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 has been created from the WebSphere Message Broker Version 6.1 (December 2007) 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 these attempt to link to a PDF that is called after the topic identifier (for example, ac12340_.pdf) and therefore fail. 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

**1**

# 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 in conjunction with the nodes and parsers that are supplied with the product, and with third-party supplied nodes and parsers.

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. A user-defined extension can be either a node or a parser.

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

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 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): 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 sample

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

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 what you want it for. Most tasks can be performed using the functions already provided with WebSphere Message Broker, so 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 bear in mind 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 topics below 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" 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

This topic deals with issues relating to errors and exception handling that you need to consider when developing 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.

Correct handling of errors and exceptions is important for correct broker operation. You should be aware of this, and understand how and when your user-defined extension needs to handle errors and exceptions.

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

Utility functions, by convention, normally 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, then 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 below. The value of `returnCode` can always be tested safely to determine whether a utility function was successful.

If the invocation of a utility function causes the broker to generate an exception, this 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

This means that a user-defined extension would be unable to perform any of 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, you should 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 any 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 needed.

The user-defined extension can perform its own error recovery, if required, by setting a non-null value for returnCode. The utility function calls return to the user-defined extension and pass their status through returnCode. All exceptions occurring in any utility function must be passed back to the message broker for additional error recovery to be performed, that is, when CCI_EXCEPTION is returned in returnCode. You do this by invoking 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 message broker. Note that, similar to a 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 any 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 message 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 Broker 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 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 are generated by message parsers for errors which 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 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 are generated when a Throw node throws a user-defined exception.

**Database**

These 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 then the parsers are as well.

A user-defined extension must use this model. If a user-defined node requires global data or resources, then 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 "cniCreateNodeContext" on page 134.

## ODBC restrictions

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 182
- "cniSqlExecute" on page 185
- "cniSqlSelect" on page 187
- "cniSqlDeleteStatement" on page 184

**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, and ensure that you 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 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 12
- "Java user-defined input node life cycle" on page 14
- "C user-defined message processing nodes life cycle" on page 17
- "Java user-defined message processing nodes life cycle" on page 19
- "User-defined output node life cycle" on page 24
- "User-defined parser life cycle" on page 25

**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 then 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 similar to procedural programming. Nodes that you insert into a message flow are similar to 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 "cniCreateNodeContext" on page 134 for information about the cniEvaluate function.
  - 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:

# 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, and user-defined exits.

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

## User-defined node

User-defined nodes are the main mechanism for extending the functionality 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 24

**User-defined input nodes:**

A user-defined input node is an extension to the broker that provides a new input node in addition to those 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 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 third-party 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:* This topic guides you through the various stages in the life of a user-defined message processing node for the C programming language. It 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 recreated when the broker starts.
  - Execution group is being reloaded (user has run mqsireload). Any message flows that have been deployed previously are recreated 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 recreating 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 recreate 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 recreating them with the new configuration.

*Java user-defined message processing nodes life cycle:* This topic guides you through the various stages in the life of a user-defined message processing node for the Java programming language. It covers the objects that are created and destroyed, and the methods 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, from a message, you can use either type 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 makes itself known to the broker, or registers with the broker.

Whenever a broker starts, it loads all relevant LILs 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 broker's classpath.

*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's class is called.

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 in order 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 need only to create these output terminals by using your constructor class. However, if you need to initialize attribute values, you should 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 to where WebSphere Message Broker errors are propagated.

Make sure that any 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 will not attempt to handle the exception. If your message flow does not contain any method of exception handling, any 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 re-throw any exceptions that the message processing node cannot deal with. This will cause 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 need to include anything in your code to specify that 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 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 should 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 figure.
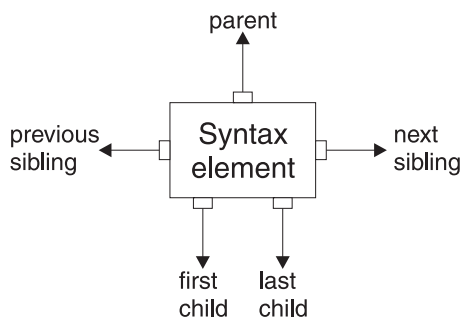


Other functions and methods support the manipulation of the elements themselves, with functions and methods to create elements, to set or query their values, to insert new elements into the tree, and to remove elements from the tree. See "C node utility functions" on page 117 and "C parser utility functions" on page 190, or the Javadoc information for more details.

The next figure describes a simple syntax element tree that shows a full range of interconnections between the elements.
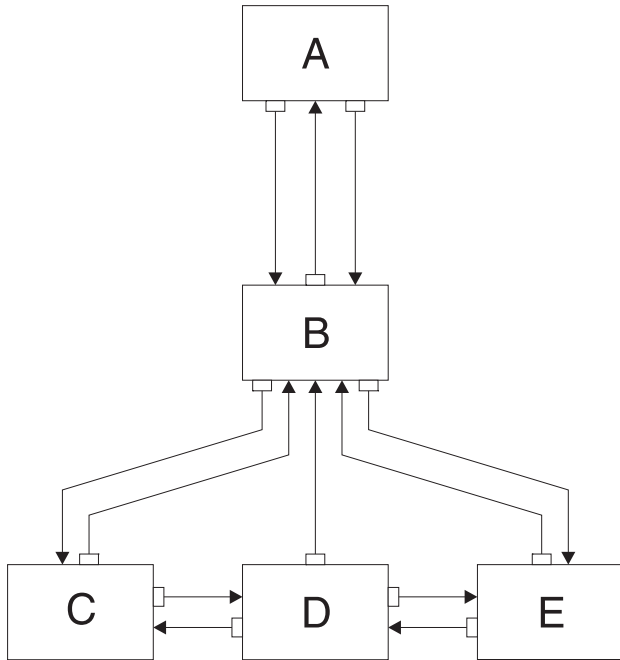
The element **A** is the root element of the tree. It has no parent because it is the root. It has a first child of element **B**. Because **A** has no other children, element **B** is also the last child of **A**.

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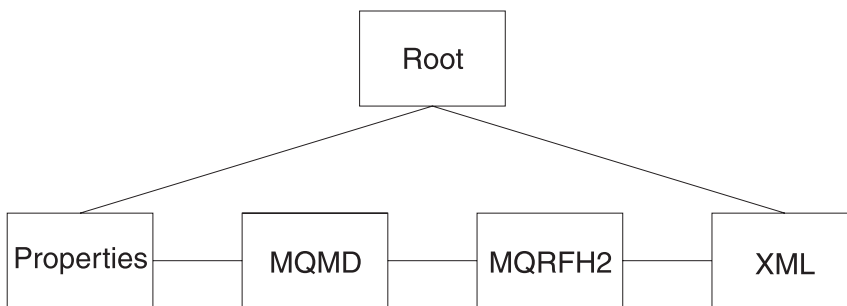


These elements at the first generation are often referred to as folders, in which syntax elements that represent message headers and message content data are stored. In this example, the first child of root is the Properties folder. The next sibling of Properties is the folder for the MQMD header. The 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 117 and "C parser utility functions" on page 190, 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 those supplied with the product.

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

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 on user-defined output nodes, read the topics that cover user-defined message processing nodes.

*User-defined output node life cycle:* For information on the life cycle of a user-defined output node, you should read the corresponding topics for user-defined message processing nodes.

The information in these topics 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.

*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 20.

## 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:** This topic introduces you to the concepts to consider before you develop a user-defined parser. When you are ready, use the instructions in "Creating a user-defined parser" on page 82 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 third-party 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 of 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 exploit 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 user exits can be invoked when one or more of the following events occurs:
- End of unit-of-work (UOW) or transaction (COMMIT or ROLLBACK)
- A message passing between two nodes
- A message dequeued from the input source



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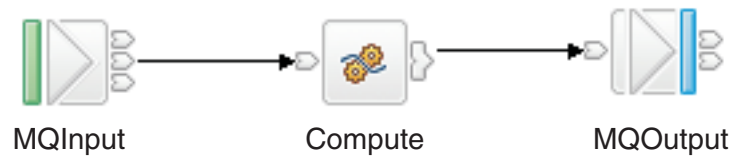
In the basic message flow shown here, you can track messages at two levels:
- Transaction level
- Node level

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

Therefore, you can track four different types of events, which occur in the following sequence:
1. Message dequeued from the input source (read into the flow)
2. Message propagated to the node for processing
3. Node processing completed
4. End of the transaction

## 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 113 and "Sample parser files" on page 115 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 71. For information on how to compile a C node or parser, see "Compiling a C user-defined extension" on page 51.
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 100
3. On the Windows computer that hosts the workbench, extract the SampleNodesProject.zip file, which is located in the *install_dir*\sample\ extensions\com.ibm.samples.nodes directory, and copy the resulting directory structure into the *install_dir*\eclipse\plugins directory.
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 Putting a test message).

   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:**
*   For a general introduction to user-defined extensions, read "User-defined extensions overview" on page 3.
*   Decide which type of user-defined extension to use.

To create a user-defined extension, follow the instructions in the appropriate topic:
*   "Implementing a user-defined node" on page 32
*   Implementing a user-defined parser
*   Implementing a user-defined exit

For user-defined nodes only, you must create a workbench Eclipse plug-in as well as the runtime .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 73. This step is not required for user-defined parsers or exits.

The following table shows the tasks that are involved in 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 71<br>2.  "Installing user-defined extension runtime files on a broker" on page 100<br>3.  "Creating the user interface representation of a user-defined node in the workbench" on page 73<br>4.  "Testing a user-defined node" on page 80 |
| To use one of the C sample nodes | 1.  "Compiling a C user-defined extension" on page 51<br>2.  "Installing user-defined extension runtime files on a broker" on page 100<br>3.  "Creating the user interface representation of a user-defined node in the workbench" on page 73<br>4.  "Testing a user-defined node" on page 80 |
| To use the sample parser | 1.  "Compiling a C user-defined extension" on page 51<br>2.  "Installing user-defined extension runtime files on a broker" on page 100 |
| To create your own Java node using the workbench | 1.  "Creating an input node in Java" on page 57 or "Creating a message processing or output node in Java" on page 62<br>2.  "Using event logging from a user-defined extension" on page 107<br>3.  "Compiling a Java user-defined node" on page 71<br>4.  "Testing a user-defined node" on page 80<br>5.  "Packaging a user-defined node workbench project" on page 101<br>6.  "Installing a user-defined extension to current and past versions of the broker" on page 102 |

| Action | Topics to view |
|---|---|
| To create your own C node | 1. "Creating an input node in C" on page 35 or "Creating a message processing or output node in C" on page 43 |
| | 2. "Using event logging from a user-defined extension" on page 107 |
| | 3. "Compiling a C user-defined extension" on page 51 |
| | 4. "Installing user-defined extension runtime files on a broker" on page 100 |
| | 5. "Creating the user interface representation of a user-defined node in the workbench" on page 73 |
| | 6. "Testing a user-defined node" on page 80 |
| | 7. "Packaging a user-defined node workbench project" on page 101 |
| | 8. "Installing a user-defined extension to current and past versions of the broker" on page 102 |
| To create your own parser | 1. "Creating a user-defined parser" on page 82 |
| | 2. "Using event logging from a user-defined extension" on page 107 |
| | 3. "Compiling a C user-defined extension" on page 51 |
| | 4. "Installing user-defined extension runtime files on a broker" on page 100 |
| To create a user exit | 1. "Developing a user exit" on page 94 |
| | 2. "Compiling a C user-defined extension" on page 51 |
| | 3. "Deploying a user exit" on page 94 |

## 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 node" on page 11

Decide which type of user-defined extension to use.

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 not available through the user-defined extension interface. The JVM initializes with various parameters that are specific to the broker's requirements. 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 a variety of 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"
3. "Packaging and distributing user-defined extensions" on page 95
4. "Testing a user-defined node" on page 80
5. "Packaging a user-defined node workbench project" on page 101

## 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.

The templates are designed to be a starting point to which new function can be added. When you design a new node, remember the following guidelines:

- Terminals on nodes follow the general pattern that is described in "Node templates" on page 301.
- Use panels and the properties that they contain in a way that is consistent with the description in "Panel definitions" on page 309.
- Do not design new terminals and panels if terminals and panels that offer the same functions already exist.

If you follow the design rules above, your user-defined nodes will look and behave like built-in nodes.

## Creating a user-defined node

**Before you start:**

Read "Designing a user-defined node."

You can write user-defined nodes in C or Java. When you have created a user-defined node, you can test it, as described in "Testing a user-defined node" on page 80. 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 95.

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 35
- "Creating a user-defined extension in Java" on page 56
- "Creating the user interface representation of a user-defined node in the workbench" on page 73

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 71<br>2. "Installing user-defined extension runtime files on a broker" on page 100<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 73<br>4. "Testing a user-defined node" on page 80 |
| To use one of the C sample nodes | 1. "Compiling a C user-defined extension" on page 51<br>2. "Installing user-defined extension runtime files on a broker" on page 100<br>3. "Creating the user interface representation of a user-defined node in the workbench" on page 73<br>4. "Testing a user-defined node" on page 80 |
| To create your own Java node using the workbench | 1. "Creating an input node in Java" on page 57 or "Creating a message processing or output node in Java" on page 62<br>2. "Using event logging from a user-defined extension" on page 107<br>3. "Compiling a Java user-defined node" on page 71<br>4. "Testing a user-defined node" on page 80<br>5. "Packaging a user-defined node workbench project" on page 101<br>6. "Installing a user-defined extension to current and past versions of the broker" on page 102 |

| 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 43 |
| | 2. "Using event logging from a user-defined extension" on page 107 |
| | 3. "Compiling a C user-defined extension" on page 51 |
| | 4. "Installing user-defined extension runtime files on a broker" on page 100 |
| | 5. "Creating the user interface representation of a user-defined node in the workbench" on page 73 |
| | 6. "Testing a user-defined node" on page 80 |
| | 7. "Packaging a user-defined node workbench project" on page 101 |
| | 8. "Installing a user-defined extension to current and past versions of the broker" on page 102 |

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

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

Complete one or more of the following steps to create user-defined extensions in C:
- "Creating an input node in C"
- "Creating a message processing or output node in C" on page 43
- "Creating a user-defined parser" on page 82
- "Compiling a C user-defined extension" on page 51

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 73
- "Testing a user-defined node" on page 80
- "Packaging and distributing user-defined extensions" on page 95

*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 116. 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 117.

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 37
3. "Setting attributes" on page 38
4. "Implementing the node functionality" on page 38
5. "Overriding the default message parser attributes (optional)" on page 39
6. "Deleting an instance of the node" on page 40

*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 should 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 node's cniRun function is called for each message that is propagated to the node. For the minimum code required to compile a C user-defined node, see the "C skeleton code" on page 289.

*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 should 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 289.

*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 listed below:

- 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 should take. If data is available for processing, the cniRun function should attempt to propagate the message. If no data is available for processing, the cniRun function should 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 primitive 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 those 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 certain parser deals with the start of the data. For example, the MQInput node only reads data from WebSphere MQ queues, so this data always has an MQMD at the beginning, and the MQInput node sets firstParserClassName to MQHMD. If your user-defined node always deals with data that begins with a structure that can be parsed by a certain parser, for example "MYPARSER", set firstParserClassName to MYPARSER as follows:

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 should call the cniDeleteNodeContext function to free any used memory and release any 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 35.

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 42

*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 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 24

When you code a message processing node or an output node, the nodes provide essentially the same services. You can perform message processing within an output node, and you can output a message to a bit stream using a message processing node. For simplicity, this topic refers mainly to the node as a message processing node but it does also discuss the functionality 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 116. You can call C node utility functions, implemented in the runtime broker, to help with the node operation; for more information, see "C node utility functions" on page 117.

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 sample

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

To create either type of node:
1. "Declaring and defining your node"
2. "Creating an instance of the node" on page 45
3. "Setting attributes" on page 47
4. "Implementing the node functionality" on page 47
5. "Deleting an instance of the node" on page 48

*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 289.

*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 113. 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 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 289.

*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**

Read the topic "Creating a message processing or output node in C" on page 43.

After you have created a user-defined node, the following options are available:
1. "Accessing message data"
2. "Transforming a message object" on page 49
3. "Accessing ESQL" on page 49
4. "Propagating a message" on page 50
5. "Writing to an output device" on page 50

*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 "C node utility functions" on page 117 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
140 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**

You must have a user-defined extension that is written in C or C++. This extension
can be one of the provided sample nodes described in "Sample node files" on page
113, the sample parser described in "Sample parser files" on page 115, or a node or
parser that you have created yourself using the instructions in "Creating a message
processing or output node in C" on page 43, "Creating an input node in C" on
page 35, or "Creating a user-defined parser" on page 82.

The file names used in these instructions are those of the supplied samples. If you
are compiling your own user-defined extensions, substitute your own file names.

You do not have to modify user-defined extension libraries that are provided by
software vendors, but you do have to import or create the associated workbench
resources.

When you compile a user-defined extension that is written in C or 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

*Compilation:*

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

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

*Compiling on AIX:*

 AIX  Compile and link the user-defined extension using a supported C compiler:

```
xlc_r \
   -I. \
   -I /opt/IBM/mqsi/6.0/include/plugin \
   -c SwitchNode.c \
   -o SwitchNode.o

xlc_r \
   -I. \
   -I /opt/IBM/mqsi/6.0/include/plugin \
   -c BipSampPluginUtil.c \
   -o BipSampPluginUtil.o

xlc_r \
   -I. \
   -I /opt/IBM/mqsi/6.0/include/plugin \
   -c Common.c \
   -o Common.o

xlc_r \
   -I. \
   -I /opt/IBM/mqsi/6.0/include/plugin \
   -c NodeFactory.c \
   -o NodeFactory.o

xlc_r -qmkshrobj \
      -bM:SRE \
      -bexpall \
      -bnoentry \
      -o SwitchNode.lil SwitchNode.o BipSampPluginUtil.o Common.o NodeFactory.o \
      -L /opt/IBM/mqsi/6.0/lib
      -l imbdfplg

chmod a+r SwitchNode.lil
```

*Compiling on HP-UX:*

 HP-UX  Compile and link the user-defined extension using a supported C compiler:

```
cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
```

```
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc +z \
-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/PA_RISC2.0 \
   -L install_dir/jre/lib/PA_RISC2.0/server \
   -l imbdfplg

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

*Compiling on HP-UX (Integrity platform):*

HP-UX   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.

```
aCC -O +DD64 -AA -mt +z\
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

aCC -O +DD64 -AA -mt +z\
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c Common.c \
```

```
    -o output_dir/Common.o

aCC -O +DD64 -AA -mt +z\
-I. \
    -I<install_dir>/include \
    -I<install_dir>/include/plugin \
    -c NodeFactory.c \
    -o output_dir/NodeFactory.o

aCC -O +DD64 -AA -mt +z\
-I. \
    -I<install_dir>/include \
    -I<install_dir>/include/plugin \
    -c SwitchNode.c \
    -o output_dir/SwitchNode.o

aCC -O +DD64 -AA -mt +z\
-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/PA_RISC2.0 \
    -L install_dir/jre/lib/PA_RISC2.0/server \
    -l imbdfplg

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

*Compiling on Linux:*

Linux    Compile and link the user-defined extension using a supported C
compiler. Enter each command as a single line of input; in the following examples
the lines have been split to improve readability.

```
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE TransformNode.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE SwitchNode.c /usr/bin/gcc -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE BipSampPluginUtil.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE Common.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE NodeFactory.c
/usr/bin/g++ -o samples.lil
  TransformNode.o SwitchNode.o BipSampPluginUtil.o Common.o NodeFactory.o
    -shared -lc -lnsl -ldl -L/opt/mqsi/lib -limbdfplg
```

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

*Compiling on Solaris:*

Solaris Compile and link the user-defined extension using a supported C compiler:

```
cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

cc -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 imbdfplg

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

*Compiling on Windows:*

Windows Compile and link the user-defined extension using 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 /I. /I..\..\..\include\plugin SwitchNode.c
BipSampPluginUtil.c Common.c NodeFactory.c TransformNode.c -link
/DLL ..\..\..\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:*

z/OS 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
```

Compile and link the user-defined extension using a supported C compiler. 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\(./SwitchNode.lst\) \
 -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\(./SwitchNode.lst\) \
 -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\(./SwitchNode.lst\) \
 -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
```

Issue 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:**

Complete one or more of the following steps to create user-defined nodes in Java:
- "Creating an input node in Java" on page 57
- "Creating a message processing or output node in Java" on page 62
- "Compiling a Java user-defined node" on page 71

- "Packaging a Java user-defined node" on page 96

You can write only user-defined nodes in Java: user-defined parsers must be written 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 73
- "Testing a user-defined node" on page 80
- "Packaging and distributing user-defined extensions" on page 95

*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 new Java project"
 2. "Declaring the input node class" on page 58
 3. "Defining the node constructor" on page 58
 4. "Receiving external data into a buffer" on page 58
 5. "Propagating the message" on page 59
 6. "Controlling threading and transactionality" on page 59
 7. "Declaring the node name" on page 60
 8. "Declaring attributes" on page 61
 9. "Implementing the node functionality" on page 61
10. "Overriding default message parser attributes (optional)" on page 61
11. "Deleting an instance of the node" on page 62

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 in this project.

*Declaring the input node class:*

Any class that implements MbInputNodeInterface and is contained in the broker's LIL path is registered with the broker as an input node. When you implement MbInputNodeInterface, you also need to 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 user's node class is called. This class is where you create the terminals of the node, and initialize any 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 any other Java program, provided that 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 using the createMessage method of the MbInputNode class, and then generate a valid message assembly from this message. For details of these methods, see the Java user-defined node API. For example, to read the input 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 new 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 node's output terminals.

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 final `try...catch` block.

*Controlling threading and transactionality:*

The broker infrastructure handles transaction issues such as controlling the commit of any 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 any unhandled exceptions, and rolls back the transaction.

You determine the behavior of transactions and threads 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 shutdown 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 should return with the TIMEOUT return code. Assuming that the broker does not need to reconfigure or shutdown, the input node's run method gets called again straight away.

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 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 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 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 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:*

As already described, the run method is called by the broker to create the input message. This method should 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 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 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 those 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 containing the node, and redeploy the configuration.

When the node is deleted, it can perform any 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 24
- "Restrictions when creating Java nodes" on page 57

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. For further details of the Java user-defined node API, see Java user-defined node API.

Complete the following steps:
1. "Creating a new Java project"
2. "Declaring the message processing node class"
3. "Defining the node constructor" on page 64
4. "Accessing message data" on page 64
5. "Transforming a message object" on page 65
6. "Propagating the message" on page 66
7. "Declaring the node name" on page 66
8. "Declaring attributes" on page 66
9. "Implementing the node functionality" on page 67
10. "Deleting an instance of the node" on page 68

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 final try/catch block to clear the
memory that is allocated for the message tree.

*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 62.

You can add one or more of the following functions:
• "Accessing ESQL"
• "Interacting with databases" on page 69
• "Handling exceptions" on page 69
• "Writing to an output device" on page 70

*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 type of transaction you want from the following:

**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(dataSource, 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 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 could 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 MQInput 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:
- 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 and MRM parser constants" on page 296.

*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 113, or a node that you have created yourself using the instructions in either "Creating a message processing or output node in Java" on page 62 or "Creating an input node in Java" on page 57.

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 below.

When you compile a Java user-defined node from the command line on any platform, you need a compatible IBM Software Developer Kit for Java. 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 checked.
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 created .jar file appears in your node project, and is ready for you to install in a broker domain (see "Installing user-defined extension runtime files on a broker" on page 100) or to package for distribution (see "Packaging a user-defined node workbench project" on page 101).

*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. The location of the jplugin2.jar file for each platform is shown below:

   **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 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 96.

The .par file that you have created is ready for you to install on a broker domain (see "Installing user-defined extension runtime files on a broker" on page 100) or to package for distribution (see "Packaging a user-defined node workbench project" on page 101).

**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 74

For user-defined parsers, just 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.

*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 is displayed.
3. Specify the name of the category for the nodes that you are creating. The node appears 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 is displayed.
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 organization's Internet domain name as part of the name. For example, the name should be of the form *com.xyz.nodegroup*, where *com.xyz* is the company Internet domain name.

   Do not enter a blank space and/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**.

   Ensure that the Plug-in ID field does not contain an underscore character. This character is not valid. The plug-in content panel of the New Plug-in Project window is displayed.
5. Accept all default values and click **Next**. The Templates panel of the New Plug-in Project window is displayed.
6. Click **Finish**.
7. If any warnings are displayed in the Tasks 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 **Not In Workspace** 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 Project**.

A project folder containing all the supporting files that are needed 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:*

Follow these steps to create the representation of a user-defined node in the workbench.

**Before you start:**

Create a user-defined node project.

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"
3. Optional: "Adding help to the node" on page 77
4. Optional: "Creating node icons" on page 78
5. Optional: "Adding a property editor or compiler" on page 78

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

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

**Before you start:**

Complete the task "Creating a user-defined node project" on page 73.

1. Switch to the Broker Application Development perspective.
2. Launch the wizard: click **File** → **New** → **Other...**. The **New** window opens.
3. Expand **Message Flow Node Development** and 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 have 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.

*Defining the node properties:*

Define the properties for a user-defined node and add input and output terminals.

**Before you start:**

Complete the following tasks:
1. "Creating a user-defined node project" on page 73
2. "Creating the user-defined node plug-in files" on page 74

When you complete the task described in "Creating the user-defined node plug-in files" on page 74, 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 add the node's properties: for example, a database name, a host server name, or a password. The properties that you set here must match the properties that are specified in the user-defined node itself 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 defaults** if you want the node to initialize with the broker's default values.
3. Optional: A page of properties is already created under the properties hierarchy, and is named Basic. To create additional pages of properties, click **Add**

   **Property Group**  .

   When a new group is added to the hierarchy, the name is highlighted so that you can change it.
4. To add a scalar property, click the name of a property group in the hierarchy to

   highlight it, and click **Add Scalar 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 will be shown in the Properties view when the node is included in a message flow.

c. To generate a property editor or a compiler, specify the location of these resources in the relevant field.

d. 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 complex property, click the name of a property group in the hierarchy

   to highlight it, and click **Add Complex Property**  .

   A complex property is a property to which you can assign multiple values. Complex properties are displayed in a table in the Properties view, where you can add, edit, and delete values, and change the order of the values in the table.

   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 complex property.

   a. Specify a custom compiler for the complex property.

   b. Enter the name of the custom property editor for the complex property, or select the name of one of the built-in editors from the drop-down list.
      - Default
      - XPath Property editor: that renders XPath expressions.
      - All Output Terminals Property editor: renders all output terminals that are associated with the node.
      - All Input Terminals Property editor: renders all input terminals that are associated with the node.
      - Dynamic Output Terminals Property editor: renders all dynamic output terminals that are associated with the node.
      - Dynamic Input Terminals Property editor: renders all dynamic input terminals that are associated with the node.

   c. Specify the system property:
      - 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, so you cannot add, edit or delete a row in the Message Flow editor.
      - Mandatory: the table must contain at least one row.

   d. To add a column to a complex property, click the complex property in the properties hierarchy to highlight it, and click **Add Scalar Property**.

   The scalar properties that you add are displayed as children of the complex property in the hierarchy. Define their properties in the Details section, as described in step 5.

6. Optional: Drag the properties in the properties hierarchy to change the order in which they are listed on the properties page.

7. Close the *nodename*.msgnode file.

8. Optional: You can customize the text that appears 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 78
- "Adding a property editor or compiler" on page 78

You can now test your node. Select **Run** → **Run as** → **Runtime Workbench** to launch another instance of the workbench. See "Enabling PDE runtime capabilities" on page 82, and the PDE Guide, for more information about testing using the Runtime Workbench.

*Adding help to the node:*

You can add help information for the node that you have created.

**Before you start:**

Complete the following tasks:
1. "Creating a user-defined node project" on page 73
2. "Creating the user-defined node plug-in files" on page 74
3. "Defining the node properties" on page 74

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 hover your mouse over the node.

All three forms of help are optional; you can create any one or more of the three resources described below.
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 node's online help appear 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 appear 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 table of contents:

   ```
   <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 table of contents 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 on 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 id 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 node's online help, described above. Code the link as shown below:

```
<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 your node's hover help, which shows the node name when the palette is not wide enough to display it all.

If you do not want to add any of the optional features, you can test your node at this point. Launch another instance of the workbench; see "Enabling PDE runtime capabilities" on page 82, and select **Run** → **Run as** → **Runtime Workbench** for further guidance. See the PDE Guide for more information about testing using the Runtime Workbench.

*Creating node icons:*

**Before you start:**

You must complete the following tasks:
1. "Creating a user-defined node project" on page 73
2. "Creating the user-defined node plug-in files" on page 74
3. "Defining the node properties" on page 74

When a node is created, a set of default icons are created; 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.

To test your node, launch another instance of the workbench (see "Enabling PDE runtime capabilities" on page 82) and select **Run** → **Run as** → **Runtime Workbench**. See the PDE Guide for more information about using the Runtime Workbench for testing your node.

*Adding a property editor or compiler:*

**Before you start:**

You must complete the following tasks:
1. "Creating a user-defined node project" on page 73
2. "Creating the user-defined node plug-in files" on page 74
3. "Defining the node properties" on page 74

To control how the properties of your node are displayed, create a property editor by using the IPropertyEditor interface. A property editor can contain many controls, such as text fields and lists.

To create a custom compiler, for example to encrypt a value before sending it to the server, use the IPropertyCompiler interface.

*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, as follows:
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 just 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, launch another instance of the workbench (see "Enabling PDE runtime capabilities" on page 82) and select **Run** → **Run as** → **Runtime Workbench**. See the PDE Guide for more information about testing using the Runtime Workbench for testing your node.

To test your compiler, deploy to a broker the flow that contains your user-defined node.

## Testing a user-defined node

When you have created and installed the required resources, you can test your user-defined node.

**Before you start**

Ccomplete the following tasks:
- "Creating a user-defined extension in C" on page 35 or "Creating a user-defined extension in Java" on page 56
- "Creating the user interface representation of a user-defined node in the workbench" on page 73
- "Installing user-defined extension runtime files on a broker" on page 100

1. Enable the Eclipse Plug-in Development environment. This task is described in "Enabling PDE runtime capabilities" on page 82. For more information about the PDE and the Plug-in Development Perspective, see the PDE Guide.
2. Click **Run** → **Run as** → **Runtime Workbench** to start a new copy of the workbench that includes your new nodes.
3. Open the Message Flow editor. Your new nodes appear in the node palette.
4. Create a message flow that includes your node. Read Adding a message flow node for guidance on how to complete this task.
5. Deploy the message flow to a broker. This task is described in 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. See Resolving problems with user-defined extensions for a description of some common problems and their solutions.
   b. Check the event log. Details are provided in Event Log editor.
   c. Write entries to the event log from your node. See "Using event logging from a user-defined extension" on page 107 for more information.
   d. Switch on user trace at debug level. See Using trace for details of how to complete this task.

      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 invoked. 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 invoking the Java `evaluate()` method of a user-defined node.

- BIP3905: a message traced before invoking the C `cniEvaluate()` implementation function (iFpEvaluate member of CNI_VFT) of a user-defined node.

- BIP4142: a debug message that is traced when invoking 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 invoked by a user-defined extension, can modify the internal state of a message broker's object. 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 invoking 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 a broker's object.

- 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.

For information on the C language user-defined node API, see "C language user-defined parser API" on page 189 and "C language user-defined node API" on page 116. For information on the Java user-defined node API, see Java user-defined node API.

e. Add a Trace node to your message flow, and check the output that is generated. See the Trace node for further information.

f. Use the flow debugger to debug the flow that contains your node. Start with 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 101). Until you do this, the new nodes are available only in your test workbench session on your local system.

**Enabling PDE runtime capabilities:**

To access PDE Runtime facilities you must first enable the PDE capabilities in your workbench.

To enable the PDE capabilities:
1. Click **Window> Preferences** to open the Preferences window.
2. Expand **Workbench** in the left hand pane, and click **Capabilities**.
3. Expand **Eclipse Developer** in the Capabilities pane.
4. Select the **Eclipse Plug-in Development** check box.
5. Click **OK** or **Apply** to apply your changes.

The PDE and PDE runtime views are now available in the Message Brokers Toolkit.

# Creating 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 25

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 have to write are listed in "C parser implementation functions" on page 189. The utility functions that are provided by WebSphere Message Broker to help you are listed in "C parser utility functions" on page 190.

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" on page 83
2. "Creating an instance of the parser" on page 84
3. "Deleting an instance of the parser" on page 84

## 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 should 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 invoking
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 should allocate 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:
- "Creating a user-defined parser" on page 82

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 86
- "Implementing output functions" on page 86
- "Implementing a message header parser" on page 87

**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 219
- "cpiParseBufferEncoded" on page 220
- "cpiParseBufferFormatted" on page 221

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 241
- "cpiWriteBufferEncoded" on page 242
- "cpiWriteBufferFormatted" on page 243

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**

You must have a user-defined extension that is written in C or C++. This extension
can be one of the provided sample nodes described in "Sample node files" on page
113, the sample parser described in "Sample parser files" on page 115, or a node or
parser that you have created yourself using the instructions in "Creating a message
processing or output node in C" on page 43, "Creating an input node in C" on
page 35, or "Creating a user-defined parser" on page 82.

The file names used in these instructions are those of the supplied samples. If you
are compiling your own user-defined extensions, substitute your own file names.

You do not have to modify user-defined extension libraries that are provided by
software vendors, but you do have to import or create the associated workbench
resources.

When you compile a user-defined extension that is written in C or 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

**Compilation:**

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

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

**Compiling on AIX:**

AIX Compile and link the user-defined extension using a supported C compiler:

```
xlc_r \
    -I. \
    -I /opt/IBM/mqsi/6.0/include/plugin \
    -c SwitchNode.c \
    -o SwitchNode.o

xlc_r \
    -I. \
    -I /opt/IBM/mqsi/6.0/include/plugin \
    -c BipSampPluginUtil.c \
    -o BipSampPluginUtil.o

xlc_r \
    -I. \
    -I /opt/IBM/mqsi/6.0/include/plugin \
    -c Common.c \
    -o Common.o

xlc_r \
    -I. \
    -I /opt/IBM/mqsi/6.0/include/plugin \
    -c NodeFactory.c \
    -o NodeFactory.o

xlc_r -qmkshrobj \
      -bM:SRE \
      -bexpall \
      -bnoentry \
      -o SwitchNode.lil SwitchNode.o BipSampPluginUtil.o Common.o NodeFactory.o \
      -L /opt/IBM/mqsi/6.0/lib
      -l imbdfplg

chmod a+r SwitchNode.lil
```

**Compiling on HP-UX:**

HP-UX Compile and link the user-defined extension using a supported C compiler:

```
cc +z \
-I. \
    -I<install_dir>/include \
    -I<install_dir>/include/plugin \
    -c BipSampPluginUtil.c \
    -o output_dir/BipSampPluginUtil.o

cc +z \
-I. \
    -I<install_dir>/include \
    -I<install_dir>/include/plugin \
```

Developing user-defined extensions **89**

```
   -c Common.c \
   -o output_dir/Common.o

cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc +z \
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc +z \
-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/PA_RISC2.0 \
   -L install_dir/jre/lib/PA_RISC2.0/server \
   -l imbdfplg

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

**Compiling on HP-UX (Integrity platform):**

HP-UX   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.

```
aCC -O +DD64 -AA -mt +z\
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

aCC -O +DD64 -AA -mt +z\
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

aCC -O +DD64 -AA -mt +z\
-I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c NodeFactory.c \
```

```
        -o output_dir/NodeFactory.o

aCC -O +DD64 -AA -mt +z\
-I. \
    -I<install_dir>/include \
    -I<install_dir>/include/plugin \
    -c SwitchNode.c \
    -o output_dir/SwitchNode.o

aCC -O +DD64 -AA -mt +z\
-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/PA_RISC2.0 \
    -L install_dir/jre/lib/PA_RISC2.0/server \
    -l imbdfplg

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

**Compiling on Linux:**

`Linux`  Compile and link the user-defined extension using a supported C
compiler. Enter each command as a single line of input; in the following examples
the lines have been split to improve readability.

```
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE TransformNode.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE SwitchNode.c /usr/bin/gcc -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE BipSampPluginUtil.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE Common.c
/usr/bin/g++ -c -fpic -MD -trigraphs  -I. -I/opt/mqsi/include
    -I/opt/mqsi/include/plugin -DLINUX -D__USE_GNU
    -D_GNU_SOURCE NodeFactory.c
/usr/bin/g++ -o samples.lil
  TransformNode.o SwitchNode.o BipSampPluginUtil.o Common.o NodeFactory.o
    -shared -lc -lnsl -ldl -L/opt/mqsi/lib -limbdfplg
```

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

**Compiling on Solaris:**

`Solaris`  Compile and link the user-defined extension using a supported C
compiler:

```
cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c SwitchNode.c \
   -o output_dir/SwitchNode.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c BipSampPluginUtil.c \
   -o output_dir/BipSampPluginUtil.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c NodeFactory.c \
   -o output_dir/NodeFactory.o

cc -mt \
   -I. \
   -I<install_dir>/include \
   -I<install_dir>/include/plugin \
   -c Common.c \
   -o output_dir/Common.o

cc -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 imbdfplg

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

**Compiling on Windows:**

Windows  Compile and link the user-defined extension using 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 /I. /I..\..\..\include\plugin SwitchNode.c
BipSampPluginUtil.c Common.c NodeFactory.c TransformNode.c -link
/DLL ..\..\..\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:**

z/OS  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
```

Compile and link the user-defined extension using a supported C compiler. 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\(./SwitchNode.lst\) \
 -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\(./SwitchNode.lst\) \
 -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\(./SwitchNode.lst\) \
 -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
```

Issue 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 exit

You can develop and deploy a user-defined exit.

**Before you start:**

- Read "User exits" on page 28.

Follow the steps in the topics below to develop and deploy a user-defined exit.

1. "Developing a user exit" on page 94

2. "Deploying a user exit"

## Developing a user exit

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 |
|---|---|
| Message dequeued from input source | cciInputMessageCallback |
| Message propagated to node for processing | cciPropagatedMessageCallback |
| Node completed processing | cciNodeCompletionCallback |
| Transaction ended | 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 51 for more details.

## Deploying a user exit

**Before you start:**

• Write and compile the user exit code. See "Developing a user exit."

• 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 containing user exits:

- The broker property **UserExitPath** (**UserExitPath64** for a 64-bit execution group). This property defines a list of directories separated by colons (semi-colons on Windows). Use the –x flag on mqsicreatebroker or mqsichangebroker to set this property for specific brokers.
- The environment variable **MQSI_USER_EXIT_PATH** (**MQSI_USER_EXIT_PATH64** for a 64-bit execution group) is a list of directories separated by colons (semi-colons on Windows). This environment variable is typically set to load the user exit into every broker for a specific environment.

If both are set, the environment variable takes precedence. All directories in the environment variable are searched, in the order in which they appear in the variable. Then all of the directories in the broker property are searched in the order in which they appear in the property.

2. Load the user exit library into the broker's processes.

   When the user exit library has been installed on the broker, it must be loaded. Do this in either of the following ways:
   - Stop and restart the broker.
   - Issue the mqsireload command. This causes the execution group processes to be restarted.

3. Activate the user exit.

   User exits can be active or inactive, and are inactive by default. Using the mqsichangeflowuserexits command, you can change the state of a user exit dynamically 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; this does not require the broker to be restarted.

   To set the default user exit state for a broker:

   a. Stop the broker.

   b. Set the **activeUserExits** property of the broker 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 issue the mqsireload command or restart the broker in order to load an exit from the library.
      - Issue the mqsichangeflowuserexits command in order 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 given flow, the broker invokes them in the order defined by mqsichangeflowuserexits.

# 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 31
- "Testing a user-defined node" on page 80

When you have created and tested your user-defined extension, you can distribute these resources to other computers in your broker domain:

- Copy the files generated by the compilation step to all the computers on which you have created brokers that might need these resources. This task is described in "Installing user-defined extension runtime files on a broker" on page 100. For a more automated approach, see the information in "Installing a user-defined extension to current and past versions of the broker" on page 102.

- Package the resources that make up the workbench representation of your user-defined node to create an Eclipse plug-in. This task is described in "Creating the user interface representation of a user-defined node in the workbench" on page 73. Then install the plug-in on all the computers on which your workbench users might need to use them, following the instructions in "Installing a user-defined extension to current and past versions of the broker" on page 102. 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 113, or a node that you have created yourself using the instructions in either "Creating a message processing or output node in Java" on page 62 or "Creating an input node in Java" on page 57.

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 100.

- **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" 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 100.

**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. The classes that are placed in the JAR override any classes that are in the shared classes directory or the CLASSPATH environment variable. If the deployed PAR contains more than one node type, the nodes share the same classloader. Therefore, a set of user-defined nodes that share static data should 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, so you can use this information to check that the correct classes are being loaded.

The broker uses the following classloader tree:

**Java user-defined nodes**
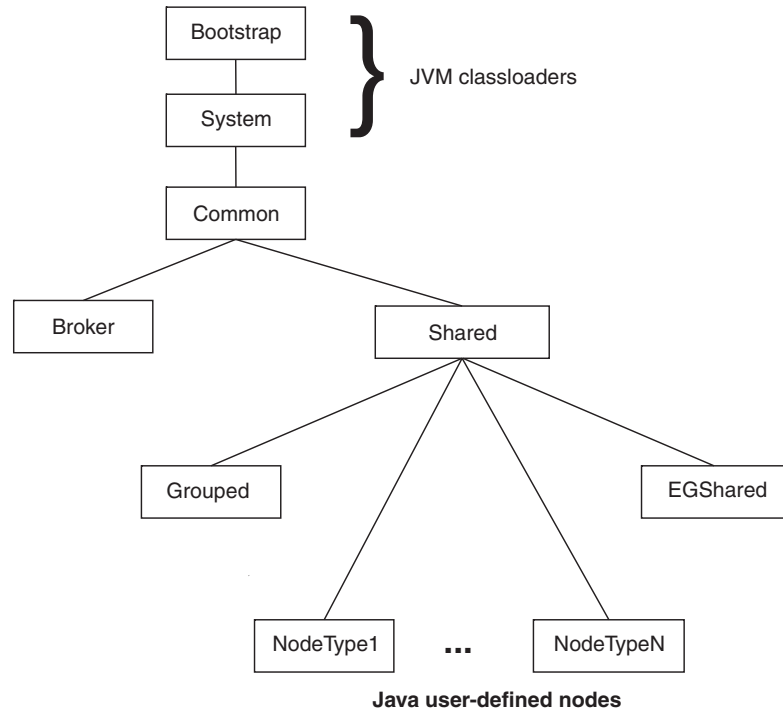
The following describes the components in the classloader tree:

- **Common classloader**: This 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 loads the broker internal classes. These classes can not be accessed by user classes.
- **Shared classloader** : This loads classes from JAR files that have been placed in the `<WorkPath>/shared-classes/` directory and the CLASSPATH environment variable. These are classes that are available to all Java user-defined nodes and JavaCompute nodes within the broker.

  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. The following should not be placed in this directory:

  - User-defined nodes
  - Classes, which have a dependency on other classes that have been deployed with a user-defined node.

- **Grouped classloader**: This loads all user-defined nodes that are packaged as JAR files. User-defined nodes that have been packaged for previous versions of WebSphere Message Broker will be loaded using this loader. User-defined nodes that are packaged in JAR files are loaded into one loader, and therefore can share static data.
- **EGShared classloader**: This loads classes from JAR files that have been deployed to the broker runtime in a BAR file. This is used to support 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. This allows the JavaCompute node to reload modified versions of the same class without needing to restart the broker.

*User-defined nodes classloading search paths:*

**User-defined nodes package in a PAR**

The broker uses the following search path to find user-defined node classes:
1. `/classes` to locate classes in the deployed PAR.
2. `/lib` to locate any JAR files in the deployed PAR.
3. `<WorkPath>/shared-classes/` to locate any JAR files.
4. `CLASSPATH` environment variable.

**User-defined nodes package in a JAR**

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.

*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 71 or "Compiling a C user-defined extension" on page 51.
  - 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 73.

This task instructs you to stop and restart brokers. This action is required in all but the two circumstances described in step 4 below, 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 haven't already got 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 could 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 chosen 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 chosen 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 above steps for every broker that needs the user-defined extension file. If all of your brokers are on the same machine 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 machine type.

   <span style="border:1px solid; padding:2px">Windows</span> On Windows, the .pdb file provides symbolic information that is used when displaying stack diagnostic information 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 invokes 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 refer to node and parser factory behavior. Java users do not need 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 in C.
2. Create the representation of your user-defined node in the workbench.
3. Test your user-defined node.

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 35 or "Creating a user-defined extension in Java" on page 56.
- 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 73.

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, generate a JAR file that contains your node's plug-in project as follows:

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, and any C source code or compiled files, to the .zip file using any file compression utility.

To distribute the workbench files, continue with "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 100.

For installation on another system, see "Installing a user-defined extension to current and past versions of the broker."

To distribute your node commercially, see the PDE Guide for information about issues such as versioning and updating your user-defined node.

## Installing a user-defined extension to current and past versions of the broker

The task described here is for users, for example for third-party vendors, who want to install user-defined extensions with the minimum of user intervention.

**Before you start**

Complete the following tasks:

1. "Compiling a Java user-defined node" on page 71 or "Compiling a C user-defined extension" on page 51

2. "Creating the user interface representation of a user-defined node in the workbench" on page 73
3. "Testing a user-defined node" on page 80
4. "Packaging a user-defined node workbench project" on page 101

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.

**The WebSphere Message Broker Toolkit installation:**

Before installing a user-defined node, establish the version of the Message Broker Toolkit to which you are installing, because a specific version of the toolkit could be a prerequisite of the user-defined extension, or it might require specific files to run.

To determine the toolkit version, see "Detecting installed versions of WebSphere Message Broker" on page 105.
1. To install your user-defined node on Message Broker Toolkit Version 6.1, place the JAR file that contains your node plug-in project into the plugins directory of the Message Broker Toolkit installation. For example, C:\Program Files\IBM\WMBT610\plugins.
2. Restart your workbench session for the changes to take effect.

**Runtime installation:**

You might need 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 105.

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 100.

**Single broker extension:**

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. For more information, see mqsicreatebroker command and mqsichangebroker command.

You cannot use the -l parameter to modify the user LILPATH for 64-bit extensions. Instead, append the directory containing the directory that holds the extension files to the environment variable MQSI_LILPATH64, as described in the following section.

**Multiple brokers extension:**

To affect all brokers on a system, you must 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).

You can do this by creating a custom environment script in %ALLUSERSPROFILE%\Application Data\IBM\MQSI\common\profiles on Windows, or /var/mqsi/common/profiles on UNIX and Linux systems. 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 should 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 should search for additional classes

**NLSPATH**
    Defines the location of message catalogues

**PATH**
    Defines the location of executable files. On Windows, this variable also defines the location of dependent libraries.

**LIBPATH / SHLIB_PATH / LD_LIBRARY_PATH**
    Defines the location of dependent libraries on UNIX and Linux systems.

**Example script:**

`Windows` Environment profile for MyExtension, installed in C:\Program Files\MyExtensions. The script is called C:\Documents and Settings\All Users\Application Data\IBM\MQSI\common\profiles\MyExtension.cmd:

```
REM Added by MyExtension install, do not modify
set MQSI_LILPATH=%MQSI_LILPATH%;"C:\Program Files\MyExtension\bin"
```

`UNIX` Environment profile for MyExtension, installed in /opt/MyExtension. The script is called /var/mqsi/common/profiles/MyExtension.sh:

```
#!/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 only runs 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:**

Detect the version of Message Broker using registry entries.

You can include, as part of your user-defined extension, code to detect the version of WebSphere Message Broker that is installed on your computer.

*Detecting installed versions on Windows:*

You can 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 6.0 runtime components**
```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
mqsi60\DisplayVersion = 6.x.x.x
```

**WebSphere Message Broker Version 6.1 toolkit**

> Check for the presence of the files \Application Data\IBM\Installation Manager\installed.xml and \Application Data\IBM\Installation Manager\installRegistry.xml in your home directory. (The default home directory is C:\Documents and Settings\All Users: this setting might have a different value on your system.)

**WebSphere Message Broker Version 6.1 runtime components**
```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
mqsi61\DisplayVersion = 6.x.x.x
```

*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 files listed below 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 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 Version 6.1 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 by completing the
following steps.

1. Stop the broker using the mqsistop command.
2. Update or overwrite the .lil or .jar file.
3. Start the broker using the mqsistart command.

In two situations it is not necessary to stop and start the broker:

- If 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 something 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 of the file lock that
  is put in place by the operating system.

These two approaches should 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 something is first deployed to it.
If you restart the broker, you 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 situations 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.

## Deleting a user-defined extension from the broker

Remove a user-defined extension file from the broker.

1. Stop the broker, using the mqsistop command.
2. Delete the .lil or .jar file from the appropriate directory.

   For C user-defined extensions:

| Platform | Location |
|---|---|
| Windows  Windows | *install_dir*\bin |
| Linux  Linux | *install_dir*/lil |
| UNIX  UNIX | *install_dir*/lil |
| z/OS  z/OS | *install_dir*/lil |

   For Java user-defined nodes:

| Platform | Location |
|---|---|
| Windows  Windows | *install_dir*\jplugin |
| Linux  Linux | *install_dir*/jplugin |
| UNIX  UNIX | *install_dir*/jplugin |
| z/OS  z/OS | *install_dir*/lil |

3. Restart the broker using the mqsistart command.

## Using event 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 choose to write significant events, error or otherwise, for problem determination and operational purposes in the same way that product components do.

In C code, use the utility function CciLog to report errors. Two of the arguments that you pass to this function, `messageSource` and `messageNumber`, define the event source and the integer representation of a message within that source, respectively.

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*

On Windows systems, the messages are written to the Windows event log, and you must create your message catalog as a DLL file.

For Linux, UNIX, and z/OS systems, the messages are written to the SYSLOG facility, and you must deliver your message catalog as an XPG4 message catalog.

The description here covers exceptions that are raised during normal 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, and then copy the file to each computer on which you are running the workbench, so that your messages can be displayed.

**Building and installing a Windows event source:**

Windows On Windows, the message catalog is delivered as a DLL file, which you create by following the instructions given here. 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 this message file to create a resource compiler input file, by issuing the command:

   ```
   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 coded in the input.mc file. You must include this header file when you compile a user-defined extension source file that uses the CciLog utility function to write an event message that you have defined. The `messageNumber` argument to CciLog must use the appropriate value hash-defined in the output header file.
3. Compile the output file (.rc) from the message compiler to create a resource (.res) file by issuing the command:

   ```
   RC /v output_file.rc
   ```
4. Create a resource DLL file from the .res file by issuing the command:

   ```
   LINK /DLL /NOENTRY resource_file.res
   ```

To install the event source into the Windows Event Log Service:
1. Start the Windows Registry Editor by issuing the command:

   ```
   regedit
   ```
2. 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 on the `messageSource` parameter of the CciLog invocation.

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 that CciLog uses.

**TypesSupported**
> Set the DWORD value to ″7″.

# 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 115
- "C Header files" on page 115
- "C language user-defined node API" on page 116
- "C language user-defined parser API" on page 189
- "C user exit API" on page 244
- "C common API" on page 258
- "C skeleton code" on page 289
- Java user-defined node API
- "Utility function return codes and values" on page 292
- "Available parsers" on page 295
- "XML and MRM parser constants" on page 296
- "Trace logging from a user-defined C extension" on page 298
- "National language support considerations for message catalogs" on page 299
- "Node templates" on page 301
- "Panel definitions" on page 309
- "Terminal definitions" on page 319

## Sample node files

**Windows** On Windows, the following sample node files are in the *WBIMB_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. |

<span style="background:#9e5b5b;color:white"> Windows </span>  On Windows, the following sample node files are in the *WBIMB_install_dir*\sample\extensions\nodes directory.

<span style="background:#9e5b5b;color:white"> Linux </span>  On Linux, the following files are in the *install_dir*/sample/Javaplugin/ com/ibm/samples directory.

<span style="background:#9e5b5b;color:white"> UNIX </span>  On UNIX, the following files are in the *install_dir*/sample/Javaplugin/ com/ibm/samples directory.

<span style="background:#9e5b5b;color:white"> z/OS </span>  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.

There are also a number of gif files that are used to represent the sample nodes in the workbench, which you can use, or replace with your own. 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 SupportPacs. For a complete list of available SupportPacs see WebSphere MQ SupportPacs Web page.

# Sample parser files

`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 SupportPacs. 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 116.
- **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 189.
- **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 261. This file also contains definitions for utility function return codes and values. For more information, see "Utility function return codes and values" on page 292.

- **BipCos.h** contains operating system specific definitions for user-defined nodes that have been written in C.

# C language user-defined node 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 invoked 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 message broker, or request a service of the broker. These utility functions are invoked by a user-defined node.

Most of the utilities are shared by any type of node, however there are a few that are specific to input nodes. These are 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 117**

## C node implementation functions

The user-defined node implements a function interface for the message broker to invoke during runtime execution. This includes functions to create a local context whenever a node instance is created, functions to set and retrieve attribute values, the function to actually perform the processing of the node itself, and functions to examine messages:

**Mandatory function**
"cniCreateNodeContext" on page 134

**Optional and conditional functions**
- "cniDeleteNodeContext" on page 138
- Either "cniEvaluate" on page 151 (for message processing and output nodes), or "cniRun" on page 167 (for input nodes)
- "cniGetAttribute" on page 155
- "cniGetAttribute2" on page 156
- "cniGetAttributeName" on page 157
- "cniGetAttributeName2" on page 158
- "cniSetAttribute" on page 173

These implementation functions are called by the broker and implemented by the node.

For certain implementation functions, it might be necessary to specify the name of a parser supplied with WebSphere Message Broker. When doing 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

Using the following system-provided functions, a C user-defined node can create or define message broker objects, such as node factories, nodes, and terminals. 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.

This section covers the following topics:

**Initialization and resource creation**
- "cniCreateNodeFactory" on page 135
- "cniDefineNodeClass" on page 137
- "cniDispatchThread" on page 139 (for input nodes only)
- "cniCreateInputTerminal" on page 132
- "cniCreateOutputTerminal" on page 136
- "cniIsTerminalAttached" on page 162
- "cniGetBrokerInfo" on page 159

**Message management**

# cniAddAfter

Adds an unattached syntax element after a specified syntax element. The currently unattached syntax element, and any child elements it possesses, is connected to the syntax element tree after the specified target element. The newly 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

Adds an unattached syntax element as the first child of a specified syntax element. The currently unattached syntax element, and any 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

Adds an unattached syntax element as the last child of a specified syntax element. The currently unattached syntax element, and any 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

Adds an unattached syntax element before a specified syntax element. The currently unattached syntax element, and any 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

Gets 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

Gets a pointer to the data buffer associated with (and owned by) the message object specified in the message argument. This function is normally 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

Gets 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

Copies 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, then 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 any 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

Creates a new syntax element and inserts it after the specified syntax element. The new element becomes the *next sibling* of the specified element.

`cniCreateElementAfter` should not be used 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 any of the following functions:
```
cniCreateElementAsFirstChildUsingParser
cniCreateElementAsLastChildUsingParser
cniCreateElementAfterUsingParser
cniCreateElementBeforeUsingParser
```

When the message body folder has been created, `cniCreateElementAfter` can be used to create elements under the folder. `cniCreateElementAfter` can be used 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

Creates a new syntax element, inserts it after the specified syntax element, and associates 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 outputting 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* 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

Creates a new syntax element as the first child of the specified syntax element.

`cniCreateElementAsFirstChild` should not be used 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 any of the following functions:

```
cniCreateElementAsFirstChildUsingParser
cniCreateElementAsLastChildUsingParser
cniCreateElementAfterUsingParser
cniCreateElementBeforeUsingParser
```

When the message body folder has been created, `cniCreateElementAsFirstChild` can be used to create elements under the folder. `cniCreateElementAsFirstChild` can be used 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

Creates a new 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 outputting 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

Creates a new syntax element as the last child of the specified syntax element.

`cniCreateElementAsLastChild` should not be used 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 any of the following functions:

```
cniCreateElementAsFirstChildUsingParser
cniCreateElementAsLastChildUsingParser
cniCreateElementAfterUsingParser
cniCreateElementBeforeUsingParser
```

When the message body folder has been created, `cniCreateElementAsLastChild` can be used to create elements under the folder. `cniCreateElementAsLastChild` can be used 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

Creates a new syntax element tree as the last child of the specified syntax element, and associates it with the specified parser. The new 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 should already exist. The root element of this output message should be passed in as the target element parameter. Because this call is only designed 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 RFH2 as the last child of root and then an XML message as the last child of root, after the RFH2.

### 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 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

Creates a new syntax element as the last 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 outputting 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 116 for a list of the supplied parsers.

If you use this function to create a BLOB parser folder, 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*.

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* 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

Creates a new syntax element and inserts it before the specified syntax element. The new element becomes the *previous sibling* of the specified element, and shares the same parent element.

cniCreateElementBefore should not be used 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 any 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

Creates a new syntax element, inserts it before the specified syntax element, and associates 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 outputting 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

Creates an input terminal on an instance of a node object, returning the address of the terminal object that was created. The terminal object is destroyed by the message 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

Creates a new output message object. For every call to this function, there should be 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

Creates any context for an instance of a node object. It is invoked by the message 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 at this point are to:
1. (Optionally) 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 any 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 invoked, the appropriate context is passed as an argument to that function. This means that a user-defined node developed in C need not 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

Creates a node factory in the message broker engine. A single instance of the named message flow node factory is created.

This function must be invoked only in the initialization function bipGetMessageFlowNodeFactory, which is called when the LIL is loaded by the message broker. If cniCreateNodeFactory is invoked 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

Creates an output terminal on an instance of a node object, returning 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

Defines 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 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

Deletes the specified message object. For every call to the cniCreateMessage function, there should be 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

Deletes any context for an instance of a user-defined node object. It is invoked by the message 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 at this point are to:
1. Release any node instance specific data areas (such as context) that were acquired at construction or during node processing.
2. Release any 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

Detaches the specified syntax element from the syntax element tree. The element is detached from its parent and siblings, but any 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

This function dispatches a new message flow thread to invoke 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 there are no threads available because they are all in use, CCI_SUCCESS is returned and *returnCode* is set to CCI_NO_THREADS_AVAILABLE. This is not an error, but means one of the following:

- The message flow was not configured to run with additional threads.
- All additional threads configured are currently running.

The cniDispatchThread function can only be issued from an input node. If it is issued 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 WebSphere Message Broker creates or reuses the thread. This is passed to the node when its cniCreateNodeContext implementation function is invoked (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 there were insufficient threads 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 issued from within an input node, CCI_FAILURE is returned and *returnCode* is set to CCI_INV_NODE_ENV.
- For any other error conditions, CCI_FAILURE is returned, and *returnCode* indicates the reason for the error.

### Example

```
cniDispatchThread(&rcDispatch, ((NODE_CONTEXT_ST *)context)->nodeObject);
```

## cniElementAsBitstream

Gets the bitstream representation of the specified element. The parser that is associated with the element serializes the element and all its children. 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 re-serialize 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 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

**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 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 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 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 queue manager's encoding should 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 queue manager's ccsid is used. A ccsid of -1 indicates that the bit stream is to be generated using ccsid information contained in the subtree consisting of the field pointed to by the element and its children. No supplied parsers support this option.

**options**

Integer value that specifies which bitstream generation mode should 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 needed 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 the options parameter can be used 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><foo>text</foo></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, and 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 may 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 may wish to */
  /* log an error here 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

Gets the value of the *name* attribute for the specified syntax element. The syntax element name will have been set previously 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

Gets the value of the *namespace* attribute for the specified syntax element. The syntax element name will have been set previously using cniSetElementNamespace or cpiSetElementNamespace.

This is used when converting 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. 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
- 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 should 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

Gets the value of the *type* attribute for the specified syntax element. The syntax element type will have been set previously 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

These functions retrieve the value of the specified syntax element.

### 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

Gets 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

Gets 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

Gets 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

Performs node processing. The broker invokes 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 any 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 WTX 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

Causes 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

Returns 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

This function gets the value of an attribute on a specific node instance. It is invoked by the message broker after all of the attributes that the user deploys are set. The results are written to the broker's persistent configuration store in order to ensure that the node is configured correctly after the execution group process is stopped and started.

The responsibilities of the node at this point 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 | 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 should 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 invoked by the message broker when the broker requires the names of the attributes that are 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 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 should 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

Queries 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

Gets the CciMessage object corresponding to the *Environment* 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 might be an input message received as an argument to the cniEvaluate implementation function, or a message created using the cniCreateMessage utility function.

### Return values

If successful, the address of the message object corresponding to the Environment is returned. Otherwise, a value of zero is returned, and the *returnCode* parameter indicates the reason for the error.

## cniGetMessageContext

Gets the address of the message context associated with the specified message. The context of an existing message is used to create an output message, for example 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

Gets the parser class name 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 any other error occurs, CCI_FAILURE is returned, and the *returnCode* parameter indicates the reason for the error.

# cniGetThreadContext

Returns the thread context for the current thread.

## Syntax

```
CciThreadContext *cniGetThreadContext(
  int                 *returnCode,
  CciMessageContext     *msgContext);
```

## Parameters

**returnCode**
This is the return code from the function (output). If the input is NULL, this signifies that errors are silently handled or are ignored by the broker. If the input is not NULL, the output signifies the success status of the call. If the msgContext parameter is not valid, then *returnCode is set to CCI_INV_MESSAGE_CONTEXT and a NULL CciThreadContext is returned.

**msgContext**
This 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

Checks 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

Returns 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

Returns 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

Returns 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

Returns 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

Propagates 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 the action that the node takes would be different (in which case you can use cniIsTerminalAttached to check whether the terminal is connected).

### 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

Gets 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 invokes 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

Searches previous siblings of the specified element for an element matching 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:

1. 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.
2. 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.
3. 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.
4. 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 only performed on those elements whose namespace is an empty string. If you want to perform a search for elements in any namespace, use one of the cniSearchElementNamespace commands.

## 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 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). This is used only if the search mode involves a match on the type.

**name**
> The element name to search for (input). This is used 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

Searches for an element matching 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.

This is used when searching 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. 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. 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 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 is different behavior to that of the **cniSearchElement** group, where only the empty string namespace is searched. When you specify one of the above modes, set the *nameSpace* parameter to empty string.

**type**
> The element type to search for (input). This is used only if the search mode involves a match on the type.

**nameSpace**
> The namespace to search (input). This is used only if the search mode involves a match on the namespace.

**name**
> The name to search for (input). This is used 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

Sets the value of an attribute on a specific node instance. It is invoked by the message 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 at this point 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, 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

Sets 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

Sets the *namespace* attribute for the specified syntax element.

This is used when manipulating 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 only has an effect 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

Sets 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

Functions to set a value into the specified syntax element.

### 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

Sets 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). The address of the value object is obtained using cniElementValueValue.

### Return values

None. If an error occurs, the *returnCode* parameter indicates the reason for the error.

## cniSetInputBuffer

Using this function, the caller can supply a buffer. It is used by input nodes only. 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

Creates a 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. 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 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

Creates a 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

Creates an SQL expression object representing the statement specified by the
statement argument, using the syntax as defined for the Compute message flow
processing node, with the exception that you are not allowed to use:

- 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, namely cniSqlExecute and cniSqlSelect.
You can create multiple SQL expression objects in a single message flow processing
node. Although you can create these objects at any time, you would typically
create them when the message flow processing 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 is the default.

>    **CCI_SQL_TRANSACTION_COMMIT**

>    >    Specifies that a commit is performed after execution of the statement,
>    >    and within 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
>    message flow processing 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

Deletes the SQLPathExpression previously created by 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. 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. If an invalid sqlPathExpression parameter was passed in, then returnCode will be set to CCI_INV_SQL_EXPR_OBJECT.

**sqlPathExpression (output)**

Specifies the SQLPathExpression object to be deleted as returned by one of the cniCreate[ReadOnly|Modifiable]PathExpression functions. May not 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, you should place the following code in _deleteNodeContext

```
cniSqlDeletePathExpression(
                    NULL,
                    ((NODE_CONTEXT_ST *)context)->pathExpression);
```

## cniSqlDeleteStatement

Deletes an SQL statement previously created 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

Executes an SQL statement that has been previously created 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

Executes the SQLPathExpression previously created with 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 can not be NULL.

**inputMessageRoot (input)**

The message representing the input message. This parameter can not be NULL.

**inputLocalEnvironment (input)**

The message representing the input local environment. This parameter can not be NULL.

**inputExceptionList (input)**

The message representing the input exception list. This parameter can not 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

Executes an SQL statement that has been previously created 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

Writes the syntax element tree associated with the specified message to the data
buffer owned by that 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 it will not work.

## 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:

1. A set of implementation functions that provide the functionality of the user-defined parser. These functions are invoked by the message 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 message broker or request a service of the broker. These utility functions can be invoked 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 190.

## C parser implementation functions

A user-defined parser implements its capability through a function interface which is invoked by the message broker during runtime execution. This interface includes functions to create and delete any local context storage associated with a parser object and the parsing operations.

Some implementation functions are mandatory, and must be implemented by the developer, as shown below.

This section covers the following topics:

**Mandatory functions**
- "cpiCreateContext" on page 200
- "cpiParseNextSibling" on page 225
- "cpiParsePreviousSibling" on page 226
- "cpiParseFirstChild" on page 223
- "cpiParseLastChild" on page 224

# C parser utility functions

The following system-provided functions allow the C user-defined parser to create or define message broker objects, such as message parser factories.

This section covers the following topics:

# cpiAddAfter
## Purpose

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

## Purpose

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** (lines
675 to 698):

```
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

## Purpose

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** (lines 209 to 228):

```
/* 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

## Purpose

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

## Purpose

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** (line
634):

```
cpiAppendToBuffer(&rc, parser, (char *)"Some test data", 14);
```

# cpiBufferByte

## Purpose

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** (lines 61 to 75):

```
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

## Purpose

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** (lines 428 to 445):

```
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

## Purpose

Gets the size of the buffer containing 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** (lines 428 to 452):

```
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

### Purpose

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

## Purpose

Creates a user-defined extension context associated with a parser object. It is invoked by the message broker when an instance of a parser object is constructed or allocated. This 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 will have created 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 responsibilities of the extension are to:
1. Allocate any parser instance specific data areas (such as context) that might be required.
2. Perform any 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 invoked, the appropriate context is passed as an argument to that function. This means that 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

## Purpose

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** (lines 198 to 225):

```
/* 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
## Purpose

Creates a single instance of the named parser factory in the message broker. It must be invoked only in the initialization function **bipGetParserFactory** which is called when the 'lil' is loaded by the message broker. If **cpiCreateParserFactory** is

invoked at any other 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** (lines 862 to 901):

```
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

### Purpose

Defines the name of a parser class that is supported by a parser factory. **functbl** is a pointer to a virtual function table containing 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** (lines 862 to 901):

```
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
## Purpose

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 any parser instance specific data areas (such as context) that were acquired at construction or during parser processing.
2. Release any 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

### Purpose

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** (lines 491 to 499):

```
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

## Purpose

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 similar to code taken from the sample parser file **BipSampPluginParser.c** (lines 491 to 499). In the sample file, the code given is for 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

## Purpose

Gets the name of the target syntax element. The syntax element name will have been set previously 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 will be 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 any other failures, CCI_FAILURE is returned and **returnCode** indicates the reason for the error.

## Sample

```
cpiElementName(&rc;, element, (CciChar*)&elementName;, sizeof(elementName));
```

# cpiElementNameSpace
## Purpose

Gets 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. 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. 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 will
be copied. A string of characters (including a NULL terminator) representing
the namespace value is copied into this buffer. The buffer should 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

cpiElementType C API command

## Purpose

Gets the type of the target syntax element. The syntax element type will have been
set previously 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** (lines 491 to 499):

```
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
## Purpose

Optional function to get the value of a specified element. It is invoked 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 will have been returned by the **cpiElementValueValue** function.

# cpiElementValue group
## Purpose

Functions to get the value of the specified syntax element.

## 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, for example, **cpiElementCharacterValue** or **cpiElementDecimalValue**, 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.

If an error occurs, **returnCode** indicates the reason for the error.

## cpiElementValueValue
### Purpose

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 can be used by parsers that override behavior by invoking 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** (lines 705 to 725):

```
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

### Purpose

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** (lines 494 to 496):

```
while ((!cpiElementCompleteNext(&rc, element))      &&
          (!cpiFirstChild(&rc, element)) &&
          (pc->iCurrentElement))
```

# cpiLastChild
## Purpose

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
## Purpose

Optional function to return the name of the next parser class in the chain, if any. It allows the parser to return to the broker the name of the parser class that handles the next section, or remainder, of the message content. Normally, for messages having a simple format type, there is only one message content parser; it is not necessary to provide this function. For messages having a more complex format

type with multiple message parsers, each parser should 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 (lines 732 to 756).

```
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

### Purpose

Optional function to return the coded character set ID (CCSID) of the data owned
by the next parser class in the chain, if any.

| 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 will apply.

### Sample

This example is taken from the sample parser file BipSampPluginParser.c (lines 820
to 839).

```
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

## Purpose

Optional function to return the encoding of data owned by the next parser class in the chain, if any.

| 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 will apply.

## Sample

This example is taken from the sample parser file BipSampPluginParser.c (lines 794 to 813).

```
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

## Purpose

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** (lines 494 to 496):

```
while ((!cpiElementCompleteNext(&rc, cpiParent(&rc, element))) &&
        (!cpiNextSibling(&rc, element))     &&
        (pc->iCurrentElement))
```

# cpiParent

### Purpose

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** (lines 116 to 173):

```
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

## Purpose

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. Each user-defined parser that is used to parse a particular message format has this function invoked to:

- Perform any 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 is necessary because another parser might own a previous portion of the message (for example, an MQMD header will have been parsed by the message broker's internal parser). The offset must be positive and be less than the size of the buffer. It is recommended that the implementation function verifies that the offset is valid, as this could improve problem determination if a previous parser is in error.

The parser must return the size of the remaining buffer for which it takes ownership. This 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 can cause unpredictable results.

If this implementation function is provided in the CPI_VFT structure, neither **cpiParseBufferEncoded()** nor **cpiParseBufferFormatted()** can be specified, because the **cpiDefineParserClass()** function will fail 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** (lines 428 to 466):

```
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
## Purpose

This function is an extension of the capability provided by the existing **cpiParseBuffer()** implementation function that 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, neither **cpiParseBuffer()** nor **cpiParseBufferFormatted()** can be specified, otherwise the **cpiDefineParserClass()** function will fail 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** (lines 428 to 466):

```
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
### Purpose

This function is an extension of the capability provided by the existing cpiParseBuffer() implementation function that provides:

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, neither **cpiParseBuffer()** nor **cpiParseBufferEncoded()** can be specified, because the **cpiDefineParserClass()** function will fail 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** (lines 428 to 466):

```
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

## Purpose

Parses the first child of a specified syntax element. It is invoked 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** (lines 477 to 508):

```
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

## Purpose

Parses the last child of a specified syntax element. It is invoked 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** (lines
515 to 544):

```
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

## Purpose

Parses the next (right) sibling of a specified syntax element. It is invoked 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** (lines 578 to 605):

```
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
## Purpose

Parse the previous (left) sibling of a specified syntax element. It is invoked 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 in the context shown above when you are parsing the bit-stream right to left.

# cpiParserType

## Purpose

Optional function to return whether the parser is an implementation of a *standard* parser. Such a parser expects that the **Format** field of the preceding header will contain the name of the parser class that follows. *Non-standard* parsers expect that the **Domain** field will contain the parser class name. If the **cpiParserType** implementation function is not provided, the message broker assumes that the parser is of the *standard* type.

| 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 and a non-zero value is returned.

# cpiRootElement

## Purpose

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** (lines 428 to 470):

```
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

### Purpose

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

## Purpose

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** (lines
289 to 318):

```
/* 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

### Purpose

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** (lines 289 to 318):

```
/* 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
## Purpose

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** (lines 209 to 228):

```
/* 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
### Purpose

Sets 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 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. (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 representing the
namespace value. An empty string is a valid value for namespace. In fact,
elements are created in the empty string namespace by default so specifying an
empty string as the namespace via this API will only have any effect if the
element was previously in another namespace and the desired 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
## Purpose

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** (lines 209 to 228):

```
/* 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
## Purpose

Optional function to set the value of a specified element. It is invoked 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** (lines 675 to 698):

```
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
## Purpose

Functions to set a value in the specified syntax element.

## 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.
Used on relevant function calls only.

### Return values

None. If an error occurs, **returnCode** indicates the reason for the error.

# cpiSetElementValueValue

## Purpose

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** (lines 675 to 698):

```
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

### Purpose

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

### Purpose

Optional function to advise a parser of the next parser in the chain. It is called
during finalize processing, and returns to the user-defined parser a string
containing the name of the next parser class in the chain. It allows a parser to take
action during the finalize phase to 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 that the
> **Format** field of the preceding header in the chain will contain the name of the
> parser class that follows. Non-standard parsers expect that the **Domain** field
> will contain the parser class name.

### Return values

None.

### Sample

This example is taken from the sample parser file **BipSampPluginParser.c** (lines
763 to 787):

```
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

## Purpose

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 should not 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, neither **cpiWriteBufferEncoded()** nor **cpiWriteBufferFormatted()** can be specified, because the **cpiDefineParserClass()** function will fail 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
## Purpose

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
serialises its element tree to an output bit stream. If serialisation is not required, for
example when the output based 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, neither
**cpiWriteBuffer()** nor **cpiWriteBufferFormatted()** can be specified, because the
**cpiDefineParserClass()** function will fail 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** (lines 612 to 642):

```
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

### Purpose

This function is an extension of the capability provided by the existing **cpiWriteBuffer()** implementation function that provides:

1. 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.
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, neither **cpiWriteBuffer()** nor **cpiWriteBufferEncoded()** can be specified, because the **cpiDefineParserClass()** function will fail 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 C language user exit API consists of:

1. A set of implementation functions that provide the functionality of the user exits. Some of these implementation functions are mandatory and others are optional.
2. A set of utility functions, which are invoked by user exits.

These functions are defined in the **BipCpi.h** header file.

This section covers the following topics:
- "C user exit implementation functions."
- "C user exit utility functions" on page 251.

# 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, as shown below.

This section covers the following topics:

**Mandatory functions**
- "bipInitializeUserExits"
- "bipTerminateUserExits" on page 246

**Optional functions**
- "cciPropagatedMessageCallback" on page 249
- "cciNodeCompletionCallback" on page 248
- "cciInputMessageCallback" on page 246
- "cciTransactionEventCallback" on page 250

## bipInitializeUserExits

This is an implementation function exported by the User Exit library (`.lel`). It is invoked when the execution group starts just after loading the `.lel`. During invocation of this function, the user's code should call **cciRegisterUserExit** to register each user exit provided by that `.lel`.

**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

This is an implementation function exported by the User Exit library (.lel). It is
invoked just before unloading the .lel which typically happens when the execution
group process is stopping. During invocation of this function, the user's code
should clean-up any resources allocated during the **bipInitializeUserExits** function.
If this function is not exported, then the .lel fails to load. It is not valid to call any
utility functions during invocation of **bipTerminateUserExits**. This function is
invoked on the same thread as **bipInitializeUserExits.**

**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

This is a function that can be registered as a callback and is invoked every time a
message is read by an input node, and before that message is propagated down
the message flow. It is invoked for every input message read 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 as the
**iFpInputMessageCallback** field of the CCI_UE_VFT struct passed to
**cciRegisterUserExit**.

**Syntax:**

```
typedef void (*cciInputMessageCallback)    (
                          CciDataContext*  userContext,
                          CciMessage*      message,
                          CciMessage*      localEnvironment,
                          CciMessage*      exceptionList,
                          CciMessage*      environment,
                          CciMessageOrigin messageOrigin,
                          CciNode*         inputNode);
```

**Parameters:**

**userContext (input)**

This is the value that was passed to the **cciRegisterUserExit** function.

**message**

This is a handle to the message object. The user exit code must not update this tree.

**localEnvironment**

This is a handle to the local environment object.

**exceptionList**

This is a handle to the exception list object.

**environment**

This is a handle to the environment object for the current message flow.

**messageOrigin**

Depending on the type of input node, the message may have originated from a bitstream (CCI_MESSAGE_ORIGIN_BITSTREAM) or from a tree (CCI_MESSAGE_ORIGIN_TREE). The user exit can therefore access one of these without causing processing by the parser. For example, in the case of the MQInputNode, you can safely access the bitstream whereas, in the case of the JMS input node, you can safely access the tree. The bitstream can be accessed by calling **cniBufferPointer**, **cniBufferSize**, or **cniBufferByte**. The tree can be accessed by calling **cniRootElement** and using the usual syntax element navigation functions (for example, **cniFirstChild** ).

**Note:** Although this parameter advises the user exit as to what it can safely access without causing processing by the parser, it is possible that the user exit code could ignore this advice and effectively alter the parse timing.

**inputNode**

This is a handle to the input node which reads this input message. It 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

This is a function that can be registered as a callback and is invoked whenever a
node has completed processing of a message and is returning control to its
upstream node. It is invoked for every message propagated 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 as the
**iFpNodeCompletionCallback** field of the CCI_UE_VFT struct passed to
**cciRegisterUserExit**.

If the node completed due to an unhandled exception, then it returns with a
**reasonCode** of CCI_EXCEPTION and that exception's details can be obtained by
calling **cciGetLastExceptionData**.

If the node completed normally (including handling an exception on the catch or
failure terminal) then it returns with a **reasonCode** of CCI_SUCCESS.

**Note:** 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)**
> This is the value that was passed to the **cciRegisterUserExit** function.

**message**
> This is a handle to the message object being propagated. The user exit code
> must not update this tree.

**localEnvironment**
> This is a handle to the local environment object being propagated.

**exceptionList**
> This is a handle to the exception list object being propagated.

**environment**
> This is a handle to the environment object for the current message flow.

**connection**
> This is a handle to the connection object between the two nodes. It can be
> used, for example, in calls to **cciGetSourceNode**, **cciGetTargetNode**,
> **cciGetSourceTerminalName**, and **cciGetTargetTerminalName**. This handle is
> valid only for the duration of this invocation of the user exit function.

**reasonCode**
> This indicates whether the node completed normally (CCI_SUCCESS) or the
> node completed due to an unhandled exception (CCI_EXCEPTION). If the

node completed due to an unhandled exception, then that exception's details can be obtained by calling **cciGetLastExceptionData**. If the node completed normally (including handling an exception on the catch or failure terminal) then 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

This is a function that can be registered as a callback and is invoked whenever a message is propagated from one node to another. It is invoked for every message propagated 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 as the **iFpPropagatedMessageCallback** field of the CCI_UE_VFT struct passed to **cciRegisterUserExit**.

**Syntax:**

```
typedef void (*cciPropagatedMessageCallback)(
                        CciDataContext* userContext,
                        CciMessage*     message,
                        CciMessage*     localEnvironment,
                        CciMessage*     exceptionList,
                        CciMessage*     environment,
                        CciConnection*  connection);
```

**Parameters:**

**userContext (input)**
> This is the value that was passed to the **cciRegisterUserExit** function.

**message**
> This is a handle to the message object being propagated. The user exit code must not update this tree.

**localEnvironment**
> This is a handle to the local environment object being propagated.

**exceptionList**
> This is a handle to the exception list object being propagated.

**environment**
> This is a handle to the environment object for the current message flow.

**connection**
> This is a handle to the connection object between the two nodes. It can be used, for example, in calls to **cciGetSourceNode**, **cciGetTargetNode**,

**cciGetSourceTerminalName**, and **cciGetTargetTerminalName**. This handle is
valid only for the duration of this invocation 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 then you should resize and retry*/

}
```

## cciTransactionEventCallback

This is a function that can be registered as a callback and is invoked every time a
message flow transaction ends. It is invoked 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 then that exception's details can be obtained 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){
...
...
}
```

# C user exit utility functions

The utility functions described in this section can be invoked by user exits.

This section covers the following topics:
- "cciGetNodeAttribute"
- "cciGetNodeName" on page 252
- "cciGetNodeType" on page 253
- "cciGetSourceNode" on page 254
- "cciGetSourceTerminalName" on page 255
- "cciGetTargetNode" on page 256
- "cciGetTargetTerminalName" on page 256
- "cciRegisterUserExit" on page 257

## cciGetNodeAttribute

This 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, then **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 then you should resize and retry*/
 /*sourceNodeQueueNameLength will hold the actual or required size */
```

## cciGetNodeName

This function returns the name of the specified node.

**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 then you should 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's type.

**length**
    The length, in CciChars, of the buffer allocated by the caller.

**Return values:**

- If 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*/
```

## 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 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 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 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's name.

**connection (input)**
This 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

This is a utility function that can be called by the user's code during invocation of **bipInitializeUserExits**. It is invoked 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;

} 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 alpha-numeric characters was specified.

**Name (input)**

This 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 alpha-numeric 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 allows the caller to provide a context pointer that is passed to the callback function when it is invoked. This parameter can be NULL.

**functionTable (input)**
This 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 invoked for that event.

**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;

  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:

1. A set of implementation functions, implemented by user-defined nodes, parsers, and user exits.
2. A set of additional utility functions.

These functions are defined in the **BipCpi.h** header file.

This section covers the following topics:
- "C common implementation functions."
- "C common utility functions" on page 261.

# C common implementation functions

The following functions are implemented by user-defined nodes or user-defined parsers. They will be called by the broker on occurrence of certain events.

These functions are defined in the **BipCci.h** header file.

## Optional functions
- cciRegCallback

## cciRegCallback

This is a function that can be registered as a callback and is invoked when the registered event occurs. The function is registered by providing a function pointer which matches the following typedef:

**Syntax:**

```
typedef int (*CciRegCallback)(CciDataContext *, cciCallbackType);
```

**Parameters:**

**type CciDataContext\***
    This is the pointer that is provided by the caller to the registration function.

**type CciCallbackType**
    This indicates the reason for the callback. This 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 should not be called again on this thread unless it is re-registered. If any other 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 should not call any node or parser utility function that depends on the existence of a node or parser. The only utility functions that may 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 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 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 are:
- 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 264
- "cciGetLastExceptionDataW" on page 265
- "cciLog" on page 267
- "cciLogW" on page 268
- "cciRethrowLastException" on page 273
- "cciThrowException" on page 277
- "cciThrowExceptionW" on page 278

The following utilities help you convert between WebSphere Message Broker's internal processing code (in UCS-2) and file code (for example, ASCII).
- "cciMbsToUcs" on page 269
- "cciUcsToMbs" on page 282

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 280
- "cciUserTrace" on page 286
- "cciUserTraceW" on page 288
- "cciUserDebugTrace" on page 283
- "cciUserDebugTraceW" on page 284
- "cciServiceTrace" on page 275
- "cciServiceTraceW" on page 276

- "cciServiceDebugTrace" on page 273
- "cciServiceDebugTraceW" on page 274

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 270

The following utility functions are available for use with user exits:
- "cciGetBrokerInfo"
- "cciGetNodeAttribute" on page 251
- "cciGetNodeName" on page 252
- "cciGetNodeType" on page 253
- "cciGetSourceNode" on page 254
- "cciGetSourceTerminalName" on page 255
- "cciGetTargetNode" on page 256
- "cciGetTargetTerminalName" on page 256
- "cciInputMessageCallback" on page 246
- "cciNodeCompletionCallback" on page 248
- "cciPropagatedMessageCallback" on page 249
- "cciRegisterUserExit" on page 257
- "cciTransactionEventCallback" on page 250

## cciGetBrokerInfo

This function queries 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.

**Note:** This differs from the existing **cniGetBrokerInfo** (Click "cniGetBrokerInfo" on page 159 for a description of that function) in that it is not necessary to provide a CciNode* handle and that **cciGetBrokerInfo** does not return any information about a message flow. Consequently, **cciGetBrokerInfo** can be called 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.

**Note:** Unless CCI_EXCEPTION is indicated you must not call cciGetLastExceptionData() as it 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 **exception_st.type** field is one of the following:

- 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;


CCI_EXCEPTION_ST exception_st = malloc(sizeof(CCI_EXCEPTION_ST));
int rc = 0;
memset(&exception_st,0,sizeof(exception_st));
cciGetLastExceptionData(&rc, &exception_st);
```

## 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.

**Note:** Unless CCI_EXCEPTION is indicated you must not call cciGetLastExceptionDataW() as 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 **exception_st.type** field is one of the following:

- 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;


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);
```

## cciLog

Logs an error, warning or informational event. The event is logged by the message
broker interface using 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**
>        A string that identifies the Windows message source or the Linux and
>        UNIX 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 below).

**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`.

**Note:** `char*` characters must be strings in either ASCII (Latin) or EBCDIC (1047).

**Note:** 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 message broker interface using 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**

A string that identifies the Windows message source or the Linux and
UNIX 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
below).

**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.

**Note:** 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

Converts multi-byte 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 has a trailing *CciChar* of 0, just as the *mbString* has a trailing byte of 0.

**ucsStringLength**

> The length (in *CciChars*) of the buffer that you have provided (input). Each byte in *mbString* expands to not more than one *CciChar* 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 should be suitable for the compiler that is being used 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 and UNIX, and for Windows platforms.
>
> On Linux and UNIX, `nl_langinfo(CODEPAGE)` gives you the code page that has been selected by `setlocale`.
>
> For OS/390 and z/OS, the default code page for WebSphere MQ, which is 500, should not be used. Instead, you should use a code page value of 1047.

**Return values:**

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). If the input is NULL, this signifies that errors are silently handled or are ignored by the broker. If the input is not NULL, the output signifys 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 provides the thread context in which to register the callback function and associated data. It is assumed that this parameter is obtained by using the cniGetThreadContext() API on the current thread. If NULL is supplied as threadContext, then the thread context is determined by the framework. This is less efficient than calling cniGetThreadContext.

**userContext**

This allows the caller to provide a context pointer that is passed to the callback function when it is invoked. This parameter can be NULL.

**callback**

This is a pointer to the callback function that is to be invoked. This function must be of the type CciRegCallback.

**type**

This specifies whether the callback is to be invoked 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 down 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 any other input node in the same message flow. This state is entered when additional instances, which have been deployed for a message flow, have been utilized 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 can happen when the broker is shutdown, the execution group process is ending in a controlled manner, or when the message flow is being deleted. This 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 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 permits the passing of data from the code where the callback is registered to the callback function.

## cciRethrowLastException

Re-throws the last exception generated on the current thread. It is used to pass the exception back to the message broker for further handling. Note that, similar to a C exit call, cciRethrowLastException does not return in this case.

**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*, then the name of that node is written to trace. If it is a CciParser*, then 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) codepage 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

This 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*, then the name of that node is written to trace. If it is a CciParser*, then 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 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 the address of a CciNode or a CciParser. If it is a CciNode, then the name of that node is written to trace. If it is a CciParser, then 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 following occurs:

- <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 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) codepage 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);
```

## 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*, then the name of that node is written to trace. If it is a CciParser*, then 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

Throws an exception. The exception is thrown by the message broker interface using 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**

A string that identifies the Windows message source or the Linux and UNIX 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`.

**Note:** `char*` characters must be strings in either ASCII (Latin) or EBCDIC (1047).

**Note:** 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 message broker interface using 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 WebSphere Message Broker
> version message catalog use 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 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`
`CciChar`.

**Note:** 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_FATAL_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

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*, then the name of that node is written to trace. If it is a CciParser*, then 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

Converts Universal Character Set (UCS) data to multi-byte 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 (BipSampPluginUtil.c) shows 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 should be suitable for the compiler that is being used 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 and UNIX, and for Windows platforms.
>
> On Linux and UNIX, `nl_langinfo(CODEPAGE)` gives you the code page that has been selected by `setlocale`.
>
> For OS/390 and z/OS, the default code page for WebSphere MQ, which is 500, should not be used. Instead, you should use a code page value of 1047.

**Return values:**

The converted length in bytes.

## cciUserDebugTrace

This function is very similar to **cciUserTrace** with the only difference being that the entry is written to user trace only when user trace is active at debug level.

**Note:** An entry is also written to service trace, when service trace is active at any level and when user trace is active at any level.

**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 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 the input 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** for 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*, then the name of that node is written to trace. If it is a CciParser*, then 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 platform, read the log on that platform, and then format the log on a different platform. For example, if the broker is running on Linux or UNIX but there is no .cat file available, the user could read the log, and 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. If there is an empty string value, the log formatter can not find the message source. Therefore, 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, 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 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 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) codepage.
- For user-defined extensions that are running on Z/OS platforms, the `char*` arguments must be in EBCIDIC (1047).

This includes 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

This function is very similar to **cciUserTraceW** with the only difference being that the entry is written to user trace only when user trace is active at debug level.

**Note:** If user trace is not active at debug level, an entry will be 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*, then the name of that node is written to trace. If it is a CciParser*, then 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 platform, read the log on that platform, and then format the log on a different platform. For example, if the broker is running on Linux or UNIX but there is no .cat file available, the user could read the log, and 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*, then the log formatter will fail 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

User-defined extensions **285**

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

Writes 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 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*, then the name of that node is written to trace. If it is a CciParser*, then 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 platform, read the log on that platform, and then format the log on a different platform. For example, if the broker is running on Linux or UNIX but there is no .cat file available, the user could read the log, and 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*, then the log formatter will fail 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 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) codepage.
- For user-defined extensions that are running on Z/OS platforms, the `char*` arguments must be in EBCDIC (1047).

This includes 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

Writes 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 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*, then the name of that node is written to trace. If it is a CciParser*, then 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 platform, read the log on that platform, and then format the log on a different platform. For example, if the broker is running on Linux or UNIX but there is

no .cat file available, the user could read the log, and 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*, then the log formatter will fail 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);
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

The following code provides a skeleton for code in a C user-defined node. It has the minimum content that is required to compile a user-defined node successfully.

```
#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 may 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}
```

# Java user-defined node API

Use the API described here for both user-defined nodes and for Java code that is called by JavaCompute nodes.

The Java classes and methods are described in the Javadoc information

# Utility function return codes and values

By convention, the return code output parameter of all utility functions is set to indicate successful completion, or otherwise. The following table lists all return codes with their meanings. These return codes are defined in the BipCci.h header file.

Table 1. Utility function return codes and values

| Return code | Explanation |
| --- | --- |
| CCI_BUFFER_TOO_SMALL | The output buffer is not large enough to store the requested data. |

*Table 1. Utility function return codes and values  (continued)*

| Return code | Explanation |
| --- | --- |
| 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. |
| 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. |

*Table 1. Utility function return codes and values (continued)*

| Return code | Explanation |
|---|---|
| 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 use of the IDOC or the XML parser is deprecated.

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 |
| MQDLH | MQDLH | MQDEAD |
| MQIIH | MQIIH | MQIMS |
| MQMD | MQMD | MQHMD |

| Parser | Root element name | Class name |
|---|---|---|
| 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 make use of user-defined parsers that have been supplied by third party vendors.

# XML and MRM parser constants

When you are writing user-defined extensions you might need to know the value of various constants.

This topic lists the names of the XML and MRM parser constants and their corresponding values.

## XML parser constants

*Table 2. XML parser names and values*

| 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 |

*Table 2. XML parser names and values  (continued)*

| 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

*Table 3. MRM parser names and values*

| Name | Value |
|---|---|
| PreDefStructureFav | 0x01000000 |
| PreDefStructure | 0x01000001 |
| SelfDefStructure | 0x01000002 |
| StructureInstance | 0x01000004 |
| MrmRoot | 0x01000008 |
| mtiSelfDefMessage | 0x01000010 |
| mtiPreDefMessage | 0x01000012 |
| mtiSelfDefIdentifier | 0x02000001 |
| mtiSdfFieldType | 0x02000002 |

*Table 3. MRM parser names and values  (continued)*

| | |
|---|---|
| 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 |

# 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.

There are 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, then the trace is associated with the execution group.

# National language support considerations for message catalogs

WebSphere Message Broker converts any message that is loaded from the code pages that are listed below 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.

# National language support considerations on Windows

Windows When building a message file for Windows that contains multiple locales, ensure that the computer's locale 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 and 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 | |

## National language support considerations on Linux and UNIX

When building message catalogs for Linux and UNIX, ensure that the catalogs are built in the following code pages:

| 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 |

## National language support considerations on z/OS

z/OS   When building message catalogs for z/OS, ensure that the catalogs are built in the following code pages:

| Locale | Code page |
|---|---|
| English | 1047 |
| Japan | 939 |
| Simplified Chinese (China) | 1388 |

# Node templates

When you are designing a user-defined node, use the node templates that are supplied with WebSphere Message Broker.

You can use the available panels and terminals in many combinations to make different types of node. The following table lists the main types of node.

| Node type | Description |
|-----------|-------------|
| Basic node | A simple node. No template is provided for this kind of node because it is already represented in WebSphere Message Broker. |
| Input node | A node that starts the processing from within a message flow. This node usually gets data from an external resource |
| Get node | A node that gets data from an external resource but does not send any business data. |
| Output node | A node that sends data to an external resource but does not receive any business data. |
| Request node | A node that sends and receives business data. |
| Reply node | A node that sends data to an external resource based on details in the current environment. No template is provided for this kind of node because it is already represented in WebSphere Message Broker. |
| Transformation node | A node that transforms a message and routes it to one or more terminals. |
| Routing node | A node that routes a message to one or more terminals. |
| Structural and utility nodes | Simple nodes that provide basic structure to message flows. The built-in nodes of this type are:<br>• Trycatch<br>• Order<br>• Throw<br>• Trace<br><br>No template is provided for this kind of node because it is already represented in WebSphere Message Broker. |
| Event drive architecture node | A node, or set of nodes, that provides complex control over the flow and control of data in a message flow, including:<br>• Combining messages from different sources<br>• Complex communications between a set of nodes, including transfer of control between nodes without data being exchanged.<br><br>For example, the aggregation nodes. No template is provided for this kind of node because it is already represented in WebSphere Message Broker. |

The following topics describe the types of node template that are available in more detail.

- "Input node template" on page 302
- "Get node template" on page 303
- "Output node template" on page 305
- "Request node template" on page 306
- "Transformation node template" on page 308
- "Routing node template" on page 309

# Input node template

An input node is the first node in a message flow; it is used to drive the rest of the message flow. The input node has the main control over the message flow, manages thread allocation, and begins and ends transactions.

The basic input node template has no input terminals so it can be driven only by its run method. The run method is called repeatedly by the broker, and it is responsible for generating messages and sending them to connected terminals. The main use of the input node is to receive (or get) data from external systems that is processed by the message flow.

## Panels

The panels that are available on the input node template are described in the following table.

| Panel name | Is the panel optional? | Description |
| --- | --- | --- |
| User-defined panels | Yes | This set of panels contains properties that are specific to this node. |
| Polling | Yes | This panel describes how to poll for events. |
| Transactionality | Yes | This panel describes how the node handles transactionality. If you do not define the transactionality panel on the node, the node is always non-transactional. |
| Record detection | Yes | This panel is required if you are using a streaming transport and need the **Parser inbound** panel to be present. |
| Parser inbound | Yes | This panel identifies the parser that generates a message from the supplied bit stream. You can remove this panel if the data from an external system is not a bit stream. |
| Retry | No | This panel describes how retry processing is done when a failure is rolled back to the input node. |
| Validation | Yes | This panel describes how a parser validates a message. This panel is applicable only if you use the **Parser inbound** panel. You can remove this panel if the data from an external system is not a bit stream. |

## Terminals

The terminals that are available on the input node template are described in the following table.

| Terminal name | Type | Description |
| --- | --- | --- |
| Out | Output data | A message that is received from an external resource is always sent to Out first if no error occurs in the input node. |
| Catch | Output data | A message that is rolled back to the input node is sent to the Catch terminal if it is connected. |
| Failure | Output data | A failure in the node is sent to the Failure terminal, including failures that are caused by the retry processing. |

You cannot add an In terminal to an input node, but you can add a control terminal.

### Flow of control

The basic flow of control for an input node involves the following steps:
1. The run method is called on the node and any new transactions are started.
2. The input node gets a message to be processed. You can use one of two methods:
   - For streaming transport, stream inbound properties are used to break up the bit stream into messages.
   - An API is called to get data from an external resource. The API defines the content of a message.
3. The data that is retrieved is converted into a message tree. You can use one of two methods:
   - Construct a parser using the **Parser inbound** panel, and use it to parse the bit stream.
   - Construct a message tree using the results from API calls; no parser is used. You must then propagate the message tree to the rest of the message flow.
4. Each separate message is propagated through the Out terminal. If no exception occurs, all transactions are committed, and the run method returns. If an exception occurs after the node, and it is rolled back to the input node, the message is propagated to the Catch terminal. This propagation occurs only if the Catch terminal is connected. If it is not connected, or a failure happens during the Catch terminal processing, the message is passed to the retry mechanism.
5. If the retry is successful, the transaction is committed and the flow ends. If retry processing finishes without processing the message successfully, either control is returned, and the flow ends, or the message flow is paused. This action depends on the transport and the properties that are set in the **Retry** panel.

Most of the required functions are supplied by the template, but you must carry out the following steps:
- Define which of the options panels you will use.
- Get data from the external system.
- Manage the transaction by supplying a resource manager for the transport that is being used.

## Get node template

A get node is used to retrieve data from an external resource, based on a request message being sent to its In terminal. A get node does not send business data out to the external system; it simply requests business data based on some metadata.

For example, an MQGet node can send a correllid to get a message that contains business data from a queue, or an e-mail node can retrieve an e-mail from a server. The **Request** panel is used to get details from the incoming message, to decide which resource to query, and which metadata to use. The results are stored using details in the **Result** panel. The get node parses the received data using the same techniques as the input node, so the code for this process can be shared. The get node does not control transactions or the allocation of threads.

## Panels

The panels that are available on the get node template are described in the following table.

| Panel name | Is the panel optional? | Description |
|---|---|---|
| User-defined panels | Yes | This set of panels contains properties that are specific to this node. |
| Transactionality | Yes | This panel describes how the node handles transactionality. If you do not define the transactionality panel on the node, the node is always non-transactional. |
| Record detection | Yes | This panel is required if you are using a streaming transport and need the **Parser inbound** panel to be present. |
| Parser inbound | Yes | This panel identifies the parser that generates a message from the supplied bit stream. You can remove this panel if the data from an external system is not a bit stream. |
| Request | No | This panel describes how to make a request. |
| Result | No | This panel describes where to store the reply. |
| Report | No | This panel describes where to store the report. |
| Validation | Yes | This panel describes how a parser validates a message. This panel is applicable only if you use the **Parser inbound** panel. You can remove this panel if the data from an external system is not a bit stream. |

## Terminals

The terminals that are available on the get node template are described in the following table.

| Terminal name | Type | Description |
|---|---|---|
| In | Input data | The node is driven by a message arriving on its In terminal. |
| Out | Output data | When a message is received from an external resource, it is sent to the Out terminal. |
| No message | Output data | If no message is retrieved, a message is sent to the No message terminal. |
| Failure | Output data | A failure in the node is sent to the Failure terminal. |

## Flow of control

The basic flow of control for a get node involves the following steps:

1. The node gets the request details based on the **Request** panel.
2. The node gets a message from the external transport, either using an API, or a stream of data, based on the type of transport that is used. If no message is available, a message is propagated to the No message terminal.
3. The result is converted into a message tree. You can use one of two methods:

- Construct a parser using the **Parser inbound** panel, and use it to parse the bit stream.
- Construct a message tree using the results from API calls; no parser is used.

4. The constructed message tree is added to the location that is set on the **Result** panel, and a status tree is constructed and added to the location that is set on the **Report** panel. The status is based on the result from making the call to the external system.

Most of the required functions are supplied by the template, but you must carry out the following steps:

- Define which of the options panels you will use.
- Request details for this get node.
- Get data from the external system.
- Create the message tree if no parser is used.
- Create a status tree.

# Output node template

An output node is used to send business data to an external system. It sends business data out but expects to receive only metadata or result data.

If you want to send and receive business data, you could use the request node.

## Panels

The panels that are available on the output node template are described in the following table.

| Panel name | Is the panel optional? | Description |
|---|---|---|
| User-defined panels | Yes | This set of panels contains properties that are specific to this node. |
| Transactionality | Yes | This panel describes how the node handles transactionality. If you do not define the transactionality panel on the node, the node is always non-transactional. |
| Request | Yes | This panel identifies the location in the message tree from which the outgoing message is retrieved. |
| Record creation | Yes | This panel is required if you are using a streaming transport and need the **Parser outbound** panel to be present. |
| Parser outbound | Yes | This panel identifies the parser that generates a bit stream from a message tree. |
| Report | No | This panel describes where to store the report. |
| Validation | Yes | This panel describes how a parser validates a message. This panel is applicable only if you use the **Parser outbound** panel. |

## Terminals

The terminals that are available on the output node template are described in the following table.

| Terminal name | Type | Description |
|---|---|---|
| In | Input data | The node is driven by a message arriving on its In terminal. |
| Out | Output data | When a message has been sent to an external resource, it is sent to the Out terminal; the message is unchanged, except for the addition of status information. |
| Failure | Output data | A failure in the node is sent to the Failure terminal. |

### Flow of control

The basic flow of control for an output node involves the following steps:

1. The message is converted into a bit stream using the parser that is specified on the **Parser outbound** panel. Alternatively, the message tree is converted into an API call directly from the tree.
2. The data is sent across the transport's API.
3. The status and result details are written to a status tree and added to the location that is given in the **Report** panel.
4. The message is propagated to the Out terminal.

Most of the required functions are supplied by the template, but you must carry out the following steps:

- Define which of the options panels you will use.
- Call the API to send data to an external system.
- Create the status tree for the result of sending the data.

## Request node template

A request node sends a request out to an external system, and waits for reply data to return. It is different to an output node because it expects both business data and status to be returned. A request node is typically used for synchronous transports where the request and the reply actions are tightly coupled.

### Panels

The panels that are available on the request node template are described in the following table.

| Panel name | Is the panel optional? | Description |
|---|---|---|
| User-defined panels | Yes | This set of panels that contain properties that are specific to this node. |
| Transactionality | Yes | This panel describes how the node handles transactionality. If you do not define the transactionality panel on the node, the node is always non-transactional. |
| Record detection | Yes | This panel is required if you are using a streaming transport and need the **Parser inbound** panel to be present. |
| Parser inbound | Yes | This panel identifies the parser that generates a message from the supplied bit stream. |

| Panel name | Is the panel optional? | Description |
|---|---|---|
| Record creation | Yes | This panel is required if you are using a streaming transport and need the **Parser outbound** panel to be present. |
| Parser outbound | Yes | This panel identifies the parser that generates a bit stream from a message tree. |
| Request | No | This panel describes how to make a request. |
| Result | No | This panel describes where to store the reply. |
| Report | No | This panel describes where to store a report. |
| Error | Yes | This panel describes how errors are handled. |
| Validation | Yes | This panel describes how a parser validates a message. This panel is applicable only if you use the **Parser inbound** panel. |

## Terminals

The terminals that are available on the request node template are described in the following table.

| Terminal name | Type | Description |
|---|---|---|
| In | Input data | The node is driven by a message arriving on its In terminal. |
| Out | Output data | When a message has been sent to an external resource, it is sent to the Out terminal; the message is unchanged, except for the addition of status information. |
| Error | Output data | An error from the external system is sent to the Error terminal. |
| Failure | Output data | A failure in the node is sent to the Failure terminal. |

## Flow of control

The basic flow of control for a request node involves the following steps:

1. The message is converted into a bit stream using the parser that is specified on the **Parser outbound** panel. Alternatively, the node converts the message tree into an API call.
2. The data is sent across the transport's API.
3. The node waits for a reply message to be returned. If the reply contains an external system error condition, it is sent to the error handling mechanism that is controlled by the **Error** panel.
4. The reply message is used to construct a message tree, using one of the following methods:
   - Using a parser that is specified on the **Parser inbound** panel
   - Constructing a tree without using a parser, using the results of the API call
5. The constructed message tree is added to the location that is specified in the **Result** panel. A status tree is also constructed and added to the location that is specified in the **Report** panel. The status is based on the result of making the call to the external system.
6. The resulting message is propagated.

Most of the required functions are supplied by the template, but you must carry out the following steps:

- Define which of the options panels you will use.
- Construct a bit stream from the message tree if no parser is used.
- Send data to an external system.
- Receive data from an external system.
- Calculate reported error codes from the external system.
- Construct a message tree if no parser is used.

# Transformation node template

A transformation node transforms a message tree, and sends the modified message tree to one or more output terminals.

The transformation is based only on the message content; do not access external resources.

## Panels

The panels that are available on the transformation node template are described in the following table.

| Panel name | Is the panel optional? | Description |
|---|---|---|
| User-defined panels | Yes | This set of panels contains properties that are specific to this node. |
| Parser target | Yes | This panel identifies that parser that generates a message from the resulting bit stream. |
| Parser source | Yes | This panel identifies that parser that generates a bit stream from the incoming message tree. |
| Validation | Yes | This panel describes how a parser validates a message. |

## Terminals

The terminals that are available on the transformation node template are described in the following table.

| Terminal name | Type | Description |
|---|---|---|
| In | Input data | The node is driven by a message arriving on its In terminal. |
| *xxxx* | Output data | A terminal with a user-assigned name for routing the message after this node. |
| Out | Output data | When a message has been transformed, it is sent to the Out terminal. |
| Failure | Output data | A failure in the node is sent to the Failure terminal. |

### Flow of control

The basic flow of control for a transformation node involves the following steps:

1. A message enters the node via the In terminal.
2. The message is transformed.
3. The message is sent to the Out terminal. In some cases, you can define new terminals, through which the message is routed after the transformation, as described in the Terminals table.

# Routing node template

A routing node routes a message to an output terminal based on the content of the message. The message tree is not modified.

### Panels

The panels that are available on the routing node template are described in the following table.

| Panel name | Is the panel optional? | Description |
|---|---|---|
| User-defined panels | Yes | A set of panels that contain properties that are specific to this node. |

### Terminals

The terminals that are available on the routing node template are described in the following table.

| Terminal name | Type | Description |
|---|---|---|
| In | Input data | The node is driven by a message arriving on its In terminal. |
| *xxxx* | Output data | A terminal with a user-assigned name for routing the message after this node. |
| Failure | Output data | A failure in the node is sent to the Failure terminal. |

### Flow of control

The basic flow of control for a routing node involves the following steps:

1. A message enters the node via the In terminal.
2. The message is routed to an output terminal.

# Panel definitions

Use a panel to group node properties into related groups.

The following topics describe the properties on each of the available panels:
- "Validation panel" on page 310
- "Record detection panel" on page 310

## Validation panel

The **Validation** panel contains validation options that are required by the parser.

The following table shows the properties that are available on the **Validation** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Validate | List | No | Indicates whether validation is required. Valid values are:<br>• None: no validation is performed.<br>• Content: content checks are performed, such as content validation and composition.<br>• Content and value: content checks are performed, such as content validation and composition, and value checks, such as whether the value conforms to data type, length, range, and enumeration.<br>• Inherit: all the validation options that are provided with the input message tree are used in preference to any that are supplied on the node. Inherit always resolves to None, Content, or Content and value. If Inherit is selected, the other validation properties are unavailable. |
| Failure action | List | No | The action to be taken when a validation failure occurs. Valid values are:<br>• Exception: an exception is issued on the first validation failure that is encountered.<br>• Exception list: an exception is issued if validation failures are encountered, but only when the current parsing or writing operation has completed.<br>• User trace: all validation failures are logged to user trace.<br>• Local error log: all validation failures are logged to the local error log. |

## Record detection panel

The **Record detection** panel contains properties that are required to define how a bit stream is split into messages.

The following table shows the properties that are available on the **Record detection** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Record detection | List | No | The mechanism to use to split the bit stream messages. Valid values are:<br>• End of bit stream: the node keeps reading until the end of the bit stream is reached (for example, the end of a file, or the end of a TCP/IP stream connection).<br>• Fixed: a fixed size is used to determine the length.<br>• Delimited: a delimiter is used to determine the length.<br>• Encoded: a length encoded value is used to determine the length.<br>• Parser: a built-in parser is used to parse the bit stream directly. The properties on the **Parser inbound** panel are used. |
| White space | List | No | Use this property to remove white space between records. White space can be:<br>• Space<br>• New line<br>• Tab<br><br>The default value is No, so white space is left between messages. If you set this property to Yes, white space before and after a record is removed. Do not set this property to Yes if you are using binary integer encoding. Delimiter matching is performed before white space is stripped. |
| Inclusive | Check box | No | Indicates whether the outgoing message includes the record detection parts of the message. If the check box is cleared, the detection data is removed (for example, the delimiter is removed). If the check box is selected, the detection data is left in the message. |
| Fixed size | Integer | No | The size to use if Record detection is set to Fixed. When Record detection is set to Fixed, this property is mandatory. |
| Delimiter | String | No | The delimiter to use if Record detection is set to Delimiter. This field contains plain text, and it can contain the following expressions, which can be used in any combination:<br>• Plain string: uses local systems code pages<br>• Hexadecimal string: \x'0d0a'<br>• Unicode string: \u'0031'<br>• New line: a new characters code for the host system |
| Encoding type | List | No | The encoding type to use if Record detection is set to Encoded. When Record detection is set to Encoded, this property is mandatory. The following values are valid:<br>• Binary integer: Uses binary C-style integers<br>• String integer: Uses text-based integers that are padded by zeros; for example, 0099 means 99 |
| Encoding endian | List | No | The endian type to use for encoding. Valid values are:<br>• Automatic: Uses the local computer's encoding<br>• Big endian: Uses big endian for integer encoding<br>• Little endian: Uses little endian for integer encoding |
| Encoding size | List | No | The size of the encoding variable to use if the message length is encoded. When Record detection is set to Encoded, this property is mandatory. The value that is set for this property depends on the Encoding type property.<br>• Binary integer: A value from the list of 1, 2, 4, and 8<br>• String integer: A value in the range 1 to 16 |

## Record creation panel

The **Record creation** panel contains properties that are required to define demarcation of records or messages that are being written to the data stream, particularly for transports that do not supply a mechanism to split data streams into messages consistently.

The following table shows the properties that are available on the **Record creation** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Record demarcation | List | No | The mechanism used to enhance the message data with demarcation information. Valid values are:<br>• None: No demarcation information is written to the stream. Use this option when the message data contains enough information to distinguish records in the stream (for example, the delimiter is already present in the message data), or when each message is written to a separate target (for example, when each file contains only one record).<br>• Fixed: a fixed size is used to determine the length. If you select this value, the Fixed size, Pad, and Truncate properties are available.<br>• Delimited: A delimiter is used to separate the records.<br>• Encoded: A length encoded value is written to the stream before the record. |
| Fixed size | Integer | No | The size to use if Record demarcation is set to Fixed. |
| Pad | Check box | No | Specifies whether the data is to be padded when it is shorter than the specified fixed length. You can specify a value for this property only if Record demarcation is set to Fixed. If the check box is cleared, no padding data is added. An error occurs if the message data is shorter than the fixed length. If the check box is selected, padding data is used, as specified by the Padding string property. |
| Padding string | String | No | The string to use when padding records. This field uses plain text, and it can contain the following expressions, in any combination:<br>• Plain string: uses local systems code pages<br>• Hexadecimal string: \x'0d0a'<br>• Unicode string: \u'0031'<br>• New line: a new characters code for the host system |
| Truncate | Check box | No | Specifies whether the data is to be truncated if it is longer than the specified fixed length. If the check box is cleared, the message data is not truncated. An error occurs if the message data is longer than the fixed length. If the check box is selected, the message data is truncated before it is written to the stream. |
| Delimiter | String | No | The delimiter to use if Record demarcation is set to Delimiter. This field contains plain text, and it can contain the following expressions, which can be used in any combination:<br>• Plain string: uses local systems code pages<br>• Hexadecimal string: \x'0d0a'<br>• Unicode string: \u'0031'<br>• New line: a new characters code for the host system |
| Encoding type | List | No | The encoding type to use if Record demarcation is set to Encoded. When available, this property is mandatory. The following values are valid:<br>• Binary integer: Uses binary C-style integers<br>• String integer: Uses text-based integers that are padded by zeros; for example, 0099 means 99 |
| Encoding endian | List | No | The endian type to use for encoding. Valid values are:<br>• Automatic: Uses the local computer's encoding<br>• Big endian: Uses big endian for integer encoding<br>• Little endian: Uses little endian for integer encoding |

| Property | Type | Configurable? | Description |
|----------|------|---------------|-------------|
| Encoding size | List | No | The size of the encoding variable to use if the message length is encoded. When available, this property is mandatory. The value that is set for this property depends on the Encoding type property.<br>• Binary integer: A value from the list of 1, 2, 4, and 8<br>• String integer: A value in the range 1 to 16 |

## Parser inbound (Parser target) panel

The **Parser inbound** panel contains properties that are required to define the parser to create a message tree from a bit stream. The properties on the **Parser target** panel perform the same functions, but the **Parser target** panel is used when the bit stream is not sent externally.

The following table shows the properties that are available on the **Parser inbound** and **Parser target** panels.

| Property | Type | Configurable? | Description |
|----------|------|---------------|-------------|
| Parse timing | List | No | Specifies whether to use on-demand parsing when parsing a message. This property also controls the timing of MRM input message validation. This property has no effect on the validation of output message. Valid values are:<br>• On demand: Validation of a field in the message is delayed until it is parsed by partial parsing.<br>• Immediate: Partial parsing is overriden, and everything in the message is parsed and validated.<br>• Complete: Partial parsing is overriden, and everything is parsed and validated. |
| Override | List | No | Specifies whether the following properties (Domain, Message set, Message type, and so on) override the message properties in the incoming message. Valid values are:<br>• No: The values of the properties in this group are used only as defaults; they take effect only if a property is not already present in the message.<br>• Yes: The values of the properties in this group override any values in the incoming message.<br>• Use outbound parser: This value is applicable only if the **Parser outbound** panel is used. All values are taken from the **Parser outbound** panel. Use this value for a request node when a bit stream is sent to an external service as a request, and the expected response is a bit stream that matches the same model. |
| Domain | String | No | The domain to use to parser and serialize the bit stream. |
| Message Set | String | No | The message set to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Message Type | String | No | The message type to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Message Format | String | No | The message format to use when parsing the bit stream. This property applies only to parsers that support this option. |
| CCSID | Integer | No | The CCSID to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Encoding | Integer | No | The encoding to use when parsing the bit stream. This property applies only to parsers that support this option. |

# Parser outbound (Parser source) panel

The **Parser outbound** panel contains properties that are required to define the parser to create a bit stream from a message tree. The properties on the **Parser source** panel perform the same functions, but the **Parser source** panel is used when the bit stream was originated internally.

The following table shows the properties that are available on the **Parser outbound** and **Parser source** panels.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Override | List | No | Specifies whether the following properties (Domain, Message set, Message type, and so on) override the message properties in the incoming message. Valid values are:<br>• No: The values of the properties in this group are used only as defaults; they take effect only if a property is not already present in the message.<br>• Yes: The values of the properties in this group override any values in the incoming message. |
| Domain | String | No | The domain to use to parser and serialize the bit stream. |
| Message Set | String | No | The message set to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Message Type | String | No | The message type to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Message Format | String | No | The message format to use when parsing the bit stream. This property applies only to parsers that support this option. |
| CCSID | Integer | No | The CCSID to use when parsing the bit stream. This property applies only to parsers that support this option. |
| Encoding | Integer | No | The encoding to use when parsing the bit stream. This property applies only to parsers that support this option. |

# Request panel

The **Request** panel is used for nodes that interface with an external resource, and it contains properties that determine what is being asked of that external resource. The panel contains field references to the top of the metadata and the location of the message data to be sent out.

The following table shows the properties that are available on the **Request** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Request message location | String | No | The location of the message to be sent as part of the request. You can remove this property when you do not need to get data from the incoming message. The default value is Root.Body (the body of the incoming message). |
| Request metadata location | String | No | The location of the metadata that is used to control the sending of data. This location is the top of the structure that is used to contain the metadata. The default value depends on the node that you are creating because each node has a different default location in the LocalEnvironment. |

# Result panel

The **Result** panel contains properties for nodes that interface with an external resource. These properties control how the data that is returned from the source is processed.

The resulting data might be the result of a transformation, or a piece of data that is retrieved from an external resource. This panel is available to message enriching nodes (such as get or request nodes) and transformation nodes. The properties on this panel specify the location in the message assembly where the data is placed, and which trees are copied to the outgoing message assembly. Every message assembly that is propagated to a node contains the following trees:
- Message
- LocalEnvironment
- ExceptionList
- Environment

In the node, these trees are referred to as:
- InputRoot
- InputLocalEnvironment
- InputExceptionList
- Environment

Each node also creates new Message, LocalEnvironment, and ExceptionList trees. In the node, these trees are referred to as:
- OutputRoot
- OutputLocalEnvironment
- OutputExceptionList

The following controls specify if these new trees are initially empty, or are a copy of the input:
- CopyLocalEnvironment
- CopyMessage
- CopyExceptionList

The following table shows the properties that are available on the **Result** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Output data location syntax | List | No | Specifies whether the output data location syntax is ESQL or XPath. The default value is ESQL. |
| Output data location | String | No | The part of the message assembly that is to be populated with the resulting message tree. |

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Generate mode | List | No | Specifies which parts of the input message assembly are propagated as part of the output, and which parts are generated, defining whether input or output trees are propagated from the node. If input trees are propagated, any changes to output trees are lost. Valid values are:<br>• Message: Propagates a new Message tree and the original LocalEnvironment and ExceptionList trees<br>• LocalEnvironment: Propagates a new LocalEnvironment tree and the original Message and ExceptionList trees<br>• Message and LocalEnvironment: Propagates new Message and LocalEnvironment trees, and the original ExceptionList tree<br>• Exception: Propagates a new ExceptionList tree, and the original Message and LocalEnvironment trees<br>• Exception and message: Propagates new ExceptionList and Message trees, and the original LocalEnvironment tree<br>• Exception and LocalEnvironment: Propagates new ExceptionList and LocalEnvironment trees, and the original Message tree<br>• All: Propagates new Message, LocalEnvironment, and ExceptionList trees |
| Copy message | List | No | Specifies which parts of the input message tree are copied to the output message tree. Valid values are:<br>• Copy headers: Copies every part of the input message except the last child of root<br>• None: Does not copy any part of the input message<br>• Copy entire message: Copies the whole input message |
| Copy local environment | List | No | Specifies how the LocalEnvironment is copied to the output message. If you set it to Copy Entire LocalEnvironment, at each node in the message flow, a new copy of the LocalEnvironment is created in the tree, and it is populated with the contents of the LocalEnvironment from the preceding node. So if a node changes the LocalEnvironment, the upstream nodes do not see those changes because they have their own copies. This behavior might be an issue if you are using a FlowOrder node, or if you use the PROPAGATE statement on a Compute node.<br><br>If you set this property to None, each node does not generate its own copy of the LocalEnvironment, but it uses the LocalEnvironment that is passed to it by the previous node. Therefore, if a node changes the LocalEnvironment, those changes are seen by the upstream nodes. |
| Copy exception list | List | No | Specifies which parts of the input exception list tree are copied to the output message tree. Valid values are None and Copy entire exception list. |

## Report panel

The **Report** panel contains properties for nodes that interface with an external resource. These properties control how the metadata that is returned from the source is processed, including what information is reported, and where in the message assembly that data is placed.

The following table shows the properties that are available on the **Report** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Include report | Check box | No | Specifies whether a status report is generated. |

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Report location | String | No | Specifies which part of the message assembly is populated with the status report. |
| Report location syntax | List | No | Specifies whether the report location syntax is ESQL or XPath. The default value is ESQL. |

## Error panel

The **Error** panel contains properties for treating errors from external systems, such as how to treat the error, and where in the message assembly to place the error data.

The following table shows the properties that are available on the **Error** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Treat error as failure | Check box | No | If this check box is selected, all reported errors are handled as failures by the Failure terminal. |
| Error data location | String | No | Specifies which part of the message assembly is populated with the error data. |
| Error data location syntax | String | No | Specifies whether the error data location syntax is ESQL or XPath. The default value is ESQL. |

## Retry panel

The **Retry** panel contains properties that specify the behavior of the node when an error occurs that is not handled by the message flow.

This panel is usually found on input nodes. When the message that is propagated by the input node causes an unhandled error in the message flow, retry processing begins. When the retry mechanism is activated, the node becomes single threaded and no additional instances are allocated.

Retry processing has three modes:
- Failure: A message is tried only once before it is processed as a failure.
- Short retry followed by failure: The message is processed a specified number of times, with a pause of a specified time between each attempt, before it is processed as a failure.
- Short retry followed by long retry: The message is processed in the same way as for Short retry followed by failure, but when the retry count has been exceeded, the message continues to be processed using a long wait interval between each retry.

The following table shows the properties that are available on the **Retry** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Retry mechanism | List | No | The mechanism to use for retry processing. Valid values are: <br> • Failure: The message is sent automatically to the failure processing when an exception occurs. <br> • Short retry: The message is retried using the short retry properties. When short retry processing reaches its limit, processing stops. <br> • Short and long retry: The message is retried using the short retry properties. When short retry processing reaches its limit, long retry processing is performed |

| Property | Type | Configurable? | Description |
|----------|------|---------------|-------------|
| Retry threshold | Integer | Yes | The number of times that the node attempts to propagate messages if failures occur repeatedly. This value must be a positive integer. The default value is 0, therefore the message is tried only once. |
| Short retry interval | Integer | Yes | The length of time in milliseconds to wait between retry attempts until the number of retries exceeds the Retry threshold. The default value is 0, therefore no delay occurs between each retry. |
| Long retry interval | Integer | Yes | The length of time in milliseconds to wait between attempts until a message is successful, the message flow is stopped, or the message flow is redeployed. The default value is 300,000, therefore a 5-minute delay occurs between each retry. |

## Security panel

The **Security** panel contains properties that are required for security details.

You can also control secure operation for an HTTPInput node by using SSL services for connectivity, and you can configure the Real-timeInput node to use User Name Server ACL lists to control user access by publish/subscribe topic.

The following table shows the properties that are available on the **Security** panel.

| Property | Type | Configurable? | Description |
|----------|------|---------------|-------------|
| Security required | Check box | Yes | If this check box is selected, authentication is switched on for the node. The node uses the authentication protocols that are appropriate to the function that is being performed. Separate configuration steps might be necessary before deployment, including setting up the external resource. By default, this check box is cleared. |
| Encryption required | Check box | Yes | If this check box is selected, encryption is required. By default, this check box is cleared. |
| Identity | String | Yes | An account identifier that can be used to retrieve user and resource credentials. By default, this field is empty, therefore the node does not need a specific account name, and it operates in Anonymous mode for authentication purposes. |

## Polling panel

The **Polling** panel contains properties that control an input node that is polling an external source for events to process.

The following table shows the properties that are available on the **Polling** panel.

| Property | Type | Configurable? | Description |
|----------|------|---------------|-------------|
| Polling mode | List | Yes | The polling mechanism to use. Valid values are:<br>• Properties: Uses the properties that are provided on the **Polling** panel to define when polling occurs.<br>• Control: Polling is controlled by control message that are sent to the Control terminal.<br>• Properties and control: Both mechanisms are used for polling. The control message overrides the Polling frequency. |
| Use start and finish time | Check box | Yes | Specifies whether to use start or finish times. By default, the check box is cleared, therefore the node polls at any time of the day. |

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Time zone | List | Yes | The time zone to use to calculate start and finish times. The default value is Local. |
| Start time | Time | Yes | The time to start polling if Polling mode is set to Properties. The format is hh:mm:ss, and the default value is 00:00:00. |
| Finish time | Time | Yes | The time to finish polling if Polling mode is set to Properties. The format is hh:mm:ss, and the default value is 23:59:59. |
| Polling frequency | Integer | Yes | The frequency in milliseconds at which polling is performed if Polling mode is set to Properties. The default value is 5000 milliseconds (5 seconds). |

## Transactionality panel

The **Transactionality** panel contains properties for defining how transactions are to be handled.

The following table shows the properties that are available on the **Transactionality** panel.

| Property | Type | Configurable? | Description |
|---|---|---|---|
| Transaction mode | List | Yes | Specifies how the node controls transactions. Valid values are:<br>• Automatic: The mode that is chosen depends on the transport that is being used.<br>• Yes: Transactionality is used.<br>• No: Transactionality is not used. |

---

# Terminal definitions

Terminals identify points where messages can enter and leave a node.

The four main types of terminal are:
• Input data: A terminal to send data into a node
• Output data: A terminal to send data out of a node
• Input control: A terminal to send control information into a node
• Output control: A terminal to send control information out of a node

| Name | Type | Description |
|---|---|---|
| In | Input data | The standard terminal that receives data to be processed.<br><br>The In terminal is typically connected to the output terminal of the previous node in the message flow. The message format and type information is set in the incoming message tree. A warning message is displayed in the workbench if this terminal is not connected. |
| Out | Output data | The standard terminal for sending data out of the node.<br><br>The Out terminal is typically connected to the input terminal of the next node in the message flow. The message format and type information can be set by the properties on the **Parser** panel, if it is present. |
| Failure | Output data | A message is passed to the Failure terminal if a failure occurs in the node.<br><br>The original input messages to this node, and the exception list, are propagated to the Failure terminal. This terminal is typically connected to the input terminal of a node that handles exceptions. If this terminal is not connected, the failure is passed back up the message flow until it is caught by the first node that has a Catch terminal. |

| Name | Type | Description |
|------|------|-------------|
| Error | Output data | A message is passed to the Error terminal if an error occurs when accessing an external application.<br><br>A message with the error details is sent to this terminal. Control this terminal's mode of operation using the properties on the **Error** panel. If this terminal is not connected, and the Treat error as failure property is not set, the messages are sent, with the error details, to the default output terminal. |
| Catch | Output data | A message is passed to the Catch terminal if an error happens upstream of the node.<br><br>The original input message to this node, and the exception list, are propagated out of this terminal. The Catch terminal is typically present only on input and TryCatch nodes. If this terminal is not connected, the terminal does not take control of message flow processing. The failure continues flowing back upstream to the next node that has a Catch terminal. If this node is the first node in the message flow, the current transaction is rolled back. |
| Status | Output data | An acknowledgement message is sent to the Status terminal when processing through the output terminals has completed.<br><br>The message provides information on the result of propagation to each output terminal. |
| No Message | Output data | A message is passed to the No Message terminal when the node has tried to get some data, or a message, and there is nothing to get. |
| TimeOut | Output data | A message is passed to the TimeOut terminal when the node has waited until the timeout period that is specified for the node. |
| ControlIn | Input control | A terminal to receive control information in a node.<br><br>This terminal is required only if control information cannot be sent via the In terminal. |
| ControlOut | Output control | A terminal to send control information out of a node.<br><br>This terminal is required only if control information cannot be sent via the Out terminal. |
| Trigger | Input control | A terminal to send a trigger message to cause an action in a node. |

# Part 3. Appendixes

# Appendix. 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:

*IBM World Trade Asia Corporation*
*Licensing*
*2-31 Roppongi 3-chome, Minato-ku*
*Tokyo 106-0032,*
*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.

**323**

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

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

| | | |
|---|---|---|
| AIX | CICS | DB2 |
| DB2 Connect | DB2 Universal Database | developerWorks |
| Domino | | Everyplace |
| FFST | First Failure Support Technology | IBM |
| IBMLink | IMS | IMS/ESA |
| iSeries | Language Environment | Lotus |
| MQSeries | MVS | NetView |
| OS/400 | OS/390 | POWER |
| pSeries | RACF | Rational |
| Redbooks | RETAIN | RS/6000 |
| SupportPac | S/390 | Tivoli |
| VisualAge | WebSphere | xSeries |
| z/OS | zSeries | |

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

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

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds 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