WebSphere Message Broker

User-defined Extensions

Version 6 Release 0

WebSphere Message Broker

IBM

User-defined Extensions

Version 6 Release 0

Note

Before using this information and the product it supports, read the information in the Notices appendix.

Fifth Edition (December 2006)

This edition applies to IBM® WebSphere® Message Broker Version 6.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this topic collection

This PDF has been created from the WebSphere Message Broker Version 6.0.0.3 (with Message Brokers Toolkit Version 6.0.2.0 update, December 2006) 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

Developing user-defined extensions

This section contains details on how to implement a user-defined node or parser to enhance the functionality of WebSphere Message Broker.

You can write user-defined nodes in C or Java. You can write user-defined parsers only in C. For a general introduction on user-defined extensions, read "User-defined extensions" on page 4. For information about designing and creating user-defined nodes and user-defined parsers, see the following topics:

- "Designing user-defined extensions" on page 8
- "Creating user-defined extensions" on page 33

When you have created a user-defined node, you can test it; this task is described in "Testing a user-defined node" on page 85. 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 87.

Consider the following restrictions and factors when developing user-defined extensions:

- 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 there is only one JVM 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 nodes can be deployed in WebSphere Event Broker. When creating user-defined nodes for WebSphere Event Broker users, you must ensure that you do not expose users to the ability to evaluate ESQL code. For example, nodes that expose the input to MbSQLStatement as a node property would effectively be emulating a Compute node. Use of ESQL in WebSphere Event Broker is not supported.
- User-defined input nodes can only support XML, BLOB, and the MQ parsers.
- Avoid using operating system specific functions. If you code in this way, your user-defined extensions can work on a variety of platforms without requiring changes to the source code.

The following table shows the topics that you must read based on the type of user-defined extension that you want to create.

Action	Topics to view
To use one of the Java sample nodes:	1. "Compiling a Java user-defined node" on page 75 2. "Installing user-defined extension runtime files on a broker" on page 87 3. "Creating the user interface representation of a user-defined node in the workbench" on page 79 4. "Testing a user-defined node" on page 85

Action	Topics to view
To use one of the C sample nodes:	1. "Compiling a C user-defined extension" on page 54 2. "Installing user-defined extension runtime files on a broker" on page 87 3. "Creating the user interface representation of a user-defined node in the workbench" on page 79 4. "Testing a user-defined node" on page 85
To use the sample parser:	1. "Compiling a C user-defined extension" on page 54 2. "Installing user-defined extension runtime files on a broker" on page 87
To create your own Java node using the workbench:	1. "Creating an input node in Java" on page 60 or "Creating a message processing or output node in Java" on page 66 2. "Using event logging from a user-defined extension" on page 96 3. "Compiling a Java user-defined node" on page 75 4. "Testing a user-defined node" on page 85 5. "Packaging a user-defined node workbench project" on page 89 6. "Installing a user-defined extension to current and past versions of the broker" on page 90
To create your own C node:	1. "Creating an input node in C" on page 34 or "Creating a message processing or output node in C" on page 42 2. "Using event logging from a user-defined extension" on page 96 3. "Compiling a C user-defined extension" on page 54 4. "Installing user-defined extension runtime files on a broker" on page 87 5. "Creating the user interface representation of a user-defined node in the workbench" on page 79 6. "Testing a user-defined node" on page 85 7. "Packaging a user-defined node workbench project" on page 89 8. "Installing a user-defined extension to current and past versions of the broker" on page 90
To create your own parser:	1. "Creating a parser in C" on page 49 2. "Using event logging from a user-defined extension" on page 96 3. "Compiling a C user-defined extension" on page 54 4. "Installing user-defined extension runtime files on a broker" on page 87

User-defined extensions

A user-defined extension is a component that you can design and implement to add to the function of your implementation of WebSphere Message Broker.

With WebSphere Message Broker, you can create and implement the following types of user-defined extensions:

- User-defined input nodes

- User-defined message processing nodes
- User-defined output nodes
- User-defined parsers

The user-defined nodes and parsers that you create can be used in conjunction with both the nodes and parsers supplied with the product, and with third-party supplied nodes and parsers. You can also configure a user-defined node to use a user-defined parser that you have written rather than one of the supplied parsers.

A user-defined parser must be written in the C programming language. User-defined nodes can be written in the C or Java programming language. User-defined nodes and parsers written in C must be compiled into a loadable implementation library, that is, a shared library on Linux and UNIX, or a Windows DLL. User-defined nodes written in Java must be packaged as a jar file. You must integrate any user-defined extension you create into the WebSphere Message Broker tooling before you can use it.

If you plan to program using the supplied Java or C language user-defined extension API, you must install the ″Samples and SDK″ optional component on at least one system. The SDK provides the required header files and contains samples that you can modify to your own requirements.

You can use your new node types on more than one operating system, if you make them platform independent. You can achieve this platform independence by using the ANSI standard C or Java programming languages, and by avoiding the use of platform specific code in your user-defined extension.

You can deploy user-defined nodes in WebSphere Event Broker. When creating user-defined nodes for WebSphere Event Broker users, you must ensure that you do not expose users to the ability to evaluate ESQL code. For example, nodes that expose the input to MbSQLStatement as a node attribute would effectively be emulating a compute node. Use of ESQL inWebSphere Event Broker is not supported.

User-defined input nodes can only support XML, BLOB and the MQ parsers. The MRM is not shipped with WebSphere Event Broker and there is no support for user-defined parsers.

For information on each type of user-defined extension that you can create, see the relevant topics in this section of the help. The topics in this section will help you understand how your user-defined extension interact with other components of WebSphere Message Broker, such as message flows and their associated execution groups. A good understanding of the broker architecture will help you to plan and construct your user-defined extensions more effectively.

Planning user-defined extensions

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 it might not be necessary to create 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, so make sure 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 would need to 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.

There are a number of general design and development considerations that you should consider and understand when you are planning or writing a user-defined node or parser, and considerations that are specific to the type of user-defined extension you want to create. You should be familiar with the concepts covered in the topics below before designing a user-defined extension.
- General design considerations
  - "Errors and exception handling" on page 8
  - "Storage management" on page 10
  - "String handling" on page 11
  - "Threading" on page 11
- Specific design considerations
  - "Planning user-defined input nodes" on page 19
  - "Planning user-defined message processing nodes" on page 24
  - "Planning user-defined output nodes" on page 28
  - "Planning user-defined parsers" on page 31

User-defined extensions in the runtime environment

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

You should ensure that you are familiar with the following runtime components and concepts:
- Runtime environment
- Broker domains
- Configuration Manager
- Brokers
- Execution groups
- "Execution model" on page 7

You should also make sure you understand the following concepts:
- Message flows overview

When you have gained an understanding of the WebSphere Message Broker runtime environment, you need to understand how any user-defined extensions that you develop will interact with the components in that environment. The following topics will help you understand how your user-defined extension interacts with the WebSphere Message Broker runtime components.

- “C user-defined input node life cycle” on page 16
- “Java user-defined input node life cycle” on page 18
- “C user-defined message processing nodes life cycle” on page 21
- “Java user-defined message processing nodes life cycle” on page 23
- “User-defined output node life cycle” on page 28
- “User-defined parser life cycle” on page 29

Execution model

The WebSphere Message Broker execution model is the system used to execute message flows through a series of nodes.

When an execution group is initialized, the appropriate LILs are made available to the runtime. The Execution Group runtime process starts, and spawns a dedicated configuration thread. In the message flow execution environment, the message flow is thread-safe. You can concurrently run message flows on many threads, without having to consider serialization issues. Any user-defined nodes that you implement should not compromise this threading model. Note the following points:

- An input message sent to a message flow is only processed by the thread that received it. No thread or context switching takes place during message processing
- Data structures accessed by message flows are only visible to a single thread, and these data structures exist only for the lifetime of the message being processed.
- A single instance of a message flow is shared between all the threads in the message flow thread pool. This is related to the behavior of a message flow node in that it does not have state.
- The memory requirements of an Execution Group are not unduly affected by running message flows on more OS threads.
- The message flow execution environment is conceptually similar to procedural programming. Nodes that you insert into a message flow are like subroutines called using a function call interface. However, rather than a ″call-return″ interface, in which parameters are passed in the form of input message data, in WebSphere Message Broker the execution model is referred to as a ″propagation and return″ model.
- A WebSphere Message Broker message flow is inherently thread-safe, and message flows can be run concurrently on more than one thread

If, for example, you are using a user-defined node to process messages, and you are also using a user-defined parser to parse the incoming messages, both the node and parser will contain implementation functions. The broker calls these implementation functions, or callbacks, when certain events occur.

When an input message is received into the broker at that input node, it is sent to the user-defined node.

- For C nodes, the broker calls the cniEvaluate function for the user-defined node. See “cniCreateNodeContext” on page 122 for information on the cniEvaluate function.
- For Java nodes, the broker calls the evaluate method that is implemented by the user-defined node.

If the node wants to query the message to decide what to do with it, it calls a C utility function or a Java method, as appropriate for the language in which the node is written. The broker then invokes the user-defined parser on one of its implementation functions. This instructs the parser to start building the WebSphere Message Broker parse tree. The parser starts building the tree by invoking utility functions that create elements in the parse tree. The parser can be called multiple times by the broker, rather than just once.

Designing user-defined extensions

When you are writing user-defined extensions, there are a number of general planning and design issues that you should bear in mind. These issues are covered in the topics in this section.

The topics in this section deal mainly with design issues 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 should consider these as you would normally when developing Java applications.

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

This topic deals with issues relating to storage management 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 storage management methods.

All memory allocated by a user-defined extension must be released by the user-defined extension. The construction of a node at run-time causes cniCreateNodeContext 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 message 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 message broker will pass addresses of C++ objects to the user-defined extension. These are simply intended to be used as a handle to be passed back on subsequent function calls. You should not allow your C user-defined extension to try to manipulate or use this pointer in any way, by trying to release storage using the free function, for example. Such actions will cause unpredictable behavior in the message broker.

The cniCreateNodeContext implementation function is invoked whenever the underlying node object has been constructed internally. This occurs when a broker is defined with a message flow that uses a user-defined node. It is important to note that this is not necessarily the same activity as creating (or reusing) a thread to execute a message flow instance containing the node. In fact, the cniCreateNodeContext function will be 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

This topic deals with issues relating to string 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 string handling methods.

To enable a WebSphere Message Broker 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

Message processing nodes and parsers must work in a multi-instance, multithreaded environment. There can be many node objects or parser objects each with many syntax elements, and there can be many threads executing methods on these objects. The message broker design ensures that a message object and any objects it owns are used only by the thread that *receives and processes* the message through the message flow.

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. For parsers, an instance of a parser is used only by a single message flow thread.

A user-defined extension should adhere to this model, and should avoid the use of global data or resources that require semaphores to serialize access across threads. Such serialization can result in performance bottlenecks.

User-defined extension implementation functions must be reentrant, and any functions they invoke must also be reentrant. All user-defined extension utility functions are fully reentrant.

Although a user-defined extension can spawn additional threads if required, it is essential that the *same* thread returns control to the broker on completion of an implementation function. Failure to do this will compromise the integrity of the broker and will produce unpredictable results.

Execution model: When an execution group is initialized, the appropriate lils are made available to the runtime. The Execution Group runtime process starts, and spawns a dedicated configuration thread. In the message flow execution environment, the message flow is thread-safe. You can concurrently run message flows on many OS threads, without having to consider serialization issues. Any user-defined nodes that you implement should not compromise this threading model. Note the following points:

- An input message sent to a message flow is only processed by the thread that received it. No thread or context switching takes place during message processing
- Data structures accessed by message flows are only visible to a single thread, and these data structures exist only for the lifetime of the message being processed.
- A single instance of a message flow is shared between all the threads in the message flow thread pool. This is related to the behavior of a message flow node in that it does not have state.
- The memory requirements of an Execution Group are not unduly affected by running message flows on more OS threads.

Threading model: The following message flow example will help you understand some of the threading considerations you should be aware of when designing and implementing your own user-defined nodes. It considers the example of a user-defined input node.

A message flow can be configured to run on a set of threads. This is determined by the number of input nodes in the message flow, and the value of the additionalInstances property of the message flow. These two elements determine the maximum capacity of the thread pool the message flow can use. Therefore, if your message flow has particular processing requirements that dictate single threaded execution, you need to ensure that this is the case.

A typical order of events for input node processing looks like this:

1. Input node construction takes place
2. A thread is demanded from the thread pool
3. The allocated thread starts in the node’s run method
4. Configuration (or reconfiguration) is committed
5. Initialization processing is performed on the thread’s context
6. The thread connects to the broker’s queue manager
7. A message group and buffer object are created

8. A queue open request for the input queue is sent to the queue manager. This queue is kept open for the duration of the thread's lifetime.
9. The input node enters a message processing loop
10. When a message is received, the data buffer contains the header and body of the message
11. Message objects are created and attached to the thread's group
12. Thread dispatching is activated if multiple threads are specified
13. Message data is propagated downstream.

You should note the following:
- Your input node will implement the chosen message flow threading model.
- Your input node will always have at least one thread either reading from its input source or actively processing a message received by it. If a message flow has multiple input nodes, then any additional thread instances are available to service any of the input nodes, as determined by the dispatching policy of that input node.

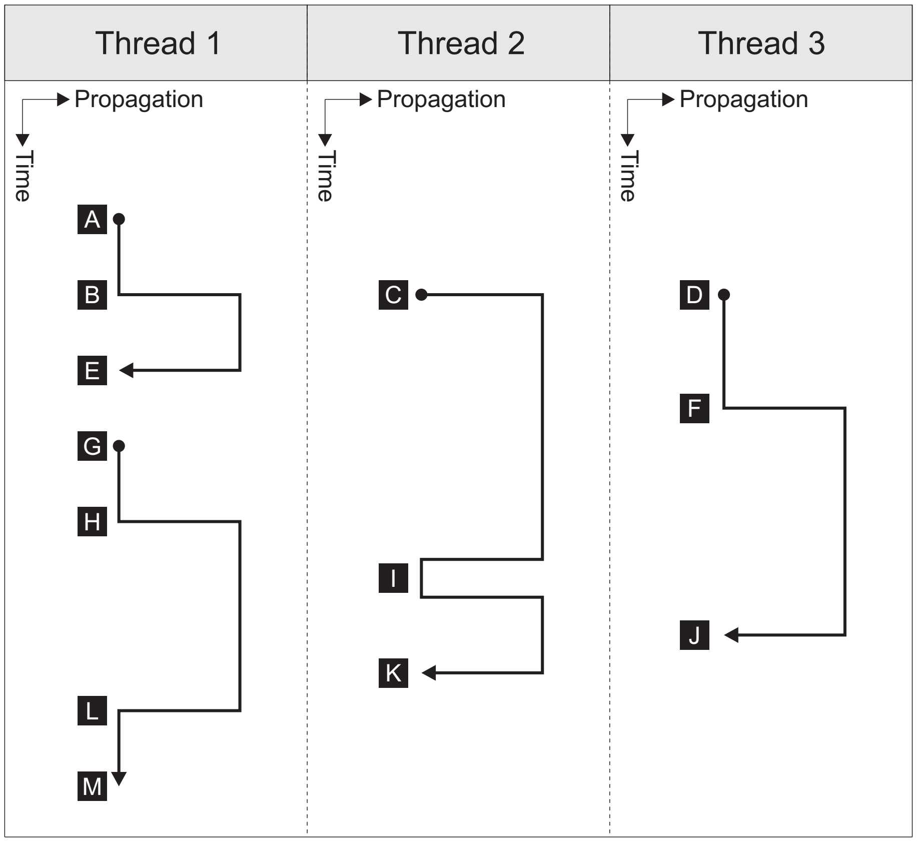
Threads can be demanded or requested. When your message flow is deployed, the input node demands an initial thread. Although the message flow has a pool of threads associated with it, it is the input node that is responsible for the dispatching policy. This means that it always needs to ensure that one instance of itself is running on a thread. Because the default value of the additionalInstances property is zero, any further requests for a thread will fail if you have defined multiple input nodes. This means that it is possible for a message flow to consume more threads that you expect. Also, this could mean that if you have defined multiple input nodes, one of the input nodes could be starved of threads.

Allowing the broker to start additional copies of the message flow in separate threads using the additionalInstances property is the most efficient way of preventing the input queue from becoming a bottleneck. However, creating separate threads allows parallel processing of messages from the message queue, so you should only use this property when the order messages are processed is not important.

Threads are created as a result of input node construction and operation. A thread remains active or idle in the thread pool, and idle threads remain in the pool until they are dispatched by an input node, or the broker is shut down.

The figure below illustrates thread allocation in a message flow.

Thread allocation in a message flow



Initially, Thread 1 is demanded (A), and waits for messages. When a message arrives (B), Thread 1 propagates the message, and dispatches Thread 2. Thread 2 receives a message immediately (C), and propagates, and dispatches Thread 3, which waits for a message (D). Thread 1 completes (E), and returns to the thread pool. Thread 3 then receives a message (F), dispatches Thread 1, and propagates the message. Thread 1 now waits for a message to arrive on the queue (G).

It is worth noting the point marked H. At this instance in the message flow, Thread 1 acquires a message, but because all other threads are active at that time, it cannot dispatch. The message is propagated.

After this message has been propagated, Thread 2 completes (I), receives a new message from the queue, and propagates this new message. Thread 3 then completes (J), and returns to the pool. Thread 2 then also completes (K). Because it did not dispatch, when Thread 1 completes (L), it cannot return to the thread pool even though there are no messages on the queue, and instead waits for a message to arrive on the queue.

Note the following points about thread behavior in WebSphere Message Broker:

- Threads are only created if required by the workload. This means that an execution group process can use fewer threads than it has been configured for.
- Unless all available threads are actively processing within a message flow, one thread will always be reading the input queue.
- If the workload increases at any point, other input nodes in the same message flow will be able to acquire threads that have been returned to the thread pool.

If a thread acquires a message, but all other threads are active at the time, it cannot dispatch. The message is propagated. When this thread completes, because it did not dispatch, it cannot return to the thread pool even though there are no messages on the queue.

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
- cniSqlExecute
- cniSqlSelect
- cniSqlDeleteStatement

Java Database Connectivity

Type 4 JDBC drivers are supported.

Node and parser factory behavior

This topic provides information on the role of the node factory and parser factory for declaring a node to the broker or defining a parser.

Each 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, the following functions are called:

- **`bipGetMessageflowNodeFactory`**

  The initialization function, `bipGetMessageflowNodeFactory`, is called by the broker after the LIL has been loaded and initialized by the operating system. 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`**

  The initialization function, `bipgetparserfactory`, is called by the broker after the LIL has been loaded and initialized by the operating system. The `bipgetparserfactory` function defines the name of the factory that the user-defined parser supports and the classes of objects, or shared object, supported by the factory. `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 following functions are called:

1. **`cniCreateNodeFactory`**

   This function creates a single instance of the node factory in the message broker.
2. **`cndDefineNodeClass`**

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

Before the parser factory is returned, the following functions are called:

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 is supported by a parser factory, and identifies the parsers that the factory can create.

See the following topics for information on these functions:
- "cniCreateNodeFactory" on page 123
- "cpiCreateParserFactory" on page 187
- "cniDefineNodeClass" on page 124
- "cpiDefineParserClass" on page 189

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 within a message 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, you should make sure you are familiar with the concepts introduced in "Planning user-defined extensions" on page 5 and "User-defined extensions in the runtime environment" on page 6.

C user-defined input node life cycle

This topic guides you through the various stages in the life of a user-defined input node written using the C programming language. It covers the following stages in an input node's life cycle:
- 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.

  Note: Your user-defined input node will only have input terminals if it is also acting as a message processing node. If this is the case, it is usually 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. This 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 will remain 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 mqsistop 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

This topic guides you through the various stages in the life of a user-defined input node written using the Java programming language. It covers the following stages in an input node's life cycle:

- 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 constructor of the input 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.

If you want to handle exceptions that are passed back to your input node, you should use createOutputTerminal to create a catch terminal for your input node. When the input node catches an error, the catch terminal will process it in the same way that a regular MQInput node would. You can allow most exceptions, such as exceptions 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 input 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, then 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. The recommended model is to make sure that threads are dispatched after a message has been created and before it is propagated. This ensures 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, then 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(), then it should be made clear to the user of that node that setting the additionalInstances property will have no effect on the input node.

For more information on the threading model for User-defined Input nodes, see "Threading" on page 11.

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

This topic outlines the planning and design considerations you should think about before developing a user-defined input node.

Analysis: Before developing a user-defined Input node, you should consider the following:

- Do you need to create a custom input node?

  You must include at least one input node in a message flow. (For more information about using more than one input node, see Using more than one input node. The 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 supplied MQInput node.
  - If the messages are sent by SCADA devices, use the SCADAInput node.

  - If the message source is any other, you must use a user-defined input node.
- To successfully input the data concerned, will the input node have to interface with third-party software? If so does the API enabling 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 co-ordinated 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
  - reset content descriptor nodes

Design considerations: Before developing and implementing your input node, you should decide on the following factors:
- The message parser that initially parses the input message.
- Whether to override the default message parser attribute values for this input node.
- The appropriate threading model for the input node.
- End of message processing and transaction support that the node supports.
- The configuration attributes required by the input node that should be externalized for alteration by the message flow designer.
- Optional node APIs the user-defined node provides.
- General development issues:
  - "Threading" on page 11
  - "Storage management" on page 10
  - "String handling" on page 11
  - "Errors and exception handling" on page 8
  - expected message formats for primitive nodes that expect specific header folders.
- When designing nodes to be run as extensions to WebSphere Event Broker, the following restrictions must be considered:
  - User-defined input nodes can only support XML, BLOB and the WebSphere MQ parsers. The MRM is not shipped with WebSphere Event Broker and there is no support for user-defined parsers.
  - User-defined nodes should not expose to users the ability to evaluate user ESQL code. For example, nodes that expose the input to MbSQLStatement as a node attribute are effectively emulating a compute node. Use of ESQL in WebSphere Event Broker is not supported.

User-defined message processing nodes

A user-defined message processing node is a node you can create to complement the primitive node types provided by WebSphere Message Broker. You can use a user-defined message processing node to provide some specific processing on receipt of a message that is not provided for by the primitive node types provided by WebSphere Message Broker.

You might want to use a user-defined message processing node in the following situations:

- If your messages need additional transformation not provided by the primitive nodes. For example, you might need a currency converter node.
- If you want to write messages into a flat file on the local system for later processing by another application or utility program.

You can use your new node types with existing primitive node types to create message flows to achieve the processing your messages require.

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

This topic provides guidance for writing your message processing node to ensure that it functions correctly. It explains how you can use your message processing node to navigate a message.

Design considerations: Before developing and implementing your message processing node, you should decide on the following:

- The message parser that will parse the message
- Whether to override the default message parser attribute values for this message processing node.
- The appropriate threading model for the message processing node
- End of message processing and transaction support that the node will support
- The configuration attributes required by the message processing node that should be externalised for alteration by the message flow designer.
- Optional node APIs the user-defined node will provide
- General development issues:
  - "Threading" on page 11
  - "Storage management" on page 10
  - "String handling" on page 11
  - "Errors and exception handling" on page 8

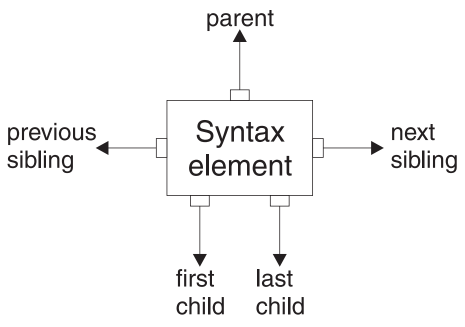
– expected message formats for primitive nodes that expect specific header folders.

Syntax element navigation: The broker infrastructure provides functions that enable a message processing node implementation to traverse the tree representation of the message, with functions and methods to allow navigation from the current element to its:

- Parent
- First child
- Last child
- Previous (or left) sibling
- Next (or right) sibling

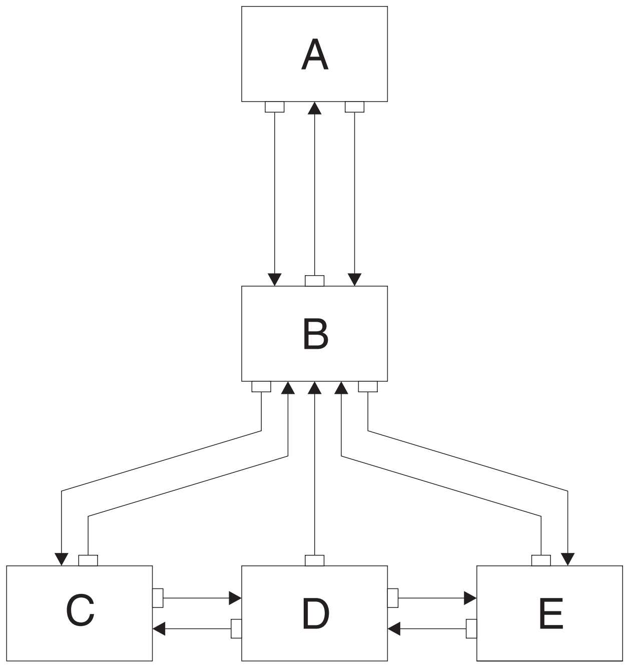
As shown in the figure below. Other functions and methods support the manipulation of the elements themselves, with functions and methods to create elements, to set or query their values, to insert new elements into the tree and to remove elements from the tree. See “C node utility functions” on page 105 and “C parser utility functions” on page 176, or the Javadoc for more information.

A syntax element with connections to other elements



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

Syntax element tree



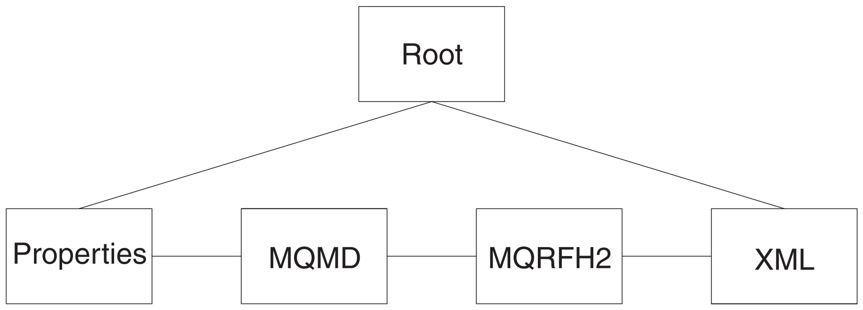
The element **A** is the *root element* of the tree. It has no parent because it is the root. It has a *first child* of element **B**. Because **A** has no other children, element **B** is also the *last child* of **A**.

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 figure below shows the first generation of syntax elements of a typical message received by WebSphere Message Broker. (Note that not all messages will have an MQRFH2 header.)

First generation of syntax elements in a typical message



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 of the incoming WebSphere MQ messages. The next sibling is the folder for the MQRFH2 header. Finally, there is the folder that represents the message content, which (in this example) is an XML message.

The figure above includes an MQMD and an MQRFH2 header. All messages received by an MQmessage processing node include an MQMD header, there are a number of other headers than can also be included.

Navigating an XML message: Suppose we have 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'

You can use node utility functions and methods (or the similar parser utility functions) to navigate through a message. Taking the XML message shown above, you need to call cniRootElement first, with the message received by the node as input to this function. In Java you need to call getRootElement on the incoming MbMessage. This returns an MbElement that represents the root of the element. The root element should not be modified by a user-defined node.

The figure above shows that the last child of the root element is the folder containing the XML parse tree. You can 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.

There is one element only (`<Business>`) at the top level of the message, so calling **cniFirstChild** (in C) or **getFirstChild** (in Java) moves to this point in the tree. You can use **cniElementType** or **getElementType** to get its type (which is `name`), followed by **cniElementName** or **getName** to return the name itself (`Business`).

`<Business>` has two children, `<Product>` and `<Company>`, so you can use **cniFirstChild** or **getFirstChild** followed by **cniNextSibling** or **getNextSibling**to navigate to them in turn.

`<Product>` has an attribute (`type='messaging'`), which is a child element. Use **cniFirstChild** or **getFirstChild**again to navigate to this element, and **cniElementType** or **getType** to return its type (which is `name-value`). Use **cniElementName** or **getName** as before 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 will be **cniElementCharacterValue**. In Java use the method **getValue**, which will return a Java object representing the element value.

`<Company>` has three children, each one having a child that is a value element (`IBM`, `Hursley`, and `WebSphere MQ`). You can 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.

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 - deliver an output message from a message flow to a WebSphere MQ queue
- MQReply - send a response to the originator of the input message.
- SCADAOutput - sends a message to a client connecting using the MQIsdp protocol
- Publication - filter output messages from a message flow and transmit them to subscribers who have registered an interest in a particular set of topics.

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

For information on planning user-defined output nodes you should read the corresponding topic for user-defined message processing nodes.

The information in that 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 construct output nodes and message processing nodes in a similar way, and you can use either type of node to perform either function.

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

In addition to the parsers provided by WebSphere Message Broker, you can provide alternative and complementary message parsers that are accessible to the broker and its message processing nodes through a standard set of parsing and construction interfaces.

If you need to process messages that do not conform to any of the defined message domains you can use the C language programming interface to create a user-defined parser.

User-defined parser life cycle

This topic guides you through the various stages in the life of a user-defined message flow parser. The stages are as follows:
- Registration
- Instantiation
- Processing
- Destruction

This topic will help you understand 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 cause, occur during, and after of each stage, and the APIs that are called. Understanding the concepts in this topic will allow 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 is triggered by the initialization phase of the start-up of the execution group process.

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 MQMD. The broker starts, loads the Loadable Implementation Library (LIL), and the parser factory. 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.

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 broker then makes a call to cpiParseBuffer. The purpose of cpiParseBuffer at this stage is to perform any necessary initialization, and to return 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 message broker. This 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: The purpose of the processing phase is to manipulate, alter, and reference elements within a message object that the parser is interpreting. 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 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 occurs when an operation within the broker requires a syntax element tree to be built or extended.

You should be aware of 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. This means that you have available a finite set of navigations
- 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 which direction to navigate in, and the order in which the navigational parser implementation functions are invoked. The user-defined parser has no control over this, and needs to respond correctly to the chosen navigation scheme. This could mean parsing right to left, as well as left to right, for instance.
- When writing a user-defined parser, it is expected that you place the actual 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 sample parser supplied with WebSphere Message Broker demonstrates this.

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. This 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 concepts you should consider before developing a user-defined parser. When you are ready, use the instructions in "Creating a parser in C" on page 49 to construct your parser.

Analysis: Before you start to create your own parser, be clear about what you want it for. You can perform most tasks using the functions already 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, you need to consider the following:

- Do you need to create a user-defined parser?

  Most tasks you need to perform can be performed using functions provided with WebSphere Message Broker. You should make sure that the task you want to perform cannot be done using built-in WebSphere Message Broker function before creating your own parser. If the available parsers in WebSphere Message Broker are not appropriate for your needs, you can 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?

  For details of message domains that the supplied WebSphere Message Broker parser can accept input messages in, and message headers that the supplied parser can work with, see Parsers.
- Does the syntax of the in-house or commercial message dictate a format that can be parsed?
- To successfully parse the message concerned, will the parser have to interface with third-party software ? If so does the API enabling 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 which 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 only parse relevant fields in a message. Using partial parsing can save system resources.

Partial and full parsing: WebSphere Message Broker supports what is called 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. Because an individual message flow might reference only a few of these fields, or none at all, 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 does not prevent a parser from processing the entire message all at once, and some parsers are written to do exactly this.)

Each syntax element in a logical message has two bits that indicate whether or not all the elements on either side of an element are complete, and whether its children are complete as well. Parsing is normally completed in a bottom to top, left to right manner. When a parser has completed 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 a 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. WebSphere Message Broker 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, then 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, also known as lazy 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 very much depends on how you anticipate the message will be processed within WebSphere Message Broker. If most field elements within the message are likely to be accessed during processing within WebSphere Message Broker, then performing a full parse of the message when an attempt is made to access it will probably prove more efficient. This is more likely to be true for smaller messages.

However, if most field elements within the message are not likely to be accessed during processing within WebSphere Message Broker, then performing a lazy parse of the message when an attempt is made to access a specific field would probably prove more efficient. This is especially true as 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.

Implementing the provided 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 101 and “Sample parser files” on page 103 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 75. For information on how to compile a C node or parser, see “Compiling a C user-defined extension” on page 54.
2. Install the user-defined extension on a broker domain. For instructions on completing this step, see “Installing user-defined extension runtime files on a broker” on page 87
3. On the Windows machine hosting the workbench, unzip 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 them. 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.

Creating user-defined extensions

You can write user-defined nodes in C or Java. You can write user-defined parsers in C only. For information on designing and creating user-defined nodes and user-defined parsers, see the following topics:

- “Creating a user-defined extension in C” on page 34

- “Creating a user-defined extension in Java” on page 60

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, and allows the new node to be included in message flows. This additional task is described in “Creating the user interface representation of a user-defined node in the workbench” on page 79. This step is not required for user-defined parsers.

When you have created your user-defined extensions, continue with the following tasks:
- “Testing a user-defined node” on page 85
- “Packaging and distributing user-defined extensions” on page 87

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 42
- “Creating a parser in C” on page 49
- “Compiling a C user-defined extension” on page 54

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 79
- “Testing a user-defined node” on page 85
- “Packaging and distributing user-defined extensions” on page 87

Creating an input node in C

Before you start

Ensure that you have read and understood the following topics:
- “Planning user-defined extensions” on page 5
- “Designing user-defined extensions” on page 8
- “User-defined input nodes” on page 16

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

The implementation functions that have to be written by the developer are listed in “C node implementation functions” on page 104. The utility functions that are provided by WebSphere Message Broker to aid this process are listed in “C node utility functions” on page 105.

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.

This topic describes the steps you need to take to create an input node using C. It outlines the following steps:

1. "Declaring and defining the node"
2. "Creating an instance of the node" on page 36
3. "Setting attributes" on page 37
4. "Implementing the node functionality" on page 37
5. "Overriding the default message parser attributes (optional)" on page 38
6. "Deleting an instance of the node" on page 39

Declaring and defining the node:

To declare and define a user-defined node to the broker you must 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 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 the broker should call the LIL. For example:
   ```
   CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix
   bipGetMessageflowNodeFactory()
   ```
2. The bipGetMessageflowNodeFactory function must then 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) passed back must be unique throughout all the LILs in the broker.
3. The LIL must then 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 capability of an input node by implementing the cniRun implementation function.

User-defined nodes have to either implement a cniRun or a cniEvaluate implementation function, otherwise 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* destinationList,
  CciMessage* exceptionList,
  CciMessage* message
){
  ...
  /* Get data from external source */
  return CCI_SUCCESS_CONTINUE;
}
```

The return value should be used to return control periodically to the broker.

When a message flow containing a user-defined input node is deployed successfully, the node's cniRun function is called for each message propagated to the node. For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 275.

Creating an instance of the node:

The following procedure shows you how 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. If you have three message flows that 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 does not typically 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 275.

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) ;
}
```

There is no limit to the number of configuration attributes that a node can have. However, a plug-in node must not implement an attribute that is already implemented as a base configuration attribute. These base attributes are:

- label
- userTraceLevel
- traceLevel
- userTraceFilter
- traceFilter

Implementing the node functionality:

When the broker knows that it has an input node, it calls the cniRun function of this node at regular intervals. The cniRun function must then decide what course of action it 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 normally 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 the following attributes that are included as default, 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. (See MQInput node for more information about 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.

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 MQInputNode only reads data from WebSphere MQ queues, so this data always has an MQMD at the beginning, and the MQInputNode 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, say "MYPARSER", you 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, it should free any used memory and release any held resources. You do this using the cniDeleteNodeContext function. For example:

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

  return;
}
```

Extending the capability of a C input node:

Before you start

Ensure that you have read and understood the following topic:
- "Creating an input node in C" on page 34

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 41

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 connections, as long as the output from the node is in the correct format. For connections to queues or databases, you should use the IBM primitive nodes and the API calls supplied, principally because the primitive nodes are already set up for error handling. Do not use the mqget or mqput commands 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 representing 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 has a responsibility to 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 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 there could be data available for the function to process on the new thread.

The term *transaction* is used generically here 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, any resource updates cannot be automatically committed by the broker. 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. Any unhandled exceptions are caught by the broker infrastructure, and the transaction is rolled back.

The following table describes each of the supported return values, the affect 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 waiting for an input message.	Calls the same thread again in the cniRun function.

The following 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, you have to decide what message flow data you want to propagate, and what terminal is to receive the data.

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

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

```
  if (terminalObject) {
    if (cniIsTerminalAttached(&rc, terminalObject)) {
      if (rc == CCI_SUCCESS) {
        cniPropagate(&rc, terminalObject, destinationList, 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 from occurring.

Creating a message processing or output node in C

Before you start

Ensure that you have read and understood the following topics:
- "Planning user-defined extensions" on page 5
- "Designing user-defined extensions" on page 8
- "User-defined message processing nodes" on page 20
- "User-defined output nodes" on page 28

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 User-defined Extension sample which demonstrate the use of user-defined nodes, including a message processing node written in C.

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

The implementation functions that have to be written by the developer are listed in "C node implementation functions" on page 104. The utility functions that are provided by WebSphere Message Broker to aid this process are listed in "C node utility functions" on page 105.

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. There is also the User-defined Extension sample sample for you to use.

Conceptually, 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. However, when you code a message processing node or an output node, they are essentially the same thing. You can perform message processing within an output node, and likewise you can output a message to a bit stream using a message processing node. For simplicity, this topic mainly refers to the node as a message processing node, however, it discusses the functionality of both types of node.

The functions of both types of node are covered in this topic. It outlines the following steps:
1. "Declaring and defining your node"
2. "Creating an instance of the node" on page 44
3. "Setting attributes" on page 45
4. "Implementing the node functionality" on page 45
5. "Deleting an instance of the node" on page 45

Declaring and defining your node:

To declare and define a user-defined node to the broker you must 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 initialization function, bipGetMessageflowNodeFactory, 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 the broker should call the LIL. For example:

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

2. The bipGetMessageflowNodeFactory function must then 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 the broker.
3. The LIL must then 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 providing the capability of a message processing or output node by implementing the cniEvaluate function. User-defined nodes have to either implement 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, global environment, local environment, and exception list.

For example:

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

For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 275.

Creating an instance of the node:

The following procedure shows you how 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. If you have three message flows that 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. 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 should be made to cniCreateInputTerminal and cniCreateOutputTerminal to define the user node's terminals. These functions must be invoked 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) ;
    }
```

For the minimum code required to compile a C user-defined node, see "C skeleton code" on page 275.

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. The user code needs to store these attributes in its node context area, for use when processing messages later.

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) ;
}
```

There is no limit to the number of configuration attributes that a node can have. However, a plug-in node must not implement an attribute that is already implemented as a base configuration attribute. These base attributes are:

- 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 should 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:

In the event of a node being deleted, the broker calls the cniDeleteNodeContext function. This function must free up all 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:

Before you start

Ensure that you have read and understood the following topic:
- "Creating a message processing or output node in C" on page 42

After you have created a user-defined node, the following options are available:
1. "Accessing message data"
2. "Transforming a message object" on page 47
3. "Accessing ESQL" on page 47
4. "Propagating a message" on page 48
5. "Writing to an output device" on page 48

Accessing message data:

In many cases, the user-defined node needs to 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 105 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 bitstream 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 bitstream representation of an element tree you can use the cniElementAsBitstream function. Using this function, you can obtain the bitstream representation of any element in a message. See “cniElementAsBitstream” on page 128 for details of how to use this function and sample code.

Transforming a message object:

The received input message is read-only, so 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 generally 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 it might be necessary to create a new message body. To create a new message body, the following functions are available:

```
cniCreateElementAsFirstChildUsingParser
cniCreateElementAsLastChildUsingParser
cniCreateElementAfterUsingParser
cniCreateElementBeforeUsingParser
```

These functions should be used because they are specific for assigning a parser to a message tree folder.

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.XML.Result[] = (SELECT T.C1 AS Col1 FROM Database.TABLE AS T;");
  ...
  cniSqlSelect(&rc, sqlExpr, destinationList, 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, see Compute node.

Propagating a message:

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

1. If the message has changed, you should 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, destinationList, 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 from occurring.
3. If you created a new output message using cniCreateMessage, after propagating the message, you must delete the output message using the cniDeleteMessage function:
   ```
    cniDeleteMessage(&rc, outMsg);
   ```

Writing to an output device:

A transformed message needs to be serialized to a bit stream. The bit stream can then be accessed and written to an output device. You 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.

Note that a message can be serialized only once.

Note: You must use the supplied MQOutput node when writing to WebSphere MQ queues, because 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, and this would be adversely affected if WebSphere MQ MQI calls were used in a user-defined output node.

Creating a parser in C

Before you start

Ensure that you have read and understood the following topics:
- "Planning user-defined extensions" on page 5
- "Designing user-defined extensions" on page 8
- "User-defined parsers" on page 29

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 have to be written by the developer are listed in "C parser implementation functions" on page 175. The utility functions that are provided by WebSphere Message Broker to aid this process are listed in "C parser utility functions" on page 176.

WebSphere Message Broker provides the source for a sample user-defined parser called BipSampPluginParser.c. This 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. They are described in the following sections:
1. "Declaring and defining the parser"
2. "Creating an instance of the parser" on page 51
3. "Deleting an instance of the parser" on page 51

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 the broker should call the LIL. For example:
   ```
   CciFactory LilFactoryExportPrefix * LilFactoryExportSuffix
   bipGetParserFactory()
   ```
2. The bipGetParserFactory function must then call 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 must then call 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 then 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:

The following procedure shows you how to instantiate your 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 should free any used memory and release any held resources. You do this 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 parser:

Before you start

Ensure that you have read and understood the following topic:

- “Creating a parser in C” on page 49

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 53
- "Implementing output functions" on page 53
- "Implementing a message header parser" on page 54

Implementing the parser functionality:

A parser needs to implement the following types of implementation function:
1. input functions
2. parse functions
3. output functions

Each type of function is described below.

Implementing input functions:

There are three *input* functions:
- "cpiParseBuffer" on page 205
- "cpiParseBufferEncoded" on page 206
- "cpiParseBufferFormatted" on page 207

Your parser must implement one, and only one, of these input functions.

The broker will invoke 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, then 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 to not 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:

There are three *output* functions:
- "cpiWriteBuffer" on page 227
- "cpiWriteBufferEncoded" on page 228
- "cpiWriteBufferFormatted" on page 229

Your parser must implement one, and only one, of these output functions.

The broker will invoke 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 this tree needs to be output by, for example, an MQOutput node, 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:

Normally, the incoming message data is of a single message format, so 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 needs to navigate down a chain of parser classes for a message comprising 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

Before you start

You must have a user-defined extension written in C. This C program can be one of the provided sample nodes described in "Sample node files" on page 101, the sample parser described in "Sample parser files" on page 103, 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 42, "Creating an input node in C" on page 34, or "Creating a parser in C" on page 49. The name of the user-defined node must be in the form `<nodename>.lil`.

This section provides information on how to compile user-defined extensions for all supported platforms.

The filenames used in these instructions are those of the supplied samples. If you are compiling your own user-defined extensions, you must replace these filenames with your own filenames.

Prerequisites:

Before you compile your user-defined extension, make sure you have an appropriate compiler that is supported by your operating system. Examples of appropriate compilers are shown below:

Windows **Windows**
: Microsoft Visual C++ .NET 2003

AIX **AIX**
: VisualAge® C++ for AIX Version 6.0

HP-UX **HP-UX**
: HP ANSI C 03.52

Linux **Linux (x86 platform)**
: Gnu g++, version 3.2 or later

Linux **Linux (zSeries platform)**
: Gnu g++, version 3.2 or later

Linux **Linux (POWER platform)**
: Gnu g++, version 3.3 or later

Solaris **Solaris (SPARC platform)**
: Sun Studio, Forte Developer for HPC 6 update 2

Solaris **Solaris (x86-64 platform)**
: Sun Studio 10

z/OS **z/OS**
: IBM z/OS C/C++, for z/OS version 1.5 or later

Header files:

The C interfaces are defined by the following header files:

BipCni.h
: Message processing nodes

BipCpi.h
: Message parsers

BipCci.h
: Interfaces common to both nodes and parsers

BipCos.h
Platform-specific definitions

Existing customer or third-party supplied user-defined extension libraries run on a broker with no modification or recompilation. However, you do have to create them manually in the workbench.

Compilation:

Compile the source for your user-defined extension on each of the supported platforms to create the loadable implementation library (LIL) file that the broker needs to implement your user-defined extension.

Move to the directory where the user-defined extension code is located. For example:

```
  cd install_dir\sample\extensions\nodes\  (Windows)

  cd install_dir/sample/extensions/nodes  (Linux and UNIX platforms)
```

Compiling on Windows:

Windows Compile the user-defined extension using the following command, ensuring that you include a space between `SwitchNode.c` and `BipSampPluginUtil.c`, and also between `-link` and `/DLL`:

```
  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
```

This example assumes that you are using the Microsoft 32-bit C/C++ Compiler, available in Microsoft Visual Studio C++ Version 7.1.

Because of the length of this command, it is shown over several lines.

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 Linux:

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

```
/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.

Building the C plug-in with g++ requires some changes; you must define the interface function as a C style function to the C++ compiler. In the following example, the ifdefs are needed to keep your code portable and hide the `extern "C"` directives from a C compiler.

```
#ifdef __cplusplus
  extern "C" {
  #endif
  void LilFactoryExportPrefix * LilFactoryExportSuffix bipGetParserFactory()
  {
  ...
  ...
  }
  #ifdef __cplusplus
  }
  #endif
```

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 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, you can 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. If you want to create optimized builds, use **-2** in place of **-g** in the following commands:

```
cc -c \
 -Wc,DLL -g -W0,long,langlvl\(extended\),EXPORTALL,TARGET\(OSV2R8\),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,TARGET\(OSV2R8\),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,TARGET\(OSV2R8\),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,TARGET\(OSV2R8\),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
```

Set the file permissions of the user-defined extension to group read and run the program by issuing the following command:

```
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"
- "Creating a message processing or output node in Java" on page 66
- "Compiling a Java user-defined node" on page 75
- "Packaging a Java user-defined node" on page 76

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 79
- "Testing a user-defined node" on page 85
- "Packaging and distributing user-defined extensions" on page 87

Restrictions when creating Java nodes

Within 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

Before you start

Ensure that you have read and understood the following topics:
- "Planning user-defined extensions" on page 5
- "Designing user-defined extensions" on page 8
- "User-defined input nodes" on page 16

A Java user-defined node is distributed as a .jar file. This topic describes the steps you need to take to create an input node using Java. It outlines the following steps:
1. "Creating a new Java project" on page 61
2. "Declaring the input node class" on page 61

3. “Defining the node constructor” on page 62
4. “Receiving external data into a buffer” on page 62
5. “Propagating the message” on page 62
6. “Controlling threading and transactionality” on page 63
7. “Declaring the node name” on page 64
8. “Declaring attributes” on page 64
9. “Implementing the node functionality” on page 65
10. “Overriding default message parser attributes (optional)” on page 65
11. “Deleting an instance of the node” on page 65

Creating a new Java project:

You can create Java nodes from within the workbench. To do this, you must create a new Java project, as follows:
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 then develop the source for your Java node within 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
{
...
```

You can do this in the workbench as follows:
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 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, as long as the output from the node is in the correct format.

You provide an input buffer (or bit stream) to contain input data, and associate it with a message object. You 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 theJava 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 then propagate it to one of the node’s terminals.

For example, to propagate the message assembly to the ″out″ terminal :

```
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 will not necessarily be 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 execution 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. Any unhandled exceptions are caught by the broker infrastructure, and the transaction is rolled back.

You determine the behavior of transactions and threads using an appropriate return value from the following:

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 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 additionalInstances attribute 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 need to declare the name of the node as it will be identified by the workbench. All node names must end with ″Node″. You 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 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 ″ComIbmPluginsamplesBasicInputNode″:

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

Declaring attributes:

You declare node attributes in the same way as Java Bean properties. You are responsible for writing getter and setter methods for the attributes, and 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, so 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:

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 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 the following attributes that are included as default, 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. (See MQInput node for more information about 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 shutdown the broker.
- You remove the node or the message flow containing the node, and redeploy the configuration.

During node deletion, the node might want to be informed so that it can perform any cleanup operations, such as closing sockets. If the node implements the optional onDelete method, this is called by the broker just before the node is deleted.

You implement the onDelete method as follows:

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

Creating a message processing or output node in Java

Before you start

Ensure that you have read and understood the following topics:

- "Planning user-defined extensions" on page 5
- "Designing user-defined extensions" on page 8
- "User-defined message processing nodes" on page 20
- "User-defined output nodes" on page 28
- "Restrictions when creating Java nodes" on page 60

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.

Conceptually, 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. However, when you code a message processing node or an output node, they are essentially the same thing. You can perform message processing within an output node, and likewise you can output a message to a bit stream using a message processing node. For simplicity, this topic mainly refers to the node as a message processing node, however, it discusses the functionality of both types of node.

The functions of both types of node are covered in this topic. It outlines the following steps:

1. "Creating a new Java project"
2. "Declaring the message processing node class" on page 67
3. "Defining the node constructor" on page 67
4. "Accessing message data" on page 68
5. "Transforming a message object" on page 68
6. "Propagating the message" on page 69
7. "Declaring the node name" on page 69
8. "Declaring attributes" on page 70
9. "Implementing the node functionality" on page 70
10. "Deleting an instance of the node" on page 71

Creating a new Java project:

You can create Java nodes from within the workbench. To do this, you must create a new Java project, as follows:

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 then 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 is passed 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
```

You can do this in the workbench as follows:

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 **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. This is where you create the terminals of the node, and initialize any default values for attributes.

A message processing node has a number of input terminals and output terminals associated with it. The methods createInputTerminal and createOutputTerminal are used 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. Groups of utility functions are provided for use in the **evaluate** method that are used for message management, message buffer access, syntax element navigation, and syntax element access.

The **MbElement** class provides the interface to the syntax elements. For further details of the Java API, see the Javadoc.

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 a message can be transformed, 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. New elements are generally in a parser's domain.

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

For example, where 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 it might be necessary to create a new message body. To create a new message body, the following methods are available:

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

These methods should be used because they are specific for assigning a parser to a message tree folder.

When creating a message body, do not use the following methods because they 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, you have to decide what message flow data you want to propagate, and which of the node's terminals is to receive the data.

For example:

1. To propagate the message to the output terminal ″out″:

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

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

Declaring the node name:

You need to declare the name of the node as it will be identified by the workbench. All node names must end with ″Node″. You 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:

You declare node attributes in the same way as Java Bean properties. You are responsible for writing getter and setter methods for the attributes, and 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, so 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:

As described earlier, for message processing or output nodes, you must implement the evaluate method, defined in MbNodeInterface. This is called by the broker to process the message. This method should provide all the processing function for the node.

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 local environment
   - The global environment
   - The exception list
2. The input terminal on which the message has arrived.

For example, the following code extract shows how the evaluate method could be written:

```
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, global environment, local environment, and exception list, 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 shutdown the broker.
- You remove the node or the message flow containing the node, and redeploy the configuration.

During node deletion, the node might want to be informed so that it can perform any cleanup operations, such as closing sockets. If the node implements the optional onDelete method, this is called by the broker just before the node is deleted.

You implement the onDelete method as follows:

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

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

Before you start

Ensure that you have read and understood the following topic:

- “Creating a message processing or output node in Java” on page 66

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

1. “Accessing ESQL”
2. “Handling exceptions” on page 72
3. “Writing to an output device” on page 74

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 control transactions from the **evaluate** method in your user-defined node using ESQL:

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

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

2. Set the ESQL statement to execute:

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

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

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

3. Select the type of transaction you want from the following:

   `MbSQLStatement.SQL_TRANSACTION_COMMIT`
   Immediately commit the transaction upon execution of the ESQL statement.

   `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. Execute the ESQL statement:

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

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

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

For more information about ESQL, see ESQL overview.

Handling exceptions:

You use the **MbException** class to catch and access exceptions. The **MbException** class returns an array of exception objects representing 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 the **MbException** class could be used 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);
          }
      }
  }
```

Refer to the javadoc for more details of using the MbException class.

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.XML.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 needs to be converted back into a bitstream in your **evaluate** method. You do this using the getBuffer method in MbMessage, as follows:

```
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, so you can just return at this point.

Note: You must use the supplied MQOutput node when writing to WebSphere MQ queues, because 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, and this would be adversely affected if WebSphere MQ MQI calls were 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 281.

Compiling 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 101, or a node that you have created yourself using the instructions in either “Creating a message processing or output node in Java” on page 66 or “Creating an input node in Java” on page 60.

You can compile a Java user-defined node either from the command line, or from within the project in the workbench. This topic describes both options.

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, if it is not already active.
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. You should 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).
6. 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 87) or to package for distribution (see “Packaging a user-defined node workbench project” on page 89).

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 On Windows: *install_dir*\classes\jplugin2.jar

   Linux On Linux: *install_dir*/classes/jplugin2.jar

   UNIX On UNIX: *install_dir*/classes/jplugin2.jar

   z/OS On 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."

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 87) or to package for distribution (see "Packaging a user-defined node workbench project" on page 89).

Packaging 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 101, or a node that you have created yourself using the instructions in either "Creating a message processing or output node in Java" on page 66 or "Creating an input node in Java" on page 60.

You can package a user-defined node in two ways:

- **PAR**

  A Plug-in Archive (PAR) is the deployment unit for Java user-defined nodes. The PAR contains the user-defined node classes and, if required as dependencies, can contain JAR files. A PAR file is a compressed file with a .par file extension. The directory structure in the .par file has the following format:
  - `/classes`

    The user-defined node classes are stored in this location.
  - `/lib`

    JAR files that are required by the user-defined node are stored in this location. This directory is optional because it might not be necessary to include JAR files.

  The following procedure describes how to package an example user-defined node, *parexamplenode*. In this example, the PAR is to be contained in *par.example.parexamplenode.class* with a JAR file dependency *dependency.jar*.
  1. Create the directory structure; for example:
     - `/classes/par/example/parexamplenode.class`
     - `/lib/dep.jar`
  2. Issue a file compression command to create the PAR; for example:

     ```
     jar cvf parexample.par classes lib
     ```

  The PAR should be placed in the LIL path that is specified in "Installing user-defined extension runtime files on a broker" on page 87.
- **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.

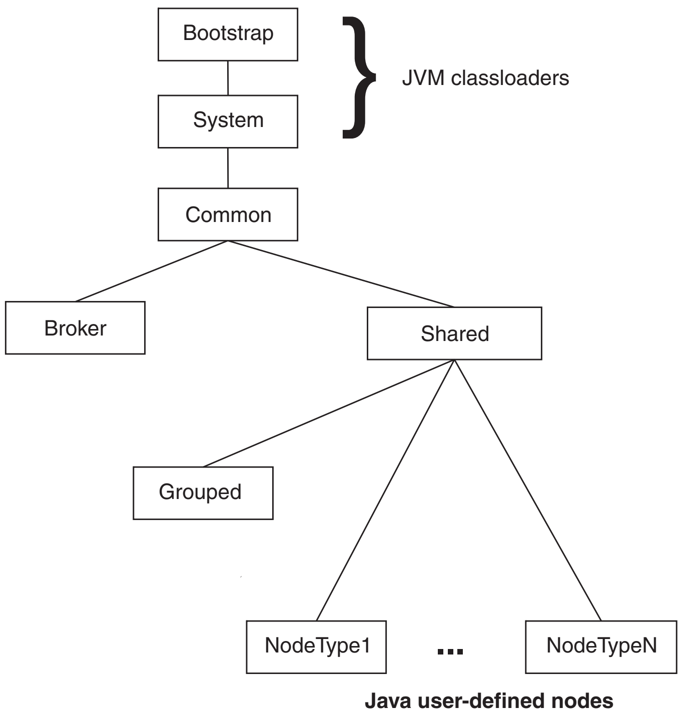
Deployment dependencies:

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:



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. These are classes that are available to all user-defined nodes within the broker.

  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.

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());
```

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

For user-defined nodes only, you must create the user interface representation of it in the workbench.

The topics below describe the steps that you must complete:

1. "Creating a user-defined node project" on page 80
2. "Creating a user-defined node" on page 81

If you have a plug-in node from Version 2.1, you must migrate the node to Version 6.0 using the mqsimigratemsgflows command, as described in Migrating a message flow, and then follow the instructions in the steps listed above.

For user-defined parsers, you have to install only the compiled .lil file. You do not manipulate parsers from within the workbench; they are only referenced by the broker from within a message flow. Therefore you do not create a user interface representation of user-defined parsers.

When you have completed the workbench representation, you must test your user-defined node

Creating a user-defined node project

The project is the container for user-defined nodes and their supporting files. You must create the project before you can create the interface for the node.

To create a new project for your user-defined node:

1. Switch to the Broker Application Development perspective.
2. Launch the New Message Flow Project wizard: Click **File> New> Project...** then **Message Flow Node Development> Message Flow Plug-in Node Project**.
3. Click **Next**. The New Message Flow Plug-in Node Project window is displayed.
4. Specify the name of the category for the nodes that you are creating. This is the name of the category under which the node appears in the palette. Either choose the name of an existing category, or enter a name to create a new category.
5. Click **Next**. The New Plug-in Project window is displayed.
6. Specify a name for your project. To be consistent with the supplied nodes, and to avoid conflict with other ISVs node project names, use your organization's Internet domain name as part of the name. For example, the name should be of the form `com.ibm.`*`nodegroup`*. You can save any number of nodes in a single project.
7. Click **Next**. The Plug-in content panel of the New Plug-in Project window is displayed.
8. Optional: Specify a plug-in runtime library name.
9. Do one of the following tasks:
   - If you intend to develop Property Editors or write a compiler class for your node, specify the JAR file name you want to use.
   - If you want to do any Java development, accept the defaults or specify the source folder in which to store your Java source.
   - If you do not want to do any Java development, you can either accept the default values or remove them.
10. Clear the ″Generate the Java class that controls the plug-in's life cycle (recommended)″ checkbox as this is not required with Eclipse Version 3 or higher.
11. Click **Next**. The Templates panel of the New Plug-in Project window is displayed.
12. Click **Finish**.
13. If there are any task-list warnings associated with the newly created project, perform the following steps.

a. Go to **Window> Preferences**
b. Expand **Plug-In Development** and select **Target Platform**
c. Click **Not In Workspace** to select all loaded plugins
d. Click **OK**
e. Select your user-defined node project in the Package Explorer, 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.

The project is stored in the file system as: <workspace path>/*project_name*

Creating a user-defined node

Before you start:

You must have created a user-defined node project before you can create a user-defined node.

To create the visual representation of your user-defined node in the workbench perform the following tasks:
1. "Creating the user-defined node plug-in files"
2. "Defining the properties"

You can also perform the following optional tasks:
- "Optional: Adding help to the node" on page 82
- "Optional: Creating node icons" on page 83
- "Optional: Adding a property editor or compiler" on page 83

Once you have created the node, you cannot move it to another folder.

Creating the user-defined node plug-in files:
1. Switch to the Broker Application Development perspective.
2. Launch the Plug-in node project wizard: Click **File> New> Project...** then **Message Flow Node Development> Message Flow Plug-in Node Project**.
3. Specify a filename for the node. The node 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 filename should be `Basic`.
4. Click **Finish** A .msgnode file for this new node is opened in the message node editor.

Defining the properties:
This topic describes how to define the properties when you are creating a user-defined node plug-in.
1. Add the terminals for your node. You must define an input terminal. Output terminals are optional. For each input terminal you want, put your cursor in the In Terminals field, and click the **Add** button. Likewise, for each output terminal you want, put your cursor in the Out Terminals field, and click the **Add** button.
2. To rename each terminal, right-click on the terminal name and click **Rename**.
3. Click on the Properties tab at the bottom of the editor area. From here you can add the node's attributes, such as a database name, a host server name, or a password. The attributes you set here must match the attributes specified in

the user-defined node itself using the `get` and `set` methods. You can also 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>=<descriptive text>`.

4. If the node is an input node, select the node name in the hierarchy and check the **Input node** checkbox. From here you can also specify that you want the node to initialize with the broker's default values by checking the **Use broker defaults** checkbox.
5. Right-click on Basic in the hierarchy and click **Add Property**. To create separate pages of properties, you can use the **Add Property Group** function.
6. Select the correct attribute type. This can be one of the built-in types, or a type to match the list of values the property can have.
7. Enter any default values. This value applies as if the flow developer had set this value themselves. It will be shown in the Properties view.
8. If you want to generate a property editor or a compiler, specify the location for these resources in the relevant field.
9. Specify the system property for each attribute that you define. This can be one of the following:
   - Hidden: The property is not displayed in the Properties view or promotion 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.
10. Close the *nodename*.msgnode file.

You can change the order in which the properties are listed by reordering them using drag-and-drop.

If you do not want to add any of the optional features, you can test your node at this point. To do this you need to launch another instance of the workbench, see "Enabling PDE runtime capabilities" on page 87, and select **Run> Run as> Runtime Workbench**. See the PDE Guide for more information about testing using the Runtime Workbench.

Optional: Adding help to the node:

1. Add online help to the node. You can create a help.html file within the project to contain the online help that explains what the node does and how to use it. If you have several files, you might want to consider creating a separate doc subdirectory in the plugin project and storing the online help files in there. You can make the node's online help appear integrated with the product-supplied information. In the WebSphere Message Broker product information, under **Reference → Message flows** there is a leaf node called ″User-defined nodes″. To make the online help for your node appear at that point, you need to:
   a. Modify the plugin.xml file to include the following extension point to the WebSphere Message Broker product documentation:
   ```
   <extension point="org.eclipse.help.toc">
   <toc file="toc.xml"/>
   </extension>
   ```
   b. Create a toc.xml file in your plugin node project, and modify the `link_to` attribute to link to the ″UDNodes″ anchor that is already defined in the WebSphere Message Broker documentation table of contents as follows:

```
<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>
```

You should then be able to see your help topic under **Reference → Message flows → user-defined nodes** in the table of contents.

The sample nodes provided with the product demonstrate this feature.

For further explanation of extension points and how to use them, see the PDE Guide.

2. Add context sensitive (F1) help to the node. This is the help that you see 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 assigns a context id based on the name of the node. You can modify the HelpContexts.xml file for your node by changing the text on 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 had 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 containing further information, for example, to the node's online help, described above. The link should be coded as follows:

   `<topic href"../plugin directory/html file" label="Link title">`

3. Add hover help (tooltip help) to the node. When a node is created, a palette.properties file is created. You can modify this file to contain hover help for your node, which is used to show 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. To do this you need to launch another instance of the workbench, see "Enabling PDE runtime capabilities" on page 87, and select **Run → Run as → Runtime Workbench**. See the PDE Guide for more information about testing using the Runtime Workbench.

Optional: Creating node icons:

When a node is created, a set of default icons are created: clc16.gif and obj16.gif are used for the node in the palette on the Broker Application Development perspective, obj30.gif is used for the node in the Message Flow Editor (that is, when it is dragged-and dropped into a message flow). To change the default icons to your own icons, replace the supplied .gif files with your files in the icons subdirectory of the plug-in project.

You can test your node at this point. To do this you need to launch another instance of the workbench , see "Enabling PDE runtime capabilities" on page 87, and select **Run> Run as> Runtime Workbench**. See the PDE Guide for more information about testing using the Runtime Workbench.

Optional: Adding a property editor or compiler:

If you need to control how the properties of your node are displayed, you can create a property editor by implementing the IPropertyEditor interface. A property editor is not limited in content. It can contain many controls, like text fields, lists, and so on.

If you want to create a custom compiler, for example to encrypt a value before sending it to the server, you can create a compiler by implementing 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. The com.ibm.etools.mft.api plug-in is selected. 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 on your user-defined node project, and click **Update Classpath**.

Creating the Java class using WebSphere Message Broker Version 6.0:
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 on 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:

If you want to test your property editor, launch another instance of the workbench, see “Enabling PDE runtime capabilities” on page 87, and select **Run> Run as> Runtime Workbench**. See the PDE Guide for more information about testing using the Runtime Workbench.

If you want to test your compiler, deploy the flow containing your user-defined node on a broker.

Testing a user-defined node

Before you start

Ensure that you have completed the following tasks:

- “Creating a user-defined extension in C” on page 34 or “Creating a user-defined extension in Java” on page 60
- “Creating the user interface representation of a user-defined node in the workbench” on page 79
- “Installing user-defined extension runtime files on a broker” on page 87

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

1. Enable the Eclipse Plug-in Development environment. This task is described in “Enabling PDE runtime capabilities” on page 87. 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 96 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. This includes all utility functions that start with `cniSetElement*`, where * represents all nodes with that stem.
- 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 message 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. This includes all utility functions that start with `cpiSetElement*`, where * represents all nodes with that stem.

  For information on the C language user-defined node API, see the "C language user-defined parser API" on page 175 and the "C language user-defined node API" on page 104.
- 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.

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 89). 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.

Packaging and distributing user-defined extensions

Before you start:

To complete this task, you must have completed the following tasks:

- "Creating user-defined extensions" on page 33
- "Testing a user-defined node" on page 85

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." For a more automated approach, see the information in "Installing a user-defined extension to current and past versions of the broker" on page 90.
- 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 79. Then install the plug-in on to 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 90. This step is not necessary for user-defined parsers.

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 75 or "Compiling a C user-defined extension" on page 54.
  - The files that have been created for extension created in C depend on the underlying broker operating system:

    Windows On Windows systems, a dynamic link library (DLL), named with a file type of '.lil'.

    Linux On Linux systems, a shared object, again with a file type of '.lil'.

UNIX On UNIX systems, a shared object, again with a file type of '.lil'.

z/OS On 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 79.

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 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 platform 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.

   Windows 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

Before you start

1. You must have created and compiled your user-defined node in Java or in C.
2. You must have created the representation of your user-defined node in the workbench.
3. You must have tested 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 it is ready to be used 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 34 or “Creating a user-defined extension in Java” on page 60.
- 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 79.

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:

1. Switch to the Plug-in Development perspective.
2. Right-click the node project that you want to package for distribution.
3. Click **File** → **Export**.
4. From the list displayed, select **Zip file**.
5. Click **Next**.
6. The resources that are available for you to export as a compressed file are listed. Select your user-defined node by selecting the check box next to its project name. Resources that are automatically selected for the node include the .msgnode file, the .properties file, plugin.xml, and palette.properties.
7. Deselect the following files and directories (all are selected as default):
   - .classpath
   - .project
   - build.properties
   - build.xml
   - /bin
   - /src

   - /temp.folder
8. Specify a name and location for your compressed file, specifying the same name as that of your user-defined node project.
9. Click **Finish**.

The compressed file is saved at the location that you specified. Java source code that you developed within the project is included in the compressed file. You can add your C source code or compiled files to the compressed file using any file compression utility. You then have a self-contained package that you can distribute.

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 87.

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

1. You must have created and compiled your user-defined node, in Java or in C.
2. You must have created the workbench representation of your user-defined node.
3. You must have tested your user-defined node.
4. You must have packaged the user-defined node workbench project.

You must install user-defined extensions on all appropriate broker computers, and, if the extension is a user-defined node, on the toolkit computers (user-defined parsers have no toolkit component). Components can be installed separately, or as part of one process. The components can be on different systems, so you should ensure that the installations are completed on all affected systems.

Toolkit installation

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

Windows To determine the toolkit version, see "Detecting Installed versions of WebSphere Message Broker" on page 93.

If the product is shell-sharing with another product, the installation path is determined by the first Rational program that was installed. To determine the location of the toolkit installation, look at the cdi_ref.properties file which is set up by Rational. This file is installed by the Message Brokers Toolkit and by Rational Version 6.x products; the first time one of these products is installed the file is created. It is shared by all subsequent installations. If all Message Brokers Toolkit and Rational Version 6.x products are uninstalled, the file is removed when the last product is uninstalled.

1. If you are using Installshield Multiplatform Edition to determine the location, the location resolves to $D(os_main)/IBM/RAT60/.sdpinst/cdi_ref.properties

   In this instance, $D(os_main) is an Installshield variable, which the CDI install framework used.
2. If you are not using Installshield Multiplatform Edition:
   a. The location resolves to /etc/IBM/RAT60/.sdpinst/cdi_ref.properties
   b. Look for the file in two locations, in the following order:
      1) %windir%/IBM/RAT60/.sdpinst/cdi_ref.properties
      2) %SystemRoot%/IBM/RAT60/.sdpinst/cdi_ref.properties
3. Use the cdi_ref.properties file to detect the presence of a Message Brokers Toolkit installation; search for `c_wmbt_specific.products=wmbt`

   For the location, look at the following variable: `c_wmbt_specifc.b_wmbt_specific.location`.

   Windows For example, C:\Program Files\IBM\MessageBrokersToolkit\6.0

   Linux For example, /opt/ibm/MessageBrokerToolkit/6.0

   For the location of the workbench look at variable `c_wb.b_wb.location`.

   Windows For example, C:\Program Files\IBM\MessageBrokersToolkit\6.0

   Linux For example, /opt/ibm/MessageBrokerToolkit/6.0

The value of `c_wb.b_wb.location` might not be the same as the value of `c_wmbt_specifc.b_wmbt_specific.location`. This discrepancy can occur if another Rational product has been installed before WebSphere Message Broker.

If cdi_ref.properties does not exist, no Rational products are installed, and the Message Brokers Toolkit is not installed.

To set up Message Brokers Toolkit with icons and options for a new user-defined node, set up a new Eclipse link file that points to the directory containing the Eclipse plug-in files. The link file must contain one line which specifies the full path of the target directory. When you create the Eclipse link file, place it in <c.wb.b_wb.location>/eclipse/links/.

Copy the compressed file that you created in “Packaging a user-defined node workbench project” on page 89 to the directory identified by your link file. Extract the contents into that directory. For example:

Linux The file named opt/ibm/MessageBrokerToolkit/6.0/eclipse/links/ Myextension.link might contain the line `path=/opt/My/Extension/Nodes/eclipse/plugins/`*`your_node_name`*. The directory pointed to by the path variable must contain the contents of the compressed file that you created earlier when you packaged the project.

When you have installed the extension, restart the target workbench with the `-clean` option. You can do this from the command line, or by modifying the menu shortcut. You should use the `-clean` option whenever any changes are made to user-defined extensions, to make sure the changes are picked up by the message flow node palette. When the workbench has restarted, the new category of nodes appears on the palette of the flow editor.

If you are an experienced Eclipse user or plug-in developer, you might want to use more advanced Eclipse functions to handle additional products like user-defined extensions. For example, you can package user-defined extensions as Eclipse features, instead of plug-ins.

Features have several advantages:

- You can include many related plug-ins in a single feature.
- You can define a feature such that it is restricted to use with particular versions of your workbench.
- You can provide automated updates to features using the Eclipse Update Manager.

For a full description of these and other advanced Eclipse options, see the PDE Guide which includes a section about creating features. You might also find useful the description of the feature manifest in ″Navigating and customizing the workbench″.

Runtime installation

You might need to detect the version of the runtime that is 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 93.

To add .jar or .lil files to runtime installations on WebSphere Business Integration Message Broker Version 5.0 or later, see “Installing user-defined extension runtime files on a broker” on page 87. For Version 2 brokers, add the plug-in files to *install_dir*/lil/ and restart the broker.

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 below.

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. 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 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.

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

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

Detecting installed versions on Windows:

You can use the following instructions in your installer scripts to test for the following versions of WebSphere Message Broker. To detect each version, look for the registry key given for each version.

MQSeries Integrator Version 2

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
WebSphere MQ Integrator V2.1
```

WebSphere Business Integration Message Broker Version 5.0 toolkit

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
mqsi.studio\DisplayVersion = 5.x.x.x
```

In this example, x can be any integer.

WebSphere Business Integration Message Broker Version 5.0

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
mqsi.ib\DisplayVersion = 5.x.x.x
```

In this example, x can be any integer.

WebSphere Message Broker Version 6.0 toolkit

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
WMBT60\DisplayVersion = 6.x.x.x
```

In this example, x can be any integer.

WebSphere Message Broker Version 6.0

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
mqsi60\DisplayVersion = 6.x.x.x
```

In this example, x can be any integer.

Detecting installed versions on Linux and UNIX systems:

UNIX platforms do not have a common packaging method: you must check which files are present in the filesystem. Look for the files listed below for each version of WebSphere Message Broker that you want to detect.

MQSeries Integrator Version 2 runtime components

AIX You should check for the presence of /usr/opt/mqsi/bin/mqsilist. You should also check that /usr/opt/mqsi/bin/mqsiprofile and /usr/opt/mqsi/bin/mqsisetdbparms are not present.

On other UNIX systems:

You should check for the presence of /opt/mqsi/bin/mqsilist and make sure that opt/mqsi/bin/mqsiprofile and /opt/mqsi/bin/mqsisetdbparms are not present.

WebSphere Business Integration Message Broker Version 5.0 runtime components

AIX You should check for the presence of /usr/opt/mqsi/bin/mqsilist. You should also check that /usr/opt/mqsi/bin/mqsiprofile is not present.

On Linux and other UNIX systems:

You should 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 runtime components

Version 6 and later runtime components can be detected by looking for /var/mqsi/install.properties. Each line in this file contains a install path and V.R.M.F version information.

WebSphere Message Broker Version 6.0 toolkit

Version 6 and later toolkits can be detected by looking for the existence of /etc/IBM/WebSphereMessageBrokersToolkit/products/com.ibm.wbmt.

To determine the version, you can 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
```

Changing a user-defined extension

On all systems, you can change a user-defined extension file by completing the following steps:

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

There are two situations where it is not necessary to stop and start the broker:

- If have created an execution group in the Toolkit, and there is nothing yet deployed to it, you can add the .lil, .pdb, and .jar file to your chosen directory.
- If something has already been deployed to the execution group you that want to use, add the lil/pdb/jar file to your chosen directory and then use the mqsireload command to restart the group. It is not possible to overwrite an existing file on the Windows platform 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 will also detect the new .lil, .pdb, and .jar files when that execution group is restarted, or when something is first deployed to it. By using the more conventional way of restarting 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

On all types of system, you can remove a user-defined extension file from the broker as follows:

1. Stop the broker, using the mqsistop command.
2. Delete the .lil or .jar file from the appropriate directory. This is one of the following:

   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

Message processing nodes and parsers are unlikely to need to write directly to the local error log, because it is recommended that a user-defined extension reports errors using exceptions. However, you can choose to write significant events, error or otherwise, for problem determination and operational purposes in the same manner as WebSphere Message Broker.

With C code, you use the utility function **CciLog** to do this. Two of the arguments accepted by this function, `messageSource` and `messageNumber`, define the event source and the actual integer representation of a message within that source, respectively.

For Java code, the class `MbService` provides static methods to log information to the event log. To log messages to the event log, you need to 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 below:

- 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*

For Windows systems, the messages are written to the Windows event log, and your message catalog must be delivered as a Windows DLL.

For Linux and UNIX systems, these messages are written to the SYSLOG facility, and your message catalog must be delivered as an XPG4 message catalog.

The above covers exceptions raised during normal processing. You must also provide for exceptions raised when deploying and configuring a message flow. Messages resulting from these configuration exceptions are reported back to the workbench for display to the workbench user. To facilitate this, you must create an appropriately named Java properties file and copy it to each workbench.

Building and installing a Windows event source

Windows On Windows, the message catalog is delivered as a Windows DLL, which you must create as described below. This contains definitions of your event messages to enable the event viewer to display a readable format, based on the event message written by your application. When you compile a message catalog, a header file is created, which defines symbolic values for each event message number you have created. This header file is included by 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 website, `http://msdn.microsoft.com`, 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 path and directory for the output files and `mymsg.mc` is the name of the input file.

   The message compiler produces an output header (.h) file which contains symbolic #defines that map to each message number coded in the input.mc file. This header file must be included when compiling a user-defined extension source file that uses the **CciLog** utility function to write an event message 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 <filename>.rc
   ```
4. Create a resource DLL using the .res file by issuing the command:
   ```
   LINK /DLL /NOENTRY <filename>.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 defined in:

```
HKEY_LOCAL_MACHINE
   SYSTEM
      CurrentControlSet
         Services
            EventLog
               Application
```

Right-click on *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.

You must create the following values for this entry:

- The `EventMessageFile` String value must contain the fully qualified path for the `.dll` you have created to contain your messages. This is the message catalog used by **CciLog**.
- The `TypesSupported` DWORD value must contain the value ″7″.

Part 2. Reference

User-defined extensions

The following information is contained within this section:

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.

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/Javaplugin/com/ibm/samples directory.

UNIX On UNIX, the following files are in the *install_dir*/sample/Javaplugin/com/ibm/samples directory.

z/OS On z/OS, the following files are in the *install_dir*/sample/Javaplugin/com/ibm/samples directory.

JavaSwitchPluginNode.java	Java source file containing a sample implementation of a message processing node that routes a message to one of five output terminals, depending on the content.
JavaTransformPluginNode.java	Java source file containing a sample implementation of a simple fixed transformation of an input message into an output message.

The files that the workbench needs to recognize the Switch node and Transform node are in the *install_dir*\sample\extensions\nodes\com.ibm.samples.nodes directory. You can add this directory to your workspace using the Update Manager, or you can copy it across to your workspace directory and restart the workbench to see the nodes. The help files (HelpContexts.xml, SwitchNode.htm and TransformNode.htm) demonstrate some features of Eclipse help by adding themselves into the main topic tree, referencing topics in the main tree, and so on.

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/.

Header files

The C interfaces are defined by the following header files:

- **BipCni.h**: this header file 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 104.
- **BipCpi.h**: this header file 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 175.
- **BipCci.h**: this header file 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 247. This file also contains definitions for utility function return codes and values. See “Utility function return codes and values” on page 278 for more information.
- **BipCos.h**: this header file 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 105

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 122

Optional and conditional functions
- "cniDeleteNodeContext" on page 126
- Either "cniEvaluate" on page 139 (for message processing and output nodes), or "cniRun" on page 153 (for input nodes)
- "cniGetAttribute" on page 141
- "cniGetAttribute2" on page 142
- "cniGetAttributeName" on page 143
- "cniGetAttributeName2" on page 144
- "cniSetAttribute" on page 158

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

Parser	Root element name	Class name
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 123
- "cniDefineNodeClass" on page 124
- "cniDispatchThread" on page 127 (for input nodes only)
- "cniCreateInputTerminal" on page 120
- "cniCreateOutputTerminal" on page 124
- "cniIsTerminalAttached" on page 148
- "cniGetBrokerInfo" on page 145

Message management

- "cniCreateMessage" on page 121
- "cniDeleteMessage" on page 126
- "cniFinalize" on page 140

- “cniGetMessageContext” on page 146
- “cniGetEnvironmentMessage” on page 146
- “cniPropagate” on page 151

Message buffer access

- “cniBufferByte” on page 109
- “cniBufferPointer” on page 110
- “cniBufferSize” on page 110
- “cniSetInputBuffer” on page 163 (for input nodes only)
- “cniWriteBuffer” on page 174

Syntax element navigation

- “cniRootElement” on page 152
- “cniParent” on page 150
- “cniNextSibling” on page 150
- “cniPreviousSibling” on page 151
- “cniFirstChild” on page 141
- “cniLastChild” on page 149
- “cniSearchElement group” on page 154
- “cniSearchElementInNamespace group” on page 156
- “cniSqlCreateReadOnlyPathExpression” on page 167
- “cniSqlCreateModifyablePathExpression” on page 169
- “cniSqlNavigatePath” on page 172
- “cniSqlDeletePathExpression” on page 174

Syntax element access

- “cniAddAfter” on page 107
- “cniAddBefore” on page 109
- “cniAddasFirstChild” on page 108
- “cniAddasLastChild” on page 108
- “cniCopyElementTree” on page 111
- “cniCreateElementAfter” on page 112
- “cniCreateElementAfterUsingParser” on page 112
- “cniCreateElementBefore” on page 119
- “cniCreateElementBeforeUsingParser” on page 119
- “cniCreateElementAsFirstChild” on page 113
- “cniCreateElementAsFirstChildUsingParser” on page 114
- “cniCreateElementAsLastChild” on page 115
- “cniCreateElementAsLastChildFromBitstream” on page 116
- “cniCreateElementAsLastChildUsingParser” on page 118
- “cniDetach” on page 127
- “cniElementAsBitstream” on page 128
- “cniElementName” on page 133
- “cniElementNamespace” on page 134
- “cniElementType” on page 135
- “cniElementValue group” on page 136
- “cniElementValueState” on page 137

- “cniElementValueType” on page 138
- “cniElementValueValue” on page 138
- “cniGetParserClassName” on page 147
- “cniSetElementName” on page 159
- “cniSetElementNamespace” on page 160
- “cniSetElementType” on page 161
- “cniSetElementValue group” on page 161
- “cniSetElementValueValue” on page 163

SQL statement handling
- “cniSqlCreateStatement” on page 164
- “cniSqlExecute” on page 166
- “cniSqlSelect” on page 167
- “cniSqlDeleteStatement” on page 165

Miscellaneous
- “cniGetThreadContext” on page 148

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 104 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.

This function must be called 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 address is returned from cniCreateNodeContext.

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.

This function must be called 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 address is returned from cniCreateNodeContext.

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 options 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 memory allocated by the user. 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 being used, and the options specified. All parsers support the following modes:
- RootBitStream, in which the bitstream generation algorithm 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 bitstream generation algorithm the same as that used by an output node, but also the following are determined, if not explicitly specified, in the same way as the output node, which means they are determined by searching the previous siblings of *element* on the assumption that these 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). 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

`element`
The syntax element to be serialized (input.)

The syntax element to be serialized (input.) Must be last child of the message root.

`value`
A pointer to a CciByteArray struct containing 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). A NULL pointer means that this parameter is ignored. Also, if the parser associated with the element 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 bit stream from the element tree (input). A NULL pointer means that this parameter is ignored. Also, if the

parser associated with the element has no interest in this value, for example, if it is a generic XML parser, the parameter is also ignored.

messageFormat
The format used to create the bit stream from the element tree (input). A NULL pointer means that this parameter is ignored. Also, if the parser associated with the element 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 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. You can specify a value of 0 to indicate that the queue manager's ccsid should be 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 element and its children. Currently no parsers support this option.

options
Integer value which specifies which bitstream generation mode should be used. It can take 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 set to CCI_BUFFER_TOO_SMALL.
- If an exception occurs during execution, *returnCode* is set to CCI_EXCEPTION.

Example

The following example demonstrates how the options parameter should 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
RFH2
<test><data><stuff>things</stuff></data></test>
```

the node will propagate 3 messages, one containing a copy of the input message in the BLOB domain. One containing a copy of the input RFH2 as the message body in the BLOB domain. One containing 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 RFH2*/
/*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 RFH2 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, destinationList, exceptionList, outMsg[0]);
    retvalue = cniPropagate(&rc, terminalObject, destinationList, exceptionList, outMsg[1]);
    retvalue = cniPropagate(&rc, terminalObject, destinationList, 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. It is invoked by the message broker when a message is received on one of the input terminals of an instance of a node object. This function forms the main processing logic of the message flow or output node. It is not used with input nodes.

You need to have defined a function table before calling this function, or it will not work.

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  *destinationList,
  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).

destinationList
The address of the input destination list object (input).

exceptionList
The address of the exception list for the message (input).

message
The address of the input message object (input).

cniFinalize

Causes the broker to request parsers that support the finalize feature to perform their finalize processing on the specified message. The behavior of this processing is specific to each parser.

If the *options* parameter is set to CCI_FINALIZE_VALIDATE, a parser should also perform validation processing to ensure that the element tree owned by it is of the correct structure. This helps prevent messages with incorrectly formed element trees being propagated to other nodes in the message flow.

It is recommended that cniFinalize is called before propagating a message (for example, before calling 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
Specifies bit flags to identify the finalize or validate options to be used (input). This parameter is optional. You can set it to CCI_FINALIZE_VALIDATE.

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,
                       destinationList,
                       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

Restriction: This function imposes a restriction on the length of the attribute value. This function is provided for backward compatibility only. You should implement cniGetAttribute2.

This function gets the value of an attribute on a specific node instance. It is invoked by the message broker as follows:

- Before the nodes configuration is deployed in order to ascertain default values of any attributes that may override attributes owned by the framework.

- After setting the deployed configuration in order to write the configuration to the Broker’s database. This ensures that the configuration persists across shutdown and restarts of the execution group

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	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

Restriction: This function imposes a restriction on the length of the attribute value. This function is provided for backward compatibility only. You should implement cniGetAttributeName2.

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 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).

Return values

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

Example

```
  cniGetBrokerInfo(0, ((NODE_CONTEXT_ST *)context)->nodeObject, &broker_info_st);
```

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. You can use it to test whether a message can be propagated to a terminal. However, it is not necessary to call this function before propagating a message with the cniPropagate utility function. Using the cniIsTerminalAttached function, a node can modify its 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 occurred, 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,
                                destinationList,
                                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 regarded as a no-op. Therefore, it is not necessary to check whether the terminal is attached before propagating 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*   destinationList,
  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.

destinationList
The address of the destination list object to be sent with the message (input).

This message object is used by the publish/subscribe node supplied by the message broker.

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. It is not used by message processing or output nodes, and you do not need to call `cniEvaluate`. WebSphere Message 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*  destinationList,
  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).

`destinationList`
The address of the input destination list object (input).

`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 associate a bit stream with this message through calling `cniSetInputBuffer`. Populating the tree of this message is not supported, therefore calls to functions such as `cniAddAsLastChild` or `cniCreateElementAsLastChildFromBitstream` will be ineffective. To build parts of the tree instead of providing a buffer to be parsed as the whole message, you should create a new message using `cniCreateMessage`.

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 then you should complete 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 shown below.

`CCI_TIMEOUT`
The input node did not receive its input data and it requires that control be returned to WebSphere Message Broker in case message flow reconfiguration is being requested. A user-defined input node should return reasonably frequently to give control back to WebSphere Message Broker.

`CCI_SUCCESS_CONTINUE`
A message was successfully processed. Default transaction commit processing is performed by WebSphere Message Broker. 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. Default transaction commit processing is performed by WebSphere Message Broker. The input node has determined that the thread is not required and it is returned to the message flow thread pool. If this is performed on the only thread, or the last active thread, WebSphere Message Broker prevents this last thread being returned to the pool, otherwise there would be no other active threads that can dispatch another thread. In this situation, WebSphere Message 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 is performed on the last active thread, WebSphere Message Broker prevents this last thread being returned to the pool, otherwise there would be no other active threads that can dispatch another thread. In this situation WebSphere Message 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);
```

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.

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 previously created using the cniSqlCreateStatement utility function, as defined by the *sqlExpression* parameter. This function is used when the statement does not return data, for example, when a PASSTHRU function is used.

Syntax

```
void cniSqlExecute(
  int*               returnCode,
  CciSqlExpression*  sqlExpression,
  CciMessage*        destinationList,
  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).

`destinationList`
: The message representing the input destination list (input).

`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.

cniSqlSelect

Executes an SQL statement 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*        destinationList,
  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).

destinationList
The message representing the input destination list (input).

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.

cniSqlCreateReadOnlyPathExpression

Creates a non modifiable SqlPathExpression object that represents the path specified by the path argument. Non modifiable means that the navigated path will 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.

There is an overhead involved in creating the expression so if the same path expression is to be used for every message then this function should be called once and the CciSqlPathExpression* that is returned should used in a call to niSqlNavigatec for each message. It is possible to use the CciSqlPathExpression* on a different thread than 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 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 nodeObject parameter was passed in, then returnCode will be set to CCI_INV_NODE_OBJECT. If an invalid path parameter, such as a NULL or empty string, was passed in then returnCode is set to CCI_INV_ESQL_PATH_EXPR.

nodeObject (input)
Specifies the message flow processing node the ESQL Path Expression will be 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. NULL is allowed.

path (input)
Pointer to a NULL terminated string of CciChars. This 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, ESQL Namespace constants, because they cannot be declared. However, if it can be done in a one line ESQL path in a compute node, then it should be possible with this API. 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. Once the SQLPathExpression is no longer needed, (typically when the node is deleted) it should be deleted by calling cniSqlDeletePathExpression.

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,
                           destinationList,
                           exceptionList,
                           NULL, /* do not reference any output trees*/
                           NULL,
                           NULL);
```

Using this approach as opposed to using the cniFirstChild, cniNextSibling etc has the following advantages:

- The path is more dynamic – the path string could be determined at deploy time e.g. based on a node attribute (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.

cniSqlCreateModifyablePathExpression

Creates a modifiable SqlPathExpression object that represents the path specified by the path argument. Modifiable means that when navigated, path elements will be 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. There is an overhead involved in creating the expression so if the same path expression is to be used for every message then this function should be called once and the CciSqlPathExpression* that is returned should used in a call to cniSqlNavigate for each message. It is possible to use the CciSqlPathExpression on a different thread than 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 want to deal with errors. Any exceptions thrown during the execution of this call are re-thrown to the next upstream node in the flow. If input is not NULL, output 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 nodeObject parameter was passed in, then returnCode will be set to CCI_INV_NODE_OBJECT. If an invalid path parameter, such as NULL or an empty string, was passed in then returnCode will be set to CCI_INV_ESQL_PATH_EXPR.

`nodeObject` (input)
Specifies the message flow processing node the ESQL Path Expression will be 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 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, 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. Once the SQLPathExpression is no longer needed, (typically when the node is deleted) it should be deleted by calling cniSqlDeletePathExpression.

Example

By adding 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.XML.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,
                     destinationList,
                     exceptionList,
                     outMsg,
                     NULL,/* do not reference OutputLocalEnvironment*/
                     NULL/* do not reference OutputLExceptionList*/);
}
```

So 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>
```

Using this approach as opposed to using cniCreateElementAsLastChild etc has the following advantages:

- The path is more dynamic – the path string could be determined at deploy time e.g. based on a node attribute (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.

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 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. If an invalid CciMessage* value is passed in then returnCode will be set to CCI_INV_MESSAGE_OBJECT. If the element could not be navigated to or created, then 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 accessed.

Correlation name	Data accessed
Environment	The single Environment tree for the flow. This is determined by the broker and it is not necessary to specify it via this API.
InputLocalEnvironment	inputLocalEnvironment parameter to cniSqlNavigatePath
OutputLocalEnvironment	outputLocalEnvironment parameter to cniSqlNavigatePath
InputRoot	inputMessageRoot parameter to cniSqlNavigatePath
InputBody	Last child of InputRoot
InputProperties	InputRoot.Properties **Note:** 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
OutputDestinationList	Synonym for OutputLocalEnvironment

All other rules regarding the actual navigability and validity of paths are as per the ESQL Field Reference documentation.

Return values

If the path is successfully navigated the address of the syntax element is returned. However if the path is not navigable then a value of zero (CCI_NULL_ADDR) is returned and the returnCode parameter indicates the reason for the error.

Example

Assuming 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,
                          destinationList,
                          exceptionList,
                          NULL, /* do not reference any output trees*/
                          NULL,
                          NULL);
```

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);
```

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 176.

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 186
- "cpiParseNextSibling" on page 211
- "cpiParsePreviousSibling" on page 212
- "cpiParseFirstChild" on page 209
- "cpiParseLastChild" on page 210

Optional and conditional functions
- "cpiDeleteContext" on page 190
- "cpiElementValue" on page 195
- "cpiNextParserClassName" on page 199
- "cpiNextParserCodedCharSetId" on page 201
- "cpiNextParserEncoding" on page 202
- "cpiParseBuffer" on page 205
- "cpiParseBufferEncoded" on page 206
- "cpiParseBufferFormatted" on page 207
- "cpiParserType" on page 213
- "cpiSetElementValue" on page 221
- "cpiSetNextParserClassName" on page 226
- "cpiWriteBuffer" on page 227
- "cpiWriteBufferEncoded" on page 228
- "cpiWriteBufferFormatted" on page 229

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:

Initialization and resource creation
- "cpiCreateParserFactory" on page 187
- "cpiDefineParserClass" on page 189

Message buffer access
- "cpiAppendToBuffer" on page 181
- "cpiBufferByte" on page 182
- "cpiBufferPointer" on page 183
- "cpiBufferSize" on page 183

Syntax element navigation
- "cpiRootElement" on page 214
- "cpiParent" on page 203
- "cpiNextSibling" on page 202
- "cpiFirstChild" on page 198
- "cpiLastChild" on page 199
- "cpiAddAfter" on page 177

Syntax element access
- "cpiAddBefore" on page 180
- "cpiAddAsFirstChild" on page 178
- "cpiAddAsLastChild" on page 179
- "cpiCreateAndInitializeElement" on page 184
- "cpiCreateElement" on page 186
- "cpiElementCompleteNext" on page 191
- "cpiElementCompletePrevious" on page 192

- “cpiElementName” on page 192
- “cpiElementNameSpace” on page 193
- “cpiElementType” on page 194
- “cpiElementValue group” on page 196
- “cpiElementValueValue” on page 197
- “cpiSetCharacterValueFromBuffer” on page 215
- “cpiSetElementCompleteNext” on page 216
- “cpiSetElementCompletePrevious” on page 217
- “cpiSetElementName” on page 218
- “cpiSetElementType” on page 220
- “cpiSetElementValue group” on page 222
- “cpiSetElementValueValue” on page 224
- “cpiSetNameFromBuffer” on page 225

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 237.

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 232

Optional functions
- “cciPropagatedMessageCallback” on page 235
- “cciNodeCompletionCallback” on page 234
- “cciInputMessageCallback” on page 232
- “cciTransactionEventCallback” on page 236

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

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 247.

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 250
- “cciGetLastExceptionDataW” on page 251
- “cciLog” on page 253
- “cciLogW” on page 254
- “cciRethrowLastException” on page 258
- “cciThrowException” on page 263
- “cciThrowExceptionW” on page 264

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 255
- “cciUcsToMbs” on page 267

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 266
- “cciUserTrace” on page 272
- “cciUserTraceW” on page 273
- “cciUserDebugTrace” on page 268
- “cciUserDebugTraceW” on page 270
- “cciServiceTrace” on page 261
- “cciServiceTraceW” on page 262

- “cciServiceDebugTrace” on page 259
- “cciServiceDebugTraceW” on page 260

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 256

The following utility functions are available for use with user exits:

- “cciGetBrokerInfo”
- “cciGetNodeAttribute” on page 237
- “cciGetNodeName” on page 238
- “cciGetNodeType” on page 239
- “cciGetSourceNode” on page 240
- “cciGetSourceTerminalName” on page 241
- “cciGetTargetNode” on page 242
- “cciGetTargetTerminalName” on page 242
- “cciInputMessageCallback” on page 232
- “cciNodeCompletionCallback” on page 234
- “cciPropagatedMessageCallback” on page 235
- “cciRegisterUserExit” on page 243
- “cciTransactionEventCallback” on page 236

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** 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.

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

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("BIPv600", 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

Throws an exception. The 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 BIPV600 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 = "BIPv600";

  /* 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 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 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;

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 is a skeleton code for a C user-defined node. It has the minimum content 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* destinationList,
  CciMessage* exceptionList,
  CciMessage* message
){
  /* we would place our node's processing logic in here*/
  return;
}

int run(
  CciContext* context,
  CciMessage* destinationList,
  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,destinationList,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 is a makefile which lists the files, dependencies, and rules by which the C user-defined node should be compiled.

```
.SUFFIXES : .so .a .o .c

R1INC  = .
R1LIB  = .

# WMQI
MQSIDIR  = /cmvc/back/inst.images/x86_linux_2/shipdata/opt/mqsi
MQSIINC  = $(MQSIDIR)/include
MQSILIB  = $(MQSIDIR)/lib

# WMQ
MQIDIR  = /usr/mqm

CC   = /usr/bin/g++
LD   = ${CC}


OBJ = .o
LIL = .lil
THINGSTOCLEAN = *${OBJ}
CFLAGS  = -fpic -c #-pedantic -x c -Wall
CFLAGSADD = -I${R1INC} -I${MQSIINC} -I${MQSIINC}/plugin ${DEFINES}
DEFINES  = -DLINUX

LIBADD  = -L${MQSILIB} -limbdfplg
LDFLAG  = -shared ${LIBADD}

#CC   = /usr/bin/gcc
#LD   = ${CC}

OBJECTS = skeleton${OBJ}
.c.o : ; ${CC} ${CFLAGS} ${CFLAGSADD} $<
```

```
ALL : ${OBJECTS} Samples${LIL}
clean:
 rm *${OBJ} *${LIL}


skeleton${OBJ}: skeleton.c


Samples${LIL}: ${OBJECTS}
 ${LD} -o $@ ${OBJECTS} ${LDFLAG}
```

Utility function return codes and values

By convention, the return code output parameter of all utility functions is set to indicate successful completion, or 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.
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.

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

Return code	Explanation
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.
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.

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

Return code	Explanation
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 invoked by the broker only when that parser is required. The parser that is invoked 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 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

Parser	Root element name	Class name
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

When using the MQMD parser, the MQMD is assumed to be a V2 MQMD.

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
IntSubset	0x05000021
ExtSubset	0x05000022

Table 2. XML parser names and values (continued)

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
mtiSdfCharsCodepage	0x02000008
mtiSdfCharsEcho	0x02000010

Table 3. MRM parser names and values (continued)

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

Part 3. Appendixes

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this 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.

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

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

AIX	CICS	Cloudscape
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

Printed in USA