | -U |
| German (Standard) | 850 | -U |
| Spanish (Modern Sort) | 850 | -U |
| French (Standard) | 850 | -U |
| Italian (Standard) | 850 | -U |
| Portuguese (Brazilian) | 850 | -U |
| Japan | 932 | |
| Simplified Chinese (China) | 1381 | |
| Traditional Chinese (Taiwan) | 950 | |
| Korean | 949 | |

## Multicultural support considerations on Linux and UNIX

When you build message catalogs for Linux and UNIX systems, ensure that the catalogs are built in the code pages defined in the following table.

| Locale | Code page |
|---|---|
| English | 437 |
| German | 850 |
| Spanish | 850 |
| French | 850 |
| Italian | 850 |
| Portuguese (Brazilian) | 850 |
| Japan | 932 |
| Simplified Chinese (China) | 1381 |
| Traditional Chinese (Taiwan) | 950 |
| Korean | 949 |

## Multicultural support considerations on z/OS

z/OS When you build message catalogs for z/OS systems, ensure that the catalogs are built in the code pages defined in the following table.

| Locale | Code page |
|---|---|
| English | 1047 |
| Japan | 939 |
| Simplified Chinese (China) | 1388 |

# Part 3. Appendixes

# Appendix. Notices for WebSphere Message Broker

Read the legal notices for WebSphere Message Broker.

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 information 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 information. The furnishing of this information 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:

*Intellectual Property Licensing*
*Legal and Intellectual Property Law*
*IBM Japan Ltd.*
*1623-14, Shimotsuruma, Yamato-shi*
*Kanagawa 242-8502 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 information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information 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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 information 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 no 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 various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _*enter the year or years*_. All rights reserved.

## Trademarks in the WebSphere Message Broker Information Center

Review the trademark information for WebSphere Message Broker.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at `www.ibm.com/legal/copytrade.shtml`.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks of Intel Corporation in the United States and other countries.

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

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

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

# Index

**IBM** ®

Printed in USA