

MQSeries® Integrator



Programming Guide

Version 2.0

MQSeries® Integrator



Programming Guide

Version 2.0

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix C, "Notices" on page 189.

First edition (April 2000)

This edition applies to IBM® MQSeries® Integrator Version 2.0, and to any subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book	vii
Who this book is for	vii
What you need to know to understand this book	vii
Terms used in this book	vii
Where to find more information	viii
MQSeries Integrator publications	viii
MQSeries publications	viii
MQSeries information available on the Internet	ix

Part 1. Application programming 1

Chapter 1. Introduction	3
Overview of MQSeries Integrator	3
Chapter 2. Writing application programs	7
Sending and receiving messages	7
Point-to-point messaging	9
Chapter 3. Writing publish/subscribe applications	11
Publish/subscribe messaging	12
The publisher	13
The subscriber	16
Flow of publish/subscribe messages	20
The broker	22
Sample application	25
Using the AMI in publish/subscribe applications	27
Chapter 4. The MQRFH2 rules and formatting header	29
MQRFH2 Structure	29
Message service folders	36
Chapter 5. Publish/subscribe command messages	39
Delete Publication	40
Deregister Subscriber	42
Publish	45
Register Subscriber	48
Request Update	51
Broker Response	53
Message descriptor	55
Reason codes	58

Part 2. Programming a plug-in node or parser 61

Chapter 6. Implementing a plug-in node or parser	63
Introduction	64
Implementing a message processing node	66
Implementing a message parser	87
Accessing the message content	89
General development considerations	95

Contents

Using Windows NT event logging from a plug-in	97
Errors and exception handling	98
Chapter 7. Node implementation and utility functions	103
Node implementation function overview	104
Node utility function overview	105
Exception handling and logging overview	107
Node implementation function interface	108
Node utility function interface	112
Chapter 8. Parser implementation and utility functions	135
Parser implementation function overview	136
Parser utility function overview	137
Exception handling and logging overview	139
Parser implementation function interface	140
Parser utility function interface	146
Chapter 9. Exception handling and logging	159
<hr/>	
Part 3. Appendixes	163
Appendix A. Using filters in content-based routing	165
Field references	165
Appendix B. Plug-in node sample code	171
The sample property editor TraceSettingPropertyEditor.java	171
The sample customizer files	173
Appendix C. Notices	189
Trademarks	191
<hr/>	
Part 4. Glossary and index	193
Glossary of terms and abbreviations	195
Index	199

Figures

	1. Communication between publisher, subscriber, and broker	12
	2. Publish/subscribe without retained publications	21
	3. Using retained publications	21
	4. Publish on request only	22
	5. The results service application	25
+	6. A sample plug-in XML interface definition file	70
+	7. Sample plug-in WDP file	76
+	8. Plug-in node icons	78
+	9. Sample plug-in node help file	79
+	10. Sample plug-in node properties file	81
+	11. PropertyEditor sample code	82
+	12. Customizer sample code	84
	13. Tree representation of a message	90
	14. Syntax element tree	91
	15. First generation of syntax elements	91
	16. Tree representation of an XML message	93

Tables

1. Fields in MQRFH2	29
2. Initial values of fields in MQRFH2	34
3. Utility function return codes and values	100

Tables

About this book

This book explains how to write application programs that communicate with MQSeries Integrator Version 2.0, or to write plug-in nodes and parsers that can be installed in this product.

Part 1, “Application programming” on page 1 starts with a brief overview of the concepts and capabilities of MQSeries Integrator Version 2.0. It then describes how to write application programs using both *point-to-point* and *publish/subscribe* communication models. Full details of the publish/subscribe command messages, and the MQRFH2 message header that is used to send them, are also provided.

Part 2, “Programming a plug-in node or parser” on page 61 describes how to write *plug-in* message processing nodes and parsers to enhance the capabilities of MQSeries Integrator. Full details of the functions you need to write, and the utility functions provided to assist you, are given in this part of the book.

A glossary is also provided.

+ Changes made to this book after it was included on the MQSeries Integrator
+ Version 2.0 CD are marked with the + character.

Who this book is for

This book is for programmers who need to write application programs that will communicate with MQSeries Integrator Version 2.0, or to write plug-in nodes and parsers that will be installed in this product.

What you need to know to understand this book

To understand this book, you need to have some understanding of MQSeries, including the use of the Message Queue Interface (or the Application Messaging Interface).

You are recommended to read the *MQSeries Integrator Introduction and Planning* book before starting to write application programs. It contains information about the design of applications that communicate with MQSeries Integrator.

Terms used in this book

All references to MQSeries Integrator are to MQSeries Integrator Version 2.0 unless otherwise stated.

The book uses the following shortened names:

- MQSeries: a general term for IBM MQSeries Messaging products.
- MQSeries Publish/Subscribe: the MQSeries Publish/Subscribe SupportPac™ available on the Internet for several MQSeries server operating systems (the Internet URL is given in “MQSeries information available on the Internet” on page ix).

Where to find more information

Becoming familiar with the MQSeries Integrator library will help you accomplish MQSeries Integrator tasks quickly. The library covers planning, installation, administration, and client application tasks.

The library also contains references to complementary product libraries, including the MQSeries Family library.

MQSeries Integrator publications

The following books make up the MQSeries Integrator Version 2.0 library:

- IBM MQSeries Integrator Version 2.0 Introduction and Planning, GC34-5599
- IBM MQSeries Integrator Version 2.0 Installation Guide, GC34-5600
- IBM MQSeries Integrator Version 2.0 Messages, GC34-5601
- IBM MQSeries Integrator Version 2.0 Using the Control Center, SC34-5602
- IBM MQSeries Integrator Version 2.0 Programming Guide, SC34-5603 (this book)
- IBM MQSeries Integrator Version 2.0 Administration Guide, SC34-5792

All books in the MQSeries Integrator Version 2.0 library are available in softcopy, in Portable Document Format (PDF).

You can read these books using the Adobe Acrobat Reader, or in a Web browser (with Acrobat Reader as a plug-in). You can also print your own copies of these books.

You need Version 3 or later of Acrobat Reader. You can download a free copy of it from the Adobe Web site:

<http://www.adobe.com>

The MQSeries Integrator Version 2.0 books are also available in hardcopy.

MQSeries publications

This section describes MQSeries publications that are referred to in this book. They are available in hardcopy, HTML, and PDF formats, except where noted.

MQSeries Application Programming Guide

The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference

The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Messaging Interface

The *MQSeries Application Messaging Interface* book, SC34-5604, describes the MQSeries Application Messaging Interface SupportPac. This is a simple interface

that application programmers can use without needing to understand all the options available in the MQI. The options that are required in a particular installation are defined by a system administrator, using services and policies.

This book is available in PDF format only.

MQSeries Publish/Subscribe User's Guide

The *MQSeries Publish/Subscribe User's Guide*, GC34-5269, provides comprehensive information for users of the MQSeries Publish/Subscribe SupportPac. It includes: installation; system design; writing applications; and managing the publish/subscribe broker.

This book is available in PDF format only.

For a complete list of MQSeries product publications, refer to the information on the MQSeries web site.

MQSeries information available on the Internet

The MQSeries Business Solution, of which MQSeries Integrator is a part, has a Web site at:

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

By following links from this web site you can:

- Obtain the latest information about all MQSeries family products.
- Access all the books for the MQSeries family products.
- Down-load MQSeries SupportPacs.

You might be interested in the MQSeries Integrator problem determination Q&A SupportPac (MHI1) that you can access from:

<http://www.ibm.com/software/mqseries/txppacs/>

Part 1. Application programming

This part contains:

- Chapter 1, "Introduction" on page 3
- Chapter 2, "Writing application programs" on page 7
- Chapter 3, "Writing publish/subscribe applications" on page 11
- Chapter 4, "The MQRFH2 rules and formatting header" on page 29
- Chapter 5, "Publish/subscribe command messages" on page 39

Chapter 1. Introduction

To build a complete MQSeries Integrator Version 2.0 application, you need to consider the following activities:

- Define the information space and model.
- Build the business message flows.
- Develop or modify applications that feed messages into the message flows, and consume the messages they produce.

The first two activities are described in the *MQSeries Integrator Introduction and Planning* book and *MQSeries Integrator Using the Control Center* respectively.

This book concentrates on the third activity, how to develop applications that work with MQSeries Integrator Version 2.0. To help you understand what can be done with the product, this chapter contains an overview of its functionality.

Overview of MQSeries Integrator

MQSeries Integrator Version 2.0 is IBM's *message broker* product, addressing the needs of business and application integration through management of information flow. It provides services that allow you to:

- Route a message to several destinations, using rules that act on the contents of one or more of the fields in the message or message header.
- Transform a message, so that applications using different formats can exchange messages in their own formats.
- Store and retrieve a message, or part of a message, in a database.
- Modify the contents of a message (for example, by adding data extracted from a database).
- Publish a message to make it available to other applications. Other applications can choose to receive publications that relate to specific topics, or that have specific content, or both.
- Extend the function of MQSeries Integrator Version 1.

These services are based on the messaging transport layer provided by the MQSeries products.

Message brokers

A message broker, usually referred to simply as a broker, is a set of execution environments hosting services you create to handle your message traffic.

You can install and configure any number of brokers. Together, these make up a *broker domain*. The broker domain is the set of brokers you can administer as a single entity, using the MQSeries Integrator Version 2.0 Control Center.

Brokers are connected together to provide communication throughout your broker domain, as required. This is needed for publish/subscribe applications (see "Applications" on page 5). Communications in the broker domain are provided by MQSeries.

Message flows

When you design your broker domain, you decide what processing must be done on the messages flowing through the brokers. You define this work as a set of actions executed between receipt of the message by the broker, and delivery of the message to the target applications.

Each action, or subset of actions, is implemented as a *message processing node*, and these are grouped together in a sequence to form a *message flow*. You create message flows using the MQSeries Integrator Control Center (see “The Control Center” on page 5).

Message flows can range from the very simple, performing just one action on a message, to the complex, providing a number of actions on the message to transform its format and content. A message flow can process one message in several ways to deliver a number of output messages, perhaps with different format and content, to a number of target applications.

The message flows you create receive messages at **MQInput** nodes. Every input node represents an MQSeries queue, and every message flow must have at least one input node.

Message flows usually complete their activity by sending one or more messages to one or more recipients from **MQOutput** nodes that represent MQSeries queues, or from **Publication** nodes that redistribute the message to interested subscribers using MQSeries queues.

Other message flows might simply store the message in a database for later processing, and not use an output node at all.

The other nodes between input and output provide the actions you want taken against the messages. MQSeries Integrator supplies a number of predefined message processing nodes. In addition to the input and output nodes already mentioned, their functions include filter (on message data content) and compute (for example, add data from a database).

You can create new nodes, using a system programming interface supplied by MQSeries Integrator, to provide other options for message processing. This is described in Part 2, “Programming a plug-in node or parser” on page 61.

Messages

Each message flowing through your system has a specific content and structure, referred to as a *message template*.

Message template information identifies the structure of the data it contains. Messages sent to MQSeries Integrator can be of the following types:

- MQSeries messages, with an MQSeries message descriptor (MQMD) and data; this type of message does not have to be predefined but if the data is not one of the defined types, MQSeries Integrator will not be able to distinguish individual fields
- Message repository manager (MRM) messages (defined in the MQSeries Integrator Control Center)

- MQSeries Integrator Version 1 messages (defined in the MQSeries Integrator Version 1 user interface)
- XML (Extensible Markup Language) messages (which are self-defining)
- User-defined messages

Message template information for predefined messages is usually included in the message header, so the message flows recognize the messages when they receive them. Other messages might not use the expected header, but you can set up your message flow input nodes to indicate how the messages will be processed.

The message bit-stream is decoded by *message parsers*. MQSeries Integrator supplies several message parsers ready for use on known message templates and message headers.

You can create new parsers, using a system programming interface supplied by MQSeries Integrator, if you need to process other types of message. This is described in Part 2, “Programming a plug-in node or parser” on page 61.

The Control Center

The functions and facilities of MQSeries Integrator are controlled using a graphical interface known as the Control Center. The Control Center comes with comprehensive on-line help, and is described in *MQSeries Integrator Using the Control Center*.

You can use the Control Center to:

- Define your broker domain
- Work with message flows
- Organize your MRM messages
- Control your publish/subscribe network
- Manage your broker domain

The Control Center allows you to restrict access and authority to the functions it provides, so you can control who can do what within the broker domain.

Applications

Applications using messages to send or receive data can communicate in several ways. Applications written to the *point-to-point* model transfer information from one sender to one receiver. *Publish/subscribe* applications, on the other hand, transfer information from one or more sender to one or more receivers, with a third party acting as the intermediary so that the information requirements of the receiver are matched against the information that the sender provides.

Today, most MQSeries applications are using *point-to-point* communications. These applications might be using a one-way *send-and-forget* (or *datagram*) model, or a *request/reply* (client/server) model. Such messages can be sent to a message flow you have established in the broker, to carry out the required processing on the message before sending it on to the receiving application.

Brokers support a second type of communication model known as *publish/subscribe*. In this model, some applications (*publishers*) provide information, and others (*subscribers*) consume that information. You can also have applications that are both publishers and subscribers.

MQSeries Integrator overview

Publishers create messages and send them to one or more message flows at a local broker that support publish/subscribe. Each message has an associated *topic* that categorizes the information in the message. Subscribers register subscriptions with their local broker, specifying the types of publication they are interested in (determined, for example, by the topic and the contents of the message). When a broker receives a publication that matches a subscription that has been registered, it sends that publication to the subscriber. Brokers exchange subscriptions and publications with each other, so that subscribers can receive information published at any broker in the domain.

New and existing applications can take advantage of the broker functions through the MQSeries *Message Queue Interface* (MQI), or the MQSeries *Application Messaging Interface* (AMI). Both interfaces support point-to-point and publish/subscribe programming models. You can use the MQI to send messages that access broker functions. The AMI provides higher levels of function that are designed to simplify the messaging process, particularly for the publish/subscribe model.

If you have existing applications written to these interfaces, it should be possible, in many cases, to configure your message broker environment in such a way that the applications will run unchanged.

You can find information about the design of new applications, and the reuse of existing applications in the *MQSeries Integrator Introduction and Planning* book.

The remaining chapters in Part 1 of this book tell you how to write application programs that communicate with MQSeries Integrator Version 2.0.

Chapter 2. Writing application programs

Applications communicate with MQSeries Integrator by sending messages to the broker, or receiving messages from the broker, using MQSeries message queues. Before writing your application program, you need to decide on the following:

- The structure and format of the messages
- The queues used for sending and receiving messages
- The communication model (point-to-point, publish/subscribe, or both)
- The programming interface (Message Queue Interface or Application Messaging Interface)
- Other features (transactional processing, message ordering, message persistence)

All these aspects of the application design are covered in detail in the *MQSeries Integrator Introduction and Planning Guide*.

Some of the information in that book is summarized in this chapter, together with the information you need when writing programs to implement your design.

See “Sending and receiving messages” for information relevant to all applications.

See “Point-to-point messaging” on page 9 for specific information about point-to-point applications.

Go to Chapter 3, “Writing publish/subscribe applications” on page 11 for more information about publish/subscribe applications.

Sending and receiving messages

In both communication models (point-to-point and publish/subscribe) messages are sent to, and received from, an MQSeries Integrator broker using normal MQSeries message queues. Information needed by the broker is (optionally) encoded in an MQRFH2 rules and formatting header. This header is usually placed after the normal MQSeries message descriptor (MQMD), and before the body of the message.

You need to construct a message according to your chosen message template, including the header (if used), and send it to an input queue at the broker. This queue has to be set up by a system administrator, as an attribute of the input node of the message flow that will process your message (see *MQSeries Integrator Using the Control Center*).

If you are writing an application to receive a message from the broker, the queue that it arrives on is also set up by the system administrator (as an attribute of the output node of the message flow). In the case of publish/subscribe applications, you specify in the application which queue you want publications to be sent to.

You can use the MQSeries Message Queue Interface (MQI) or the MQSeries Application Messaging Interface (AMI) to send and receive these messages.

Message headers

MQSeries Integrator messages can have headers of the following types:

- MQRFH2
- MQRFH
- With no MQRFH2 or MQRFH header

MQRFH2

The MQRFH2 header is based on the MQRFH header, but it allows Unicode strings to be transported without translation, and it can carry numeric datatypes. New applications should use the MQRFH2 header so that they have access to all the functionality in MQSeries Integrator Version 2.0.

Following the fixed portion of the MQRFH2 header is a number of *NameValueLength* and *NameValueData* pairs. Each one of these contains one *folder*, which holds a sequence of *properties* encoded as name/value *elements* in XML (Extensible Markup Language: see the *MQSeries Integrator Introduction and Planning Guide* for more details). A <psc> folder contains publish/subscribe commands. An <mcd> folder contains a description of the message contents. This is used by MQSeries Integrator to decide which message parser to invoke if content-based operations are carried out in the message flow.

Full details of the MQRFH2 header and its contents are given in Chapter 4, “The MQRFH2 rules and formatting header” on page 29.

MQRFH or no header

Existing MQSeries Integrator Version 1 and MQSeries Publish/Subscribe applications using the MQRFH header are supported by MQSeries Integrator Version 2.0. For full details refer to *MQSeries Integrator Introduction and Planning*.

MQSeries messages that have no MQRFH2 or MQRFH header are also supported. The default message properties on a message flow can provide defaults for values normally carried in a header. This allows messages to be parsed and published.

The output message is unchanged if this function is used. See the information on **MQInput** node in *MQSeries Integrator Using the Control Center*.

Using the Message Queue Interface

The MQSeries Message Queue Interface (MQI) that is used to put (MQPUT) and get (MQGET) messages to and from queues is described in the *MQSeries Application Programming Guide* and the *MQSeries Application Programming Reference* book.

If you are using the MQI in your application programs, Chapter 4, “The MQRFH2 rules and formatting header” on page 29 describes the MQRFH2 header in detail, and the structure of the folders that are contained within it. Chapter 5, “Publish/subscribe command messages” on page 39 details the command messages that are sent to the broker, using the MQRFH2 header, in publish/subscribe applications.

Using the Application Messaging Interface

For many applications it is not necessary to understand the details of the MQRFH2 header or the MQI. The MQSeries Application Messaging Interface (AMI) has been developed to hide their complexities from an application programmer by containing them in policies and services that are set up by a systems administrator. MQSeries and MQSeries Integrator functionality, including publish/subscribe, can be accessed through the AMI using the C, C++, or Java™ programming languages.

Restriction

The MQSeries Application Messaging Interface is not available on all platforms, and the current version does not support the MQRFH2 header. For publish/subscribe applications, the AMI uses the MQRFH header as described in the *MQSeries Publish/Subscribe User's Guide*. Note that this does not support all of the publish/subscribe functionality described in Chapter 3, "Writing publish/subscribe applications" on page 11.

The latest information is available on the Internet (see "MQSeries information available on the Internet" on page ix).

If you are using the AMI, read the remainder of this chapter and then refer to the *MQSeries Application Messaging Interface* book. You will need to check with your system administrator to find out what policies and services have been defined for your application. If you are using the publish/subscribe capability of MQSeries Integrator, you should also read Chapter 3, "Writing publish/subscribe applications" on page 11.

Point-to-point messaging

MQSeries Integrator point-to-point applications involve these components:

- Sender** An application that sends a message
- Broker** The message broker that processes the message
- Receiver** An application that receives the message

There can be more than one sender or receiver in any system.

Send and forget

With send and forget (datagram) messages, no response is expected from the receiver. The sender puts a message to a queue at the broker. This should be the input queue defined for the required message flow as an attribute of the **MQInput** node. The queue manager is the one used by the broker. The sender needs to have authority to put to this queue.

If the sender and receiver applications already exist, you can use queue aliasing to route the sender's message to the broker, instead of sending it to the receiver. This means that you do not have to change the sending application.

The broker processes the message according to the message flow. If content-based filtering, routing, or message transformation is to be applied, the broker parses the message according to the structure of the message as defined in an <mcd> folder in the *NameValueData* field of the MQRFH2 header.

Point-to-point messaging

Having processed the message, the broker puts the output message onto the receiver's input queue. This queue, and its queue manager, are defined as attributes of the **MQOutput** node. Alternatively, the output node can specify that the output message is sent to the *ReplyToQ* defined in the message descriptor (MQMD).

If the message flow contains more than one input or output, each is treated in the same way as described above.

Request/reply

With request/reply messaging, after the receiver receives a request message it sends a reply back to the sender. The request message is handled as described for send and forget messages. There are two possibilities for the reply:

1. The receiver sends the reply message directly back to the sender, without involving the broker. The message is sent to the *ReplyToQ* in the message descriptor (MQMD) of the request message, which is passed unchanged by the broker.
2. The receiver sends the reply message to a reply message flow in the broker, so that it can be processed before reaching the sender. In this case the broker must replace the sender's *ReplyToQ* in the MQMD of the request message with the input queue name of the reply message flow.

The output of this reply message flow must go to the sender's *ReplyToQ*. If the name is fixed, there is no problem; otherwise, some means of associating this queue with the reply message is needed.

This can be done by setting up a message flow that stores the message descriptor of the original request message using a database node, and then retrieving it from the database in order to send the reply message to the correct destination.

Alternatively, the relevant details in the message descriptor can be copied into a folder in the MQRFH2 header, and carried with the message.

The message descriptor

Fields in the MQSeries message descriptor (MQMD) of an input message are usually passed unchanged to the output node.

However, if the message flow contains a transformation (such as a compute node), any of the MQMD fields might be changed according to how the message flow has been set up by the administrator.

Error handling

The handling of errors by an application program depends on how the message flow in the broker has been set up by the administrator.

Error events can be produced by any node in a message flow that has a failure output terminal. The failure output might be connected to an output node, in which case it can be directed to the sender of the message that caused the error. Otherwise, the error is passed to the input node of the message flow. From here it is returned to the backout-retry queue associated with the input queue.

If the broker is unable to put a message to the receiver's queue, the input node rolls back the transaction (if any) and puts the message to the backout-retry queue.

Chapter 3. Writing publish/subscribe applications

This chapter describes how to write applications that use the publish/subscribe model. If you are writing applications that use only the point-to-point model, you don't need to read this chapter.

Before writing your application program, you need to decide on the following:

- The topic trees used by publishers (including the use of wildcards by subscribers)
- The options used by publishers (retained, local, other subscribers only)
- If message ordering techniques are needed (sequence number, publication timestamp)
- The options used by subscribers (subscription point, filter, local, new publications only, publish on request only)
- The subscriber queues used to receive publications (with optional correlation identifiers)
- The persistence of published messages

All these aspects of the application design are covered in more detail the *MQSeries Integrator Introduction and Planning Guide*.

Some of the information in that book is summarized in this chapter, together with the information you need when writing programs to implement your design.

The following information is presented in this chapter:

- "Publish/subscribe messaging" on page 12
- "The publisher" on page 13
- "The subscriber" on page 16
- "The broker" on page 22
- "Sample application" on page 25
- "Using the AMI in publish/subscribe applications" on page 27

Publish/subscribe messaging

MQSeries Integrator publish/subscribe applications involve these components:

- Publisher** An application that generates publications
- Broker** The message broker that distributes the publications
- Subscriber** An application that subscribes to and receives publications

There are usually multiple publishers and subscribers in a publish/subscribe system, and there can be multiple brokers as well. Publishers can also be subscribers.

The following sections describe in detail the roles of the publisher, subscriber and broker. They communicate with each other by sending messages as shown in Figure 1.

If you are using the Message Queue Interface (MQI) to write applications, you need to understand the MQRFH2 header (see Chapter 4, “The MQRFH2 rules and formatting header” on page 29) that is used to send the command messages. These messages are described in Chapter 5, “Publish/subscribe command messages” on page 39, together with details of the message descriptor (MQMD) used when sending the messages.

If you are using the Application Messaging Interface (AMI) to write applications, you don’t need to understand the details of the MQMD and MQRFH2 header. After you have read this chapter, you might find it useful to look at Chapter 5, “Publish/subscribe command messages” on page 39 to see what options are available for each command. Then turn to the *MQSeries Application Messaging Interface* book.

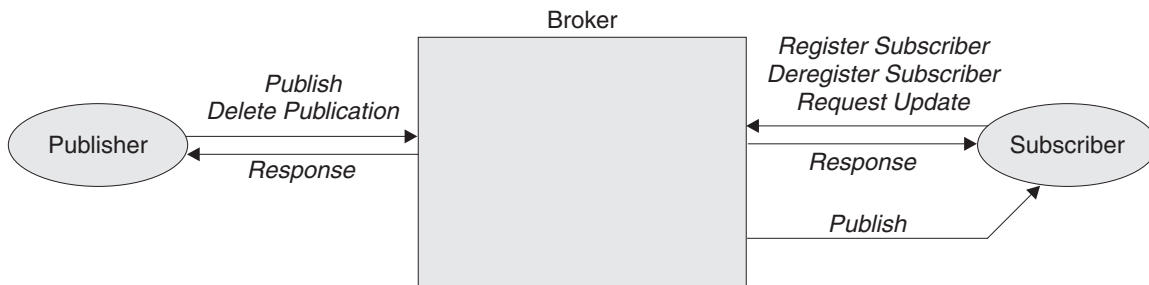


Figure 1. Communication between publisher, subscriber, and broker. The publisher can send Publish or Delete Publication messages to the broker. The broker forwards the Publish message to subscribers that have a matching subscription. The subscriber can send Register Subscriber, Deregister Subscriber, or Request Update messages to the broker. Optional Response messages from the broker are sent to the publisher and subscriber.

The publisher

The publishing application sends a **Publish** command message (see page 45) to the input queue of a message flow that contains a **Publication** node. The input queue is defined as an attribute of the **MQInput** node of the message flow. The publisher must have authority (set by MQSeries) to put a message to this queue. It must also have access authority (set by the MQSeries Integrator system administrator) for the topic or topics that are specified for this publication.

The command is contained within a <psc> folder in the *NameValueData* field of the MQRFH2 header. The publication data is in the body of the message, following the MQRFH2 header. The contents of the publication data are (optionally) described in an <mcd> folder in the *NameValueData* field of the header, so that content-based filtering can be applied to this publication.

If there is no MQRFH2 or MQRFH header, the default topic defined on the MQInput node is used. In such cases, **Publish** is assumed.

Topics

The topic that describes the publication is specified in the **Publish** message. Topics can be defined statically by the system administrator (see *MQSeries Integrator Using the Control Center*). In addition, if permitted by the system administrator, they can be defined dynamically in a **Publish** message. Subscriber access to publications is controlled by the system administrator on a topic basis.

Topic names are case sensitive. A topic name can contain any of the characters in the Unicode character set, including the space character, but it is recommended that topic names do not use the null character. Three characters have special meanings: the separator (/), the multi-level wildcard (#) and the single-level wildcard (+).

The separator (/) is used to denote levels within a topic name, for example employee/hire/development. This enables a hierarchy of topics to be used.

Note that publishers must specify a complete topic name, without wildcards.

It is permissible to specify more than one topic for a publication.

The topic string is not limited in length. However, topic strings become less efficient as they become longer or have more levels.

Retained publications

Publishers can specify that a publication is retained by the broker. Normally, a publication is discarded as soon as the broker has sent it to all of its current subscribers. If `RetainPub` is specified as a publication option in the **Publish** message, the publication is retained by the broker. It will replace any previously retained publication for that topic.

State and event information

Retained publications are useful for information about the current *state* of something, such as the price of stock or the score in a soccer match. When the stock price (or the score) changes, the previous state information is no longer needed. In contrast, publications about individual *events* contain information such as the sale of some stock, or the scoring of a particular goal. Each of these events is independent of other events.

When to use retained publications

Retained publications do not have to be used for state information. If all the subscriptions for a topic are in place before any publications are made, it is not necessary to retain the publication. Another reason is if state information is published at frequent intervals (for example, every few seconds). On the other hand, if publications *are* retained, a subscribing application that fails can request the current retained publication using the **Request Update** command message (see page 51) after a restart. (Otherwise, it might need to store a local copy of messages received.) See “Sample application” on page 25.

Because a retained publication is stored by each broker that has a subscription for its topic, there are performance and storage implications to be considered, especially if the publications are large.

Mixed publications

Mixing retained and non-retained publications on the same topic is not recommended but, if applications do this and publish a non-retained publication, any existing retained publication for that topic is still retained.

It is not recommended for two or more applications to publish retained publications to the same topic. If they do and the timing is close to simultaneous, it is indeterminate which publication is retained. If these publishers use two different brokers, it is possible that different retained publications for the same topic could be held at each broker.

Expiry of retained publications

Use the *Expiry* field of the message descriptor (MQMD) to set an expiry interval for a retained publication.

Deleting a retained publication

Retained publications can be deleted by sending a **Delete Publication** command message (see page 40) to the broker. It must be sent to the same input queue as the corresponding publication to be deleted. Authority to issue this command message is the same as the authority needed to publish messages for the specified topics.

Note that if different publishers publish information on the same topics, the information that is deleted might have originated from a different publisher. In fact a retained publication can be deleted by any user, subject to appropriate authority.

Local publications

Publishers can publish locally (by specifying the `Local` option) or globally (the default case). Local publications are not forwarded to other brokers, and are received only by subscribers registered at the same broker (whether those subscriptions specified `Local` or not). Local retained publications are retained only at this broker. It is acceptable for applications to publish and subscribe locally and globally (including retained publications) to the same topic at different brokers; each broker will deal with them in isolation from the other brokers where necessary.

Deleting a local publication

Note that a message published locally can be deleted by a global **Delete Publication** command (that is, without the `Local` option). Similarly, a message published globally can be deleted at the local broker by a local **Delete Publication** command, in which case the message will be absent from that one broker. Therefore, care should be taken if using local and global publications on the same topic.

Conference-type applications

The `OtherSubsOnly` (other subscribers only) option allows simpler processing of conference-type applications, where a number of individual applications all publish and subscribe to the same topic (such as ‘Conference’). Normally this means that each application will receive its own publications, since it has a matching subscription.

If this option is specified, it tells the broker not to send the publication to the subscriber queue associated with that application, so that an application can publish information into the conference without receiving that information itself.

Message ordering

Messages can be published by brokers in the same order as they are received from publishers, depending on the setting of the “order mode” attribute of the **MQInput** node (provided, of course, that the publisher sends all its publications on a given topic to the same input queue). This normally means that each subscriber receives messages from a particular broker, on a particular topic, from a particular publisher, in the order that they are published by that publisher.

However, as with all MQSeries messages, it is possible for messages to be delivered out of order. This could happen:

- If additional instances of the message flow are running.
- If a link in the network goes down and subsequent messages are rerouted along another link.
- If a queue becomes temporarily full, or put-inhibited. In this case a message is put to a dead-letter queue and therefore delayed, while subsequent messages might pass straight through.

If you need to ensure that your messages are delivered in the correct order in all circumstances, you can use one of the following strategies:

- A *sequence number* parameter (`SeqNum`). A publisher can include this with each message in the `<psc>` folder, increasing the value by one for each successive message that it publishes for the same topic. The broker does not check or set this parameter; the responsibility for it lies with the publisher. The number can

The subscriber

be checked by the subscriber, which needs to remember the last sequence number it received for that topic.

- A *publish timestamp* parameter (PubTime). A publisher can include this with each message <psc> folder (with or without the sequence number parameter). This is particularly useful if subscribers are only interested in the latest information; they can check whether the timestamp is greater than that of the last **Publish** message that they processed.

The publisher and subscriber might need to remember the sequence number or publish timestamp atomically with issuing or receiving a publication. This can be accomplished by saving the information on a queue, using the same unit-of-work as the one in which the publication is put or retrieved (see “Persistence and units of work” on page 23).

Routing messages that are not from an MQSeries Integrator publisher

Messages that do not originate from an MQSeries Integrator publisher can also be sent to subscribers even though the messages do not have the usual information needed to make a routing decision. This is done by setting the `Topic` property on the `MQInput` node. If a message arrives at the `MQInput` node's queue and does not have an `MQRFH2` header that contains a <psc> folder, the message is treated as if it was a `Publish` command with this default topic. The message will be sent to subscribers who have registered for the default topic. If a subscriber had included a filter as part of the registration, this would also be applied to a message of this type.

Notes:

1. The default topic will not be added to the message; the message is processed as if it contained the topic. The subscriber will receive the original message with no `MQRFH2` <psc> folder.
2. This type of message is also handled as if it were declared with the “local” publish option. The message will not be forwarded to neighboring brokers, even if they subscribed to the default topic.
3. Response messages will not be sent to publishing applications that produce messages in this way, even if the `MQMD` is set to imply that responses should be sent.
4. Existing applications that use `MQRFH` format subscriptions will not receive these messages.

The subscriber

The subscribing application sends a **Register Subscriber** command message (see “Register Subscriber” on page 48) to the broker, to specify what publications it wants to receive (defined by topic, filter, and subscription point) and the queue for receiving the publications (the subscriber queue). The command is contained within a <psc> folder in the *NameValueData* field of the `MQRFH2` header.

The command message is sent to the control queue at the broker. This is the `SYSTEM.BROKER.CONTROL.QUEUE` (which is compatible with MQSeries Publish/Subscribe applications). The subscriber must have authority (set by MQSeries) to put a message to this queue and to the subscriber queue. It must

also have access authority (set by the MQSeries Integrator system administrator) for the topic or topics that are registered in this subscription.

Subscriptions

A subscription consists of the following:

- One or more topics. Wildcards can be used.
- An optional subscription point.
- An optional filter on the contents of the publication message.
- A subscriber queue, queue manager, and optional *CorrelId*.

When the broker receives a publication that matches the topic, subscription point, and filter, it forwards the publication to the subscriber queue (unless the subscriber registered with the publish on request only option, as explained in “Retained publications” on page 20).

Topics and wildcards

Topics associated with publications are described in “Topics” on page 13. Multiple topics can be specified in subscriptions, and wildcards can be used.

The multi-level wildcard (#) matches any number of levels (including zero). It can be used only at the beginning or end of a topic name string.

The single-level wildcard (+) matches exactly one level. It can be used anywhere in a topic name string. However, MQSeries Integrator is optimized for wildcards at the end of the topic name. It is therefore recommended that applications structure their topics into subject trees, so that subscribers can subscribe to sub-trees by placing the multi-level wildcard at the end.

An additional level of selection can be achieved using a filter on the topic name. See “Filters” on page 18.

Subscription points

A subscription point is the name by which subscribers access publications at one or more **Publication** nodes.

Each Publication node has one subscription point name, and different Publication nodes can share the same name. A subscriber registering a subscription to a particular subscription point will receive publications from all of the Publication nodes that have the specified subscription point name. This applies to all message flows running in the broker, and to all brokers connected in the network (except for local publications).

By default, Publication nodes have a null subscription point name, and subscribers that do not specify a subscription point when they register will receive publications from all such nodes.

We recommend that you leave the subscription point name of your Publication nodes blank unless you have a real need to use subscription points other than the default. The use of non-default subscription points requires a considerable amount of extra processing and may, therefore, unfavorably impact the performance of your broker network.

The subscription point name must not be more than 64 characters in length.

Filters

You can specify a content-based filter to select publications according to their contents, in addition to specifying a topic and subscription point. MQSeries Integrator needs to know the structure of the message in order to parse its contents correctly. (The structure is defined by the domain, set, type and format, as described in “Message service folders” on page 36). This can be achieved in a number of ways:

- The message is a self-defining XML message.
- The message is defined by an <mcd> folder in the MQRFH2 header (see page 36).
- Otherwise, the message is assumed to be as defined in the attributes of the **MQInput** node.

The filter itself is entered as an expression with SQL syntax, for example:

```
Body.Name LIKE 'C%'
```

This means that the contents of a field called “Name” in the body of the input message (that is, the publication data that follows the MQRFH2 header) will be extracted and matched against the string given in the expression. % is a wildcard, meaning zero or more characters. If the name in the message starts with ‘C’, the expression evaluates to TRUE and so the publication will be sent to the subscriber.

For more details about filters, see Appendix A, “Using filters in content-based routing” on page 165.

Note: In general, subscriptions to “#” should be avoided because, when using a multi-broker environment, they will cause a greater proportion of publications to be sent between brokers.

Subscriber queues

A publication is delivered to the queue and queue manager specified by the subscriber either by using <QName> and <QMName> tags or by using values taken from the MQMD (which is the default). This is the *subscriber queue*.

If required, the subscriber queue can be a temporary dynamic queue. In this case, the broker will deregister the subscription automatically if the queue is deleted (for example, when the subscriber disconnects from the queue manager). For optimal broker performance, we recommend that subscribing applications should deregister their subscriptions before terminating. If the application fails to deregister, the broker will automatically remove the subscriptions when it sees that the queue has been deleted. Note that automatic deregistration will not work if:

- The dynamic temporary queue is not local (that is, it is not on the same queue manager on which the broker is running)
- The subscriber has named a queue that is an alias of a local temporary dynamic queue

A correlation identifier can be included if required. This allows several applications to share a queue, which might be desirable if there are many clients. It also allows a single application to distinguish between publications arising from different subscriptions.

If a subscribing application has more than one subscription that matches a publication, only one copy of the publication is sent to it unless it registered with different subscriber identifiers (a combination of the MQSeries queue, queue manager, and (optionally) correlation identifier). In which case, more than one copy might be sent to the application).

The subscriber queue should not in general be the same as any defined in a message flow, because this would cause the published message to be republished. However, such a restriction is not imposed by the broker, and it is possible to use this behavior to chain message flows together dynamically.

Registration

A subscriber can register multiple times with the same or different brokers as necessary. An application can be both a subscriber and a publisher.

An existing subscriber can re-register in order to increase the range of topics, subscription points, or filters for which it wants to receive information. Similarly it can change its registration options or expiry time for a given combination of topic, subscription point and filter for which it is already registered. Only the application that originally registered a subscription can update it.

Note: When registering again, unspecified options are assumed to take their default values; they do **not** remain unchanged.

Local subscriptions

A subscriber can specify the `Local` option when registering a subscription. In this case the broker does not forward the subscription to other brokers in the network. The subscriber will not receive publications that are published to other brokers, only those published to the broker at which it registers its subscription.

Subscription expiry

The *Expiry* interval in the message descriptor (MQMD) of the **Register Subscriber** command message determines when the subscription expires. If this is set to `MQEI_UNLIMITED`, the subscription does not expire. If a subscription is re-registered, the subscription's expiry time is updated to the value of *Expiry* in the MQMD of the re-registration message.

Deregistration

One or more subscriptions for a particular subscriber can be deregistered using the **Deregister Subscriber** command message (see page 42). This is sent to the broker control queue, `SYSTEM.BROKER.CONTROL.QUEUE`. The message must be sent by the subscriber that registered the subscription in the first place.

There are other ways in which a subscription can be deregistered:

- The subscription expires, as explained above.
- A system administrator deregisters the subscription (see *MQSeries Integrator Using the Control Center*).
- If the subscriber queue is a temporary dynamic queue, and the queue is deleted (for example, when the subscriber disconnects from the queue manager), the broker will deregister the subscription automatically. However, see the restrictions listed in "Subscriber queues" on page 18.

Flow of messages

When a subscriber application sends a message to deregister a subscription, and receives a response message to say that this was processed successfully, it is possible that some publications will subsequently reach the subscriber queue if they were being processed by the broker at the same time as the deregistration. If you do not handle this situation, it could result in a buildup of unprocessed messages on the subscriber queue. We recommend that your application does a loop that includes an MQGET call with the appropriate CorrelId after sleeping for a while, to clear any such messages off the queue.

Also, if the subscriber uses a permanent dynamic queue and, when completing, it deregisters and closes the queue with the PurgeandDelete option, it is possible that the queue will not be empty. This is because publications from the broker might not yet be committed at the time that queue was deleted. In this case, a Q_NOT_EMPTY return code will be issued by the MQCLOSE call. The application can avoid this problem by sleeping and reissuing the MQCLOSE call from time to time.

Retained publications

Retained publications are normally sent directly to subscribers that have matching subscriptions. A new subscriber will be sent the current retained publication immediately after registering, unless it specified the NewPubsOnly option when it registered the subscription (in which case only new publications are sent to it).

If a subscriber registers with the PubOnReqOnly option (publish on request only) the subscriber will not receive the current retained publication until it sends a **Request Update** command message to the broker control queue SYSTEM.BROKER.CONTROL.QUEUE (see page 51). Note that with this option the subscriber will not receive any non-retained publications.

A subscriber that did not register with PubOnReqOnly can also use **Request Update** at any time. This might be necessary if the subscriber had already received the publication in the normal way, but had failed without saving it, and on restart wants to receive it again.

A subscriber can request to be informed if the subscriber is being told that the publication was sent to it as a result of a subscriber request update operation. Normal publications (even retained ones) will not have IsRetainedPub set. This is done by specifying the InformIfRet option in the **Register Subscriber** message. The broker will then set the IsRetainedPub publication option in the **Publish** message when it forwards a retained publication to the subscriber.

Flow of publish/subscribe messages

Figure 2 on page 21 shows the flow of messages in a simple publish/subscribe system with no retained publications. It is assumed that a message flow consisting of at least one **MQInput** node and a **Publication** node has been set up in the broker. The input queue in the diagram relates to this **MQInput** node. It is also assumed that the subscriber registers its subscription to the same topic that the publisher is using. Note that the subscriber does not receive publications that were published before it registered its subscription.

Figure 3 on page 21 shows the use of retained publications. In this case the subscriber receives a retained publication that was published *before* the subscriber registered its subscription.

In Figure 4 on page 22, the subscriber registers its subscription with the `PubOnReqOnly` option, so it does not receive the retained publication until it sends a **Request Update** message to the broker (by which time the earlier retained publication has been replaced by a later one).

Note: The following figures assume that the publishers and subscribers have not requested responses.

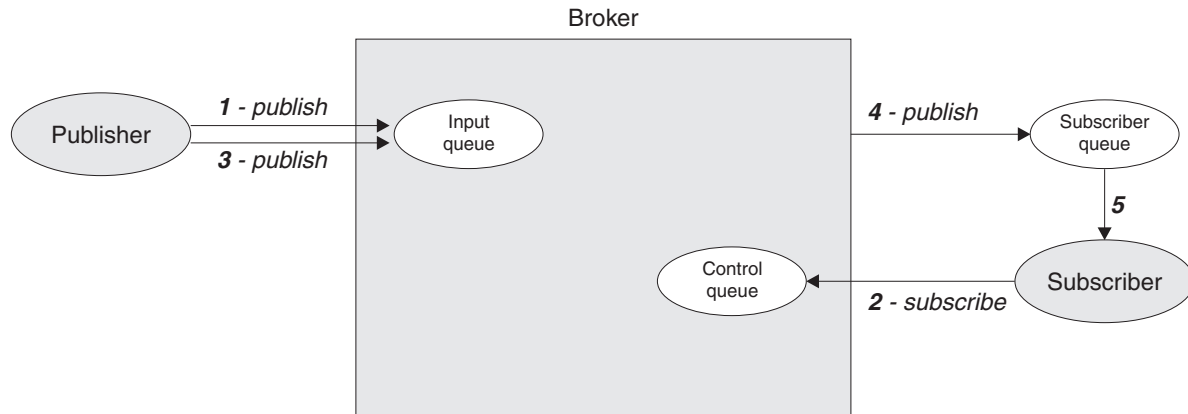


Figure 2. Publish/subscribe without retained publications. The publisher sends a publication to the input queue (1). The subscriber does not receive this publication because it has not yet registered a subscription. After it subscribes (2), the next publication (3) is sent to the subscriber queue (4), from where the subscriber can get it (5).

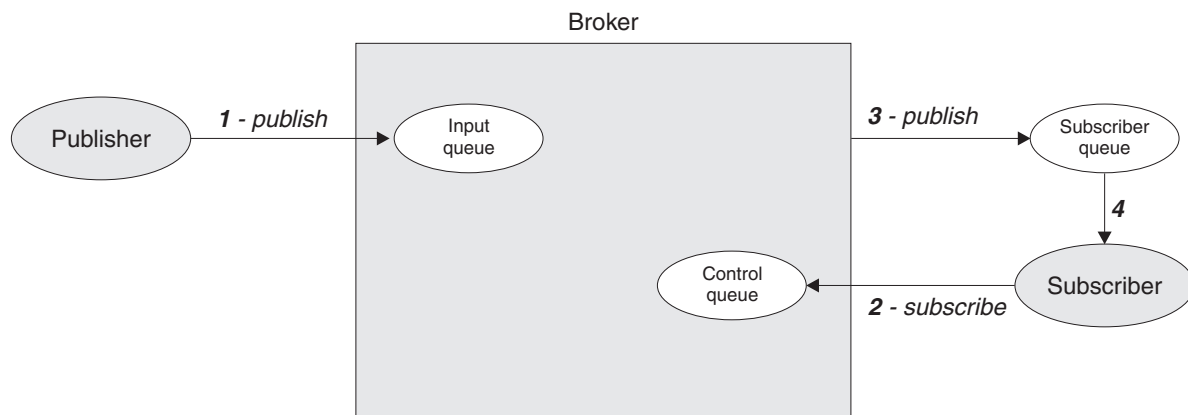


Figure 3. Using retained publications. The publisher sends a retained publication to the broker (1). When the subscriber registers (2), the current retained publication is sent to the subscriber queue (3), from where the subscriber can get it (4). Subsequent publications will be sent to the subscriber straightaway.

The broker

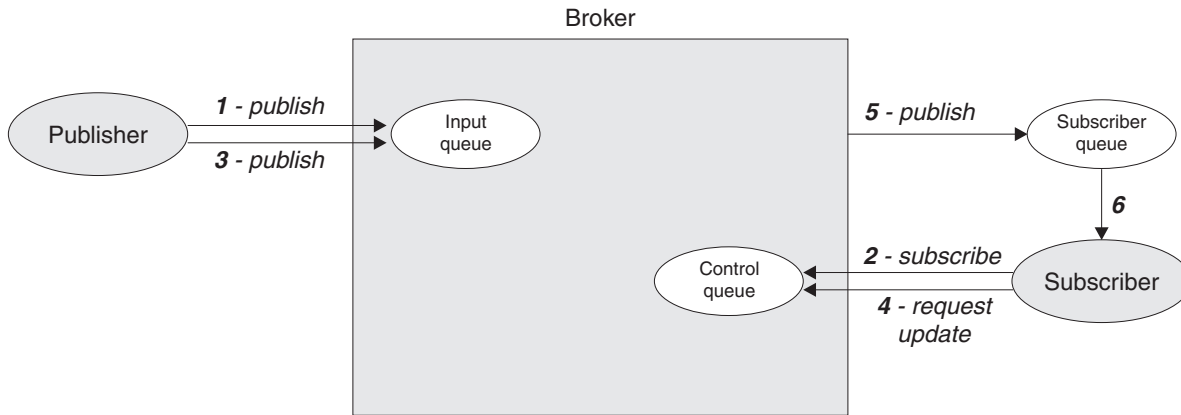


Figure 4. Publish on request only. The subscriber registers a subscription with "publish on request only" (2). Although there is a current retained publication (1), it is not sent to the subscriber until it uses "request update" (4). By this time, the retained publication has been replaced by a new publication (3) which is sent to the subscriber queue (5), from where the subscriber can get it (6).

The broker

The broker forwards **Publish** messages to subscribers, for publications that match each subscription (defined by topic, filter, and subscription point). The publications are sent to the subscriber queue as defined in the **Register Subscriber** command message.

Unless the `Local` option is specified, a broker sends subscription registrations to other brokers in the network. Matching publications will then be forwarded to that broker for distribution to its subscribers.

Only one copy of a publication is sent to each subscriber, regardless of how many matching subscriptions that subscriber has (unless it registered with different subscriber queues or correlation identifiers).

Publications sent from brokers have their message descriptor changed. Refer to "MQMD for publications forwarded by a broker" on page 56.

Broker response messages

The broker can send a message to a publisher or subscriber in response to a command message. A response message has a similar format to a command message, but it is contained in a `<pscr>` folder in the `NameValueData` field of the `MQRFH2` header.

The response message is sent to the queue identified by the `ReplyToQ` and `ReplyToQMgr` fields in the message descriptor (MQMD) of the command message. The persistence of the response message is set to the same value as for the command message. If the `ReplyToQ` is a temporary dynamic queue, the command message must be non-persistent. The `MsgType` and `Report` fields in the MQMD, together with the success or failure of the command, determine whether the response message is sent.

The broker can generate three types of response:

- ok** The command completed successfully
- warning** The command was only partially successful
- error** The command failed

For further details, see “Broker Response” on page 53.

Brokers do not request publishers or subscribers to generate responses to messages from the broker.

Broker restarts

Subscription registrations and retained publications are maintained across broker restarts. After a restart, any subsequent publications for the specified topics will be forwarded to the application. In addition, if the broker has any retained publications for these topics, the application can request to receive them after the restart using **Request Update**.

Persistence and units of work

Subscriber registration messages should normally be sent as persistent messages.

Brokers maintain the persistence and priority of publications as set by the publisher, unless changed by options in the **Register Subscriber** command (see page 48), or by the Access Control List (see MQSeries Integrator Using the Control Center). Publications will be delivered as non-persistent messages if the ‘Persistent’ flag in the Access Control List is set to False (the default), regardless of the persistence set by the publisher or the subscriber.

If a publication matches more than one subscription for an application, the persistence of the publication delivered to the subscriber queue is determined according to the following rules:

Subscription persistence	Resulting publication persistence
All subscriptions are non-persistent	Non-persistent
At least one subscription is set to persistence as publisher, all others are non-persistent	Persistence as publisher
At least one subscription is set to persistence as queue, all others are non-persistent	Persistence as queue
At least one subscription is set to persistent, or one is set to persistence as publisher and another is set to persistence as queue	Persistent

When reading messages from input queues, brokers always read persistent messages within a unit-of-work, so that they are not lost if the broker or system crashes. Non-persistent messages might or might not be read within a unit-of-work, depending on the setting of the attributes of the **MQInput** node.

The broker

Publication messages are treated so that publication to subscribers is once and once only for persistent messages. For non-persistent messages, delivery to subscribers is also once only unless *SyncPointIfPersistent* was specified in the queue manager configuration file and the broker or queue manager stops abruptly. In this case, the message might be lost for one or more subscribers. Regardless of its persistence, however, a **Publish** message is never sent more than once to a subscriber, for a given subscription (unless it is explicitly requested using the **Request Update** command).

If the subscriber queue is a temporary dynamic queue, the subscription request must specify non-persistent delivery of publications or else it will be rejected by the broker.

Publishers and subscribers can choose whether to use a unit-of-work when publishing or receiving messages. However, if the *SequenceNumber* technique described previously is used for maintaining ordering, both publisher and subscriber must retain sequencing information atomically with putting or getting a message if the application is to be restartable.

Sample application

One of the sample applications provided with MQSeries Integrator Version 2.0 uses publish/subscribe to simulate a results gathering service that reports the latest score in a sports event such as a soccer match. It receives information from one or more instances of a soccer match simulator that scores goals at random for the two teams. This is illustrated in Figure 5.

Refer to the *MQSeries Integrator Installation Guide* for details of where to find this sample application.

The match simulator does not keep track of the score. It merely indicates when a match starts or finishes, and when a goal is scored. These events are published to three different topics on the MQSI_SOCKER_PUBLICATION_QUEUE queue.

- When a match starts, the names of the teams are published on the Sport/Soccer/Event/MatchStartedtopic.
- When a goal is scored, the name of the team scoring the goal is published on the Sport/Soccer/Event/ScoreUpdate topic.
- When a match ends, the names of the teams are published on the Sport/Soccer/Event/MatchEnded topic.

The publications on these topics are *not* retained, as they contain event information and not state information.

The results service subscribes to the topic Sport/Soccer/Event/# to receive publications from any matches that are in progress. It keeps track of the current score in each match, and whenever there is a change it publishes the score as a *retained* publication on the following topic:

Sport/Soccer/State/LatestScore/Team1 Team2,

where Team1 and Team2 are the names of the teams in the match.

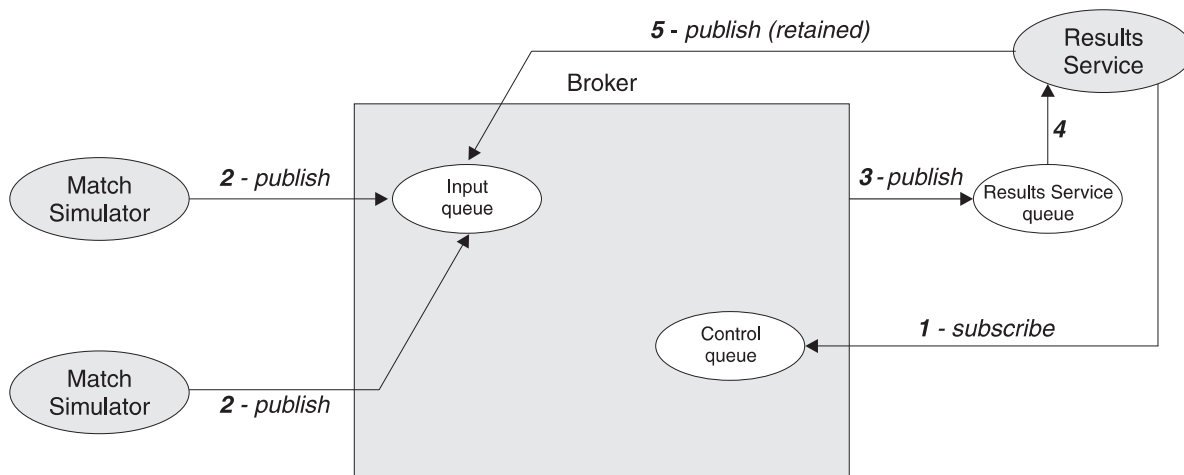


Figure 5. The results service application. The results service subscribes to the topics "match started", "score update" and "match ended" (1). When the match simulators publish event information on these topics (2), the publications are forwarded to the results service queue (3), from where the results service application gets them (4). The results services then publishes the latest scores (which are state information) as retained publications (5).

Sample application

A subscriber wanting to receive all the latest scores could register a wildcard subscription to the following topic:

```
Sport/Soccer/State/LatestScore/#
```

If it was interested in one particular team only, it could register different wildcard subscriptions to the following topics:

```
Sport/Soccer/State/LatestScore/MyTeam/+  
Sport/Soccer/State/LatestScore/+MyTeam
```

Note that the results service must be started before the match simulators, otherwise it might miss some events and hence not be able to ascertain the current state in each match. This is usually the case with event publications, in which subscriptions are static and need to be in place before publications arrive.

If it stops while matches are still in progress the results service can find out the state of play when it restarts. This is done by subscribing to its own retained publications using the `Sport/Soccer/State/LatestScore/#` topic, with the 'Publish on Request Only' option. A **Request Update** command is then issued to receive any retained publications which contain latest scores.

These publications enable the results service to reconstruct its state as it was when it stopped. It can then process all events that occurred while it was stopped by processing the subscription queue for the `Sport/Soccer/Events/#` topic. Because the subscription will still be registered (no **Deregister Subscriber** message has been sent) it will include any event publications that arrived while the results service was inactive.

This sample program illustrates the following aspects of a publish/subscribe application:

- Event information (not retained).
- State information (retained publication).
- Wildcard matching of topic strings.
- Multiple publishers on the same topics (non-retained publications only).
- The need to subscribe to a topic *before* it is published on (non-retained publications).
- A subscriber continuing to be sent publications when that subscriber (not its subscription) is interrupted.
- The use of retained publications to recover state after a subscriber failure.

Using the AMI in publish/subscribe applications

The MQSeries Application Messaging Interface (AMI) has functions to generate a number of publish/subscribe command messages, and to receive a publication from the broker. The name of the function (or method) depends on the programming language being used (C, C++, or Java). In the case of C, there are two sets of functions: the high-level interface and the object-level interface.

The AMI can be downloaded free from the Internet, complete with sample applications that demonstrate how to use its publish/subscribe function. See <http://www.ibm.com/software/mqseries/txppacs/>

AMI publish/subscribe functions

Publish command

C high-level	amPublish
C object-level	amPubPublish
C++	AmPublisher->publish
Java	AmPublisher.publish

Register Subscriber command

C high-level	amSubscribe
C object-level	amSubSubscribe
C++	AmSubscriber->subscribe
Java	AmSubscriber.subscribe

Deregister Subscriber command

C high-level	amUnsubscribe
C object-level	amSubUnsubscribe
C++	AmSubscriber->unsubscribe
Java	AmSubscriber.unsubscribe

Receive a publication

C high-level	amReceivePublication
C object-level	amSubReceive
C++	AmSubscriber->receive
Java	AmSubscriber.receive

These functions have parameters that enable you to specify some of the properties in the command message, such as the *topic*. Other properties in the command message are specified by the AMI *service* and *policy* that you use to send the message (services and policies are set up by the system administrator). You can modify these properties by changing the appropriate name/value elements before sending the command message, and helper functions are provided for this purpose. Details of these name/value elements and the options that are available for each command are given in Chapter 5, "Publish/subscribe command messages" on page 39.

There are no AMI functions to generate **Delete Publication** or **Request Update** command messages directly. You have to construct a message containing the appropriate name/value elements using the helper functions provided, and then send the message to the broker.

Refer to the *MQSeries Application Messaging Interface* book for details of how to use the functions mentioned above (including the name/value element helper functions).

Restriction

The current release of the MQSeries Application Messaging Interface does not support the MQRFH2 header. The publish/subscribe functions listed above generate command messages with the MQRFH header, using the syntax defined in the *MQSeries Publish/Subscribe User's Guide*. Because MQSeries Integrator Version 2.0 is compatible with MQSeries Publish/Subscribe, applications using the AMI will work. However, some of the options in MQSeries Integrator Version 2.0, such as message content filtering, will not be available. If you want to use message content filtering you will have to construct a complete message, including the MQRFH2 header, and send it to the broker using the MQI.

Chapter 4. The MQRFH2 rules and formatting header

The MQRFH2 header is used to pass messages to and from the MQSeries Integrator Version 2.0 broker. The MQRFH2 header follows the MQSeries message descriptor (MQMD) and precedes the message body (if present). Other headers, such as the IMS or CICS bridge headers, are allowed before or after the MQRFH2 header.

If you are using the Message Queuing Interface (MQI) to write application programs, you need to understand this header. In addition, if your application uses the publish/subscribe model, you should read Chapter 5, “Publish/subscribe command messages” on page 39.

MQRFH2 Structure

The following table summarizes the fields in the structure.

Field	Description	Page
<i>StrucId</i>	Structure identifier	30
<i>Version</i>	Structure version number	30
<i>StrucLength</i>	Total length of MQRFH2 including <i>NameValueData</i>	31
<i>Encoding</i>	Numeric encoding of data that follows <i>NameValueData</i>	31
<i>CodedCharSetId</i>	Character set identifier of data that follows <i>NameValueData</i>	31
<i>Format</i>	Format name of data that follows <i>NameValueData</i>	31
<i>Flags</i>	Flags	32
<i>NameValueCCSID</i>	Character set identifier of <i>NameValueData</i>	32
<i>NameValueLength</i>	Length of <i>NameValueData</i> . Can be repeated as many times as required.	32
<i>NameValueData</i>	Name/value data. Can be repeated as many times as required.	32

Purpose:

The MQRFH2 structure contains information about the structure and intended consumers of a message that allows an MQSeries Integrator broker to process it and to deliver or publish it to those consumers.

Format name:

This is the value that should be put in the Format field of the preceding header (usually the MQMD). The value is MQRFH2bb, where ‘b’ is a blank. MQFMT_RF_HEADER_2 is a constant defined to hold this value.

MQRFH2 – Version field

For the C programming language, the constant `MQFMT_RF_HEADER_2_ARRAY` is also defined; this has the same value as `MQFMT_RF_HEADER_2`, but is an array of characters instead of a string.

Character set and encoding:

The character set and encoding of the fields in the MQRFH2 are as follows:

- Fields other than *NameValueData* are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes the MQRFH2, or by those fields in the MQMD structure if the MQRFH2 is at the start of the application message data. The character set should be one that has single-byte characters for the characters that are valid in queue names.
- *NameValueData* is in the character set given by the *NameValueCCSID* field. Only certain Unicode character sets are valid for *NameValueCCSID* (see the description of *NameValueCCSID* for details).

Some character sets have a representation that is dependent on the encoding. If *NameValueCCSID* is one of these character sets, *NameValueData* must be in the same encoding as the other fields in the MQRFH2.

- The user data (if any) that follows *NameValueData* can be in any supported character set (single-byte, double-byte, or multi-byte), and any supported encoding.

Availability:

This structure is available in MQSeries V5.1 environments that have CSD 2 applied.

Data conversion of this structure is supported in these environments.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value must be:

`MQRFH_STRUC_ID`

Identifier for rules and formatting header structure.

For the C programming language, the constant `MQRFH_STRUC_ID_ARRAY` is also defined; this has the same value as `MQRFH_STRUC_ID`, but is an array of characters instead of a string.

The initial value of this field is `MQRFH_STRUC_ID`.

Version (MQLONG)

Structure version number.

The value must be:

`MQRFH_VERSION_2`

Version-2 rules and formatting header structure.

The initial value of this field is `MQRFH_VERSION_2`.

StrucLength (MQLONG)

Total length of MQRFH2 including *NameValueData*.

This is the length in bytes of the MQRFH2 structure, including the *NameValueLength* and *NameValueData* fields at the end of the structure. It is valid for there to be multiple pairs of *NameValueLength* and *NameValueData* fields at the end of the structure, in the sequence: length1, data1, length2, data2, *StrucLength* does *not* include any user data that may follow the last *NameValueData* field at the end of the structure.

StrucLength must be set to a multiple of four; otherwise, problems with data conversion of the user data might occur in some environments.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *NameValueLength* and *NameValueData* fields:

MQRFH_STRUC_LENGTH_FIXED_2

Length of fixed part of MQRFH2 structure.

The initial value of this field is MQRFH_STRUC_LENGTH_FIXED_2.

Encoding (MQLONG)

Numeric encoding of data that follows *NameValueData*.

This specifies the representation used for numeric values in the data (if any) that follows the last *NameValueData* field. This applies to binary integer data, packed-decimal integer data, and floating-point data.

The initial value of this field is MQENC_NATIVE.

CodedCharSetId (MQLONG)

Character set identifier of data that follows *NameValueData*.

This specifies the coded character set identifier of character strings in the data (if any) that follows the last *NameValueData* field. The following special value can be specified:

MQCCSI_INHERIT

Inherit character-set identifier of current structure.

Character data in the data that follows the current structure is in the same character set as the current structure.

The initial value of this field is MQCCSI_INHERIT.

Format (MQCHAR8)

Format name of data that follows *NameValueData*.

This specifies the format name of the data (if any) that follows the last *NameValueData* field.

The name should be padded with blanks to the length of the field. Do not use a null character to terminate the name before the end of the field, because the queue manager does not change the null and subsequent characters to blanks in the MQRFH2 structure. Do not specify a name with leading or embedded blanks.

The initial value of this field is MQFMT_NONE.

MQRFH2 – NameValueData field

Flags (MQLONG)

Flags.

The following value must be specified:

MQRFH_NONE

No flags.

The initial value of this field is MQRFH_NONE.

NameValueCCSID (MQLONG)

Character set identifier of *NameValueData*.

This specifies the coded character set identifier of the data in the *NameValueData* field. This is different from the character set of the other strings in the MQRFH2 structure, and can be different from the character set of the data (if any) that follows the last *NameValueData* field at the end of the structure.

NameValueCCSID must have one of the following values:

CCSID	Description
1200	UCS-2 open-ended
13488	UCS-2 2.0 subset
17584	UCS-2 2.1 subset (includes the euro symbol)
1208	UTF-8

For the UCS-2 character sets, the encoding (byte order) of the *NameValueData* must be the same as the encoding of the other fields in the MQRFH2 structure. Surrogate characters (X'D800' through X'DFFF') are not supported.

The initial value of this field is 1208.

The following two fields are optional, but if present they must occur as a pair. They can be repeated as a pair as many times as required, that is, if they occur multiple times they must occur in the sequence: length1, data1, length2, data2,

Note: Because these fields are optional, they are omitted from the declarations of the structure that are provided for the various programming languages supported.

NameValueLength (MQLONG)

Length of *NameValueData*.

This specifies the length in bytes of the data in the *NameValueData* field. To avoid problems with data conversion of the data (if any) that follows the *NameValueData* field, *NameValueLength* should be a multiple of four.

NameValueData (MQCHARn)

Name/value data.

This is a variable-length character string containing data encoded using an XML-like syntax. The length in bytes of this string is given by the *NameValueLength* field that precedes the *NameValueData* field. This length should be a multiple of four.

Note: Because the length of *NameValueData* is not fixed, the field is omitted from the declarations of the structure that are provided for the various programming languages supported.

The string consists of a single “folder” that contains zero or more properties. The folder is delimited by XML start and end tags whose name is the name of the folder:

```
<folder> property1 property2 ... </folder>
```

Characters following the folder end tag, up to the length defined by *NameValueLength*, must be blank. Within the folder, each property is composed of a name and a value, and optionally a data type:

```
<name>value</name>
```

In these examples:

- The delimiter characters (<, =, ", /, and >) must be specified exactly as shown.
- name is the user-specified name of the property; see below for more information about names.
- value is the user-specified value of the property; see below for more information about values.
- Blanks are significant between the > character which precedes a value, and the < character which follows the value. Elsewhere, blanks can be coded freely between tags, or preceding or following tags (for example, in order to improve readability); these blanks are not significant.
- You must not use null as a pad character.

If properties are related to each other, they can be grouped together by enclosing them within XML start and end tags whose name is the name of the group:

```
<folder> <group> property1 property2 ... </group> </folder>
```

Groups can be nested within other groups, without limit, and a given group can occur more than once within a folder. It is also valid for a folder to contain some properties in groups and other properties not in groups.

Names of properties, groups, and folders: Names of properties, groups, and folders must be valid XML tag names, with the exception of the colon character, which is not permitted in a property, group, or folder name. In particular:

- Names must start with a letter or an underscore. Valid letters are defined in the W3C XML specification, and consist essentially of Unicode categories Ll, Lu, Lo, Lt, and Nl.
- The remaining characters in a name can be letters, decimal digits, underscores, hyphens, or dots. These correspond to Unicode categories Ll, Lu, Lo, Lt, Nl, Mc, Mn, Lm, and Nd.
- The Unicode compatibility characters (X'F900' and above) are not permitted in any part of a name.
- Names must not start with the string XML in any mixture of upper or lower case.

In addition:

- Names are case-sensitive. For example, ABC, abc, and Abc are three different names.

MQRFH2 - Initial values

- Each folder has a separate name space. As a result, a group or property in one folder does not conflict with a group or property of the same name in another folder.
- Groups and properties occupy a single name space within a folder. As a result, property cannot have the same name as a group within the folder containing that property.

Generally, programs that analyze the *NameValueData* field should ignore properties or groups that have names that the program does not recognize, provided that those properties or groups are correctly formed.

Values of properties: The value of a property can consist of any characters, except as detailed below:

- If the value contains any of the following characters, each occurrence of the character must be replaced by the corresponding escape sequence:

Character Escape sequence

& &
< <

- The following escape sequences are also defined, but their use is optional:

Character Escape sequence

> >
" "
' '

Note: The & character at the start of an escape sequence must *not* be replaced by &. For example:

```
<Filter>&quot;Body.Field1"&lt;> '&amp;hello&apos;</Filter>
```

which translates as:

```
<Filter>"Body.Field1"<> '&hello'</Filter>
```

Initial values

Table 2 (Page 1 of 2). Initial values of fields in MQRFH2

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQRFH_STRUC_ID	'RFHb' (See note 1)
<i>Version</i>	MQRFH_VERSION_2	2
<i>StrucLength</i>	MQRFH_STRUC_LENGTH_FIXED_2	36
<i>Encoding</i>	MQENC_NATIVE	See note 2
<i>CodedCharSetId</i>	MQCCSI_INHERIT	-2
<i>Format</i>	MQFMT_NONE	'bbbbbbbb'
<i>Flags</i>	MQRFH_NONE	0x00000000

Table 2 (Page 2 of 2). Initial values of fields in MQRFH2		
Field name	Name of constant	Value of constant
<i>NameValueCCSID</i>	None	1208
<p>Notes:</p> <ol style="list-style-type: none"> 1. The symbol 'b' represents a single blank character. 2. The value of this constant is environment-specific. 3. In the C programming language, the macro variable MQRFH2_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure: MQRFH2 MyRFH2 = {MQRFH2_DEFAULT}; 		

Definition for the C programming language

This structure is defined in the cmqc.h header file. The constants that are used within the *NameValueData* field are defined in the BipRfc.h header file.

```
typedef struct tagMQRFH2 {
    MQCHAR4  StrucId;          /* Structure identifier          */
    MQLONG   Version;         /* Structure version number     */
    MQLONG   StrucLength;     /* Total length of MQRFH2 including
                               NameValueData                */
    MQLONG   Encoding;       /* Numeric encoding of data that follows
                               NameValueData                */
    MQLONG   CodedCharSetId; /* Character set identifier of data that
                               follows NameValueData        */
    MQCHAR8  Format;         /* Format name of data that follows
                               NameValueData                */
    MQLONG   Flags;          /* Flags                        */
    MQLONG   NameValueCCSID; /* Character set identifier of NameValueData
                               */
} MQRFH2;
```

Note: Because *NameValueData* and *NameValueLength* are optional fields, they are omitted from the above definition.

Message service folders

The following folder names are defined for use by MQSeries products:

- <mcd> Message content descriptor
- <psc> Publish/subscribe command
- <pscr> Publish/subscribe command response
- <usr> Application (user) defined properties

Each folder is contained in a separate NameValueData field, each with a preceding NameValueLength field.

Other names are available for use by independent software vendors. To avoid naming problems, we recommend that vendors prefix their chosen folder name with their internet domain name. For example, a vendor with domain name ourcompany.com should name its folders in this way:

com.ourcompany.xxx

or

com.ourcompany.ourData

The mcd folder

The <mcd> folder can contain the following elements that describe the structure of the message data in an MQSeries message. They are all character strings.

- <Msd> Message service domain
- <Set> Message set
- <Type> Message type
- <Fmt> Message format

The domain element identifies how to handle the message. The syntax of the other elements (set, type, and format) depend on the value assigned to <Msd>.

The following values for <Msd> have been allocated:

mrm MQSeries Integrator MRM-managed messages. This domain supports the following values for <Fmt>:

xml

XML representation

pdf

MTI bitstream representation

CWF identifier

The custom wire format identifier that you assigned to your message set in the Control Center.

The Set element should contain the identifier of the MRM message set to which the message belongs (this identifier is obtained from the Control Center). The Type element value is the identifier of the MRM message definition (within the specified message set) to which this message belongs.

neon	The message is parsed by the MQSeries Integrator Version 1 message parser. The values for message set, type, and format are mapped to MQSeries Integrator Version 1 equivalents. The Type element should contain the name of the MQSeries Integrator Version 1 message format, as defined in the MQSeries Integrator Version 1 user interface. The Set element normally contains the name of the MQSeries Integrator Version 1 Application Group; however, for Publication purposes, this is not relevant and can be omitted. The Fmt element is not used and is ignored.
none	The message is treated as an opaque blob, and delivered to the recipient as is. If this domain is chosen, the set and type must not be specified.
+	xml The message is treated as a self-defining XML message.

An alternative to having an <mcd> folder with <Msd> set to none or XML is to set the Format field of the MQRFH2 header to MQFMT_NONE or "xml", respectively, omitting the <mcd> folder completely from the MQRFH2 header in both cases.

The psc folder

The <psc> folder is used to convey publish/subscribe command messages to the broker. Only one psc folder is allowed in the NameValueData field.

See Chapter 5, "Publish/subscribe command messages" on page 39 for full details.

The pscr folder

The <pscr> folder is used to contain information from the broker, in response to publish/subscribe command messages. There is only one pscr folder in a response message.

See "Broker Response" on page 53 for full details.

The broker ignores this folder in messages that it receives from publishing or subscribing applications.

The usr folder

The content model of the <usr> folder is as follows:

- Any valid XML name can be used as an element name, providing that it doesn't contain a colon
- Only simple elements are permitted (no grouping)
- All elements assume a default type of string, unless modified by a dt="xxx" attribute
- All elements are optional, but should occur at most once in a folder
- An MQRFH2 instance can contain at most one <usr> folder

Multiple MQRFH2 headers

It is possible for a message to have more than one MQRFH2 header: for instance if one application forwards a message, including its header, to another application. In this case, the one added later precedes the original header.

Message service folders

- Attributes that describe the body of the message, such as the domain, set, type, and format, or the CCSID and encoding, are taken from the *last* MQRFH2 header, which must immediately precede the body of the message.
- Anything else, such as the topic for a publish/subscribe message, is taken from the *first* MQRFH2 header.

Chapter 5. Publish/subscribe command messages

This chapter describes the command messages that are sent to MQSeries Integrator in a publish/subscribe application.

If you are using the Message Queue Interface (MQI) to write applications that use the publish/subscribe model, you need to understand these messages, and the header described in Chapter 4, “The MQRFH2 rules and formatting header” on page 29.

The following information is provided:

- “Delete Publication” on page 40
- “Deregister Subscriber” on page 42
- “Publish” on page 45
- “Register Subscriber” on page 48
- “Request Update” on page 51
- “Broker Response” on page 53
- “Message descriptor” on page 55
- “Reason codes” on page 58

The commands are contained in a <pssc> folder in the *NameValueData* field of the MQRFH2 header.

The message that can be sent by a broker in response to a command message is contained in a <psscr> folder.

The command descriptions list the properties that can be contained in a folder. Unless otherwise specified, the properties are optional and can occur at most once.

- *Names* of properties are shown thus: <Command>
- *Values* must be in string format, for example: Publish
- String constants representing property values are shown in parentheses, for example (MQPSC_PUBLISH)

The commands are listed in this chapter in alphabetic order.

Delete Publication

The **Delete Publication** command message is sent to a broker from a publisher, or another broker, to tell it to delete any retained publications for the specified topics.

This message is sent to the input queue of a message flow containing a **Publication** node. Authority to put a message to this queue, and to publish on the specified topic or topics, is required.

The input queue should be the same one that the original publication was sent to.

If the user has authority on some (but not all) topics, those that can be deleted will be and a warning response will indicate those that are not deleted.

See page 55 for details of the message descriptor (MQMD) parameters needed when sending a command message to the broker.

If a **Publish** command contained more than one topic, a **Delete Publication** command matching some but not all of those topics deletes only the publications for the topics specified.

Properties

<Command> (MQPSC_COMMAND)

The value is DeletePub (MQPSC_DELETE_PUBLICATION). This property is required, and must be present.

<Topic> (MQPSC_TOPIC)

The value is a string containing a topic for which retained publications are to be deleted. This can include wildcards to cause publications on several topics to be deleted.

This property is required, and can optionally be repeated for as many topics as needed.

<DeleteOpt> (MQPSC_DELETE_OPTION)

The delete options property can take the following value:

Local (MQPSC_LOCAL)

All retained publications for the specified topics are deleted at the local broker (that is, the broker to which this message is sent), whether they were published with the Local option or not. Publications at the other brokers are not affected.

None (MQPSC_NONE)

All options take their default values. This has the same effect as omitting the delete options property. If other options are specified at the same time, None is ignored.

The default if this property is omitted is that all retained publications for the specified topics are deleted at all brokers in the network, whether they were published with the Local option or not.

Example

Here is an example of *NameValueData* for a **Delete Publication** command message. This is used by the sample application to delete, at the local broker, the retained publication that contains the latest score in the match between Team1 and Team2.

```
<psc>  
  <Command>DeletePub</Command>  
  <Topic>Sport/Soccer/State/LatestScore/Team1 Team2</Topic>  
  <DelOpt>Local</DelOpt>  
</psc>
```

Deregister Subscriber

The **Deregister Subscriber** command message is sent to a broker from a subscriber, or another application on a subscriber's behalf, to indicate that it no longer wishes to receive messages matching the given parameters.

This message is sent to SYSTEM.BROKER.CONTROL.QUEUE, the broker's control queue. Authority to put a message to this queue is required.

See page 55 for details of the message descriptor (MQMD) parameters needed when sending a command message to the broker.

An individual subscription can be deregistered by specifying the corresponding topic, subscription point and filter values to the original subscription. If any of the values were not specified (that is, they took the default values) in the original subscription, they should be omitted in the subscription deregistration.

Alternatively, all subscriptions for a subscriber, or a group of subscriptions, can be deregistered using the DeregAll option. For example, if DeregAll is specified, together with a subscription point (but no topic or filter), then all subscriptions for the subscriber on the specified subscription point are deregistered, regardless of the topic and filter. Any combination of topic, filter and subscription point is allowed (if all three are specified only one subscription can match, so DeregAll is ignored).

The message must be sent by the subscriber that registered the subscription in the first place (determined by the subscriber's user ID).

Subscriptions can also be deregistered by a system administrator (see *MQSeries Integrator Using the Control Center*). If the subscriber queue is a temporary dynamic queue and the queue is deleted, or if the subscription expires, the broker will deregister the subscription automatically.

Properties

<Command> (MQPSC_COMMAND)

The value is DeregSub (MQPSC_DEREGISTER_SUBSCRIBER). This property is required, and must be present.

<Topic> (MQPSC_TOPIC)

The value is a string containing the topic to be deregistered.

This property can be omitted if DeregAll is specified in <RegOpt>. It can, optionally, be repeated if multiple topics are to be deregistered.

The topics specified can be a subset of those registered if the subscriber wishes to retain subscriptions for other topics. Wildcards are allowed, but a topic string containing wildcards must exactly match the corresponding string that was specified in the **Register Subscriber** command message.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is a string specifying the subscription point from which the subscription is to be detached.

This property can be omitted if Topic is specified or if DeregAll is specified in <RegOpt>. Only one SubPoint is allowed at any one time. If you omit this parameter, the following happen:

- If you specify `DeregAll`, every `SubPoint` is deregistered.
- If you do **not** specify `DeregAll`, only the default `SubPoint` is deregistered.

If the `<SubPoint>` property is omitted, and `DeregAll` is specified, subscriptions matching the `<Topic>` or `<Filter>` properties on all subscription points are deregistered. (Note that you cannot specify the default subscription point explicitly, so there is no way of deregistering all subscriptions from this subscription point only.)

`<Filter>` (MQPSC_FILTER)

The value is a string specifying the filter to be deregistered. It must match exactly (including case and spaces) a subscription filter previously registered.

The filters specified can be a subset of those registered if the subscriber wishes to retain subscriptions for other filters.

This property can be omitted if `Topic` is specified or if `DeregAll` is specified in `<RegOpt>`. It can optionally be repeated if multiple filters are to be deregistered.

If you do **not** specify `DeregAll`:

- You must specify one or more `Topic` values
- `SubPoint` and `Filter` parameters are optional
- `SubPoint` must not be repeated multiple times
- `Filter` can be repeated, if required

If you **do** specify `DeregAll`:

- `SubPoint`,
- `SubPoint`, and `Filter` can be omitted
- `Topic` and `Filter` can be repeated multiple times
- `SubPoint` can be specified only once

`<RegOpt>` (MQPSC_REGISTRATION_OPTION)

The registration options property can take the following values:

`DeregAll` (MQPSC_DEREGISTER_ALL)

All matching subscriptions registered for this subscriber are to be deregistered. If this option is set, the `<Topic>`, `<Filter>` and `<SubPoint>` properties can be omitted.

`CorrelAsId` (MQPSC_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor (MQMD), which must not be zero, is used to identify the subscriber. It must match the *CorrelId* used in the original subscription.

`None` (MQPSC_NONE)

All options take their default values. This has the same effect as omitting the registration options property. If other options are specified at the same time, `None` is ignored.

The default, if this property is omitted, is that no registration options are set.

`<QMgrName>` (MQPSC_Q_MGR_NAME)

The value is the queue manager name for the subscriber queue. It must match the *QMgrName* used in the original subscription.

Deregister Subscriber

If this property is omitted, the default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the broker's queue manager name.

<QName> (MQPSC_Q_NAME)

The value is the queue name for the subscriber queue. It must match the *QName* used in the original subscription.

If this property is omitted, the default is the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

Example

Here is an example of *NameValueData* for a **Deregister Subscriber** command message. In this case the sample application is deregistering its subscription to the topics which contain the latest score for all matches. The subscriber's identity, including the *CorrelId*, is taken from the defaults in the MQMD.

```
<psc>
  <Command>DeregSub</Command>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```

When a subscriber application sends a message to deregister a subscription, and receives a response message to say that this was processed successfully, it is possible that some publications will subsequently reach the subscriber queue if they were being processed by the broker at the same time as the deregistration. If the messages are not removed from the queue, there could be a buildup of unprocessed messages on the subscriber queue. We recommend that your application does a loop that includes an MQGET call with the appropriate *CorrelId* after sleeping for a while, to clear any such messages off the queue.

Also, if the subscriber uses a permanent dynamic queue and, when completing, it deregisters and closes the queue with the MQCO_DELETE_PURGE option on an MQCLOSE call, it is possible that the queue will not be empty. This is because publications from the broker might not yet be committed at the time that queue was deleted. In this case, an MQRC_Q_NOT_EMPTY return code will be issued by the MQCLOSE call. The application can avoid this problem by sleeping and reissuing the MQCLOSE call from time to time.

Publish

The **Publish** command message is sent:

- From a publisher to a broker, or
- From a broker to a subscriber

to publish information on a specified topic or topics.

This message is sent by a publisher to the input queue of a message flow that contains a **Publication** node. Authority to put a message to this queue, and to publish on the specified topic or topics, is required.

If the user has authority on some (but not all) topics, those that can be published will be and a warning response will indicate those that are not published.

If a subscriber has any matching subscriptions, the broker forwards the **Publish** message to the subscriber queues defined in the corresponding **Register Subscriber** command messages.

See page 55 for details of the message descriptor (MQMD) parameters needed when sending a command message to the broker, and used when a broker forwards a publication to a subscriber.

The broker forwards the **Publish** message to other brokers in the network that have matching subscriptions (unless it is a local publication).

Publication data (if any) is included in the message body. The data may be described in an <mc> folder in the *NameValueData* field of the MQRFH2 header.

Properties

<Command> (MQPSC_COMMAND)

The value is Publish (MQPSC_PUBLISH). This property is required, and must be present.

<Topic> (MQPSC_TOPIC)

The value is a string containing a topic that categorizes this publication. No wildcards are allowed.

This property is required, and can optionally be repeated for as many topics as needed.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The subscription point on which the publication is published.

This property should not be included in a publication message sent to the broker but will be added automatically to publication messages by the broker before those messages are sent to any appropriate subscribers. The value of the <SubPoint> property will be the value of the Subscription Point attribute of the Publication node that is handling the publishing.

<PubOpt> (MQPSC_PUBLICATION_OPTION)

The publication options property can take the following values:

RetainPub (MQPSC_RETAIN_PUB)

The broker is to retain a copy of the publication. If this option is not set, the publication is deleted as soon as the broker has sent the publication to all of its current subscribers.

IsRetainedPub (MQPSC_IS_RETAINED_PUB)

(Can only be set by a broker.) This publication has been retained by the broker. The broker sets this option to notify a subscriber that this publication was published earlier and has been retained, provided that the subscription has been registered with the `InformIfRetained` option. It is set only in response to a **Register Subscriber** or **Request Update** command message. Retained publications that are sent directly to subscribers do not have this option set.

Local (MQPSC_LOCAL)

This option tells the broker that this publication should not be propagated to other brokers. All subscribers that registered at this broker will receive this publication if they have matching subscriptions.

OtherSubsOnly (MQPSC_OTHER_SUBS_ONLY)

This option allows simpler processing of conference-type applications, where a publisher is also a subscriber to the same topic. It tells the broker not to send the publication to the publisher's subscriber queue even if it has a matching subscription. (The publisher's subscriber queue consists of its *QMgrName*, *QName*, and optional *CorrelId*, as described below.)

CorrelAsId (MQPSC_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the MQMD (which must not be zero) is part of the publisher's subscriber queue, in applications where the publisher is also a subscriber (see `OtherSubsOnly`).

None (MQPSC_NONE)

All options take their default values. This has the same effect as omitting the publication options property. If other options are specified at the same time, None is ignored.

The default, if this property is omitted, is that no publication options are set.

<PubTime> (MQPSC_PUBLISH_TIMESTAMP)

The value is an optional publication timestamp set by the publisher. It is of length 16 characters in the format:

YYYYMMDDHHMSSSTH

using Universal Time. However, this is not checked by the broker, which merely transmits this information to subscribers if it is present.

<SeqNum> (MQPSC_SEQUENCE_NUMBER)

The value is an optional sequence number set by the publisher.

It should increase by 1 with each publication. However, this is not checked by the broker, which merely transmits this information to subscribers if it is present.

If publications on the same topic are published to different interconnected brokers, it is the responsibility of the publishers to ensure that sequence numbers, if used, are meaningful.

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is a string containing the queue manager name for the publisher's subscriber queue, in applications where the publisher is also a subscriber (see `OtherSubsOnly`).

If this property is omitted, the default is the `ReplyToQMgr` name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the broker's queue manager name.

<QName> (MQPSC_Q_NAME)

The value is a string containing the queue name for the publisher's subscriber queue, in applications where the publisher is also a subscriber (see `OtherSubsOnly`).

If this property is omitted, the default is the `ReplyToQ` name in the message descriptor (MQMD), which (if `OtherSubsOnly` is set) must not be blank.

Example

Here are some examples of *NameValueData* for a **Publish** command message. The first example is for a publication sent by the match simulator in the sample application to indicate that a match has started.

```
<psc>
  <Command>Publish</Command>
  <Topic>Sport/Soccer/Event/MatchStarted</Topic>
</psc>
```

The second example is for a retained publication. In this case the results service is publishing the latest score in the match between Team1 and Team2.

```
<psc>
  <Command>Publish</Command>
  <PubOpt>RetainPub</PubOpt>
  <Topic>Sport/Soccer/State/LatestScore/Team1 Team2</Topic>
</psc>
```

Register Subscriber

The **Register Subscriber** command message is sent to a broker by a subscriber, or another application on a subscriber's behalf, to indicate that it wishes to subscribe to one or more topics at a subscription point. A message content filter can also be specified.

This message is sent to SYSTEM.BROKER.CONTROL.QUEUE, the broker's control queue. Authority to put a message to this queue is required, in addition to access authority for the topic or topics in the subscription (set by the broker's system administrator).

If the user has authority on some (but not all) topics, those that can be registered will be and a warning response will indicate those that are not registered.

See page 55 for details of the message descriptor (MQMD) parameters needed when sending a command message to the broker.

If the queue is a temporary dynamic queue, the subscription will be deregistered automatically by the broker when the queue is closed.

Properties

<Command> (MQPSC_COMMAND)

The value is RegSub (MQPSC_REGISTER_SUBSCRIBER). This property is required, and must be present.

<Topic> (MQPSC_TOPIC)

The topic for which the subscriber wants to receive publications. Wildcards are allowed (see "Topics and wildcards" on page 17).

This property is required, and can optionally be repeated for as many topics as needed.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is the subscription point to which the subscription is attached.

If this property is omitted, the default subscription point is used.

<Filter> (MQPSC_FILTER)

The value is an SQL expression that is used as a filter on the contents of publication messages (see "Filters" on page 18). If a publication on the specified topic matches the filter, it is sent to the subscriber.

If this property is omitted, no content filtering takes place.

<RegOpt> (MQPSC_REGISTRATION_OPTION)

The registration options property can take the following values:

Local (MQPSC_LOCAL)

The subscription is local and is not distributed to other brokers in the network. Publications made at other brokers will not be delivered to this subscriber, unless it also has a corresponding global subscription.

NewPubsOnly (MQPSC_NEW_PUBS_ONLY)

Retained publications that exist at the time the subscription is registered are not sent to the subscriber, only new publications.

If a subscriber re-registers and changes this option so that it is no longer set, it is possible that a publication that has already been sent to it will be sent again.

`PubOnReqOnly` (MQPSC_PUB_ON_REQUEST_ONLY)

The broker does not send publications to the subscriber, except in response to a **Request Update** command message.

`InformIfRet` (MQPSC_INFORM_IF_RETAINED)

The broker will inform the subscriber if a publication is retained when it sends a **Publish** message in response to a **Register Subscriber** or **Request Update** command message. The broker does this by including the `IsRetainedPub` publication option in the message.

`CorrelAsId` (MQPSC_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor (MQMD), which must not be zero, is used when sending matching publications to the subscriber queue.

`Pers` (MQPSC_PERSISTENT)

Publications matching this subscription are delivered to the subscriber as persistent messages.

`NonPers` (MQPSC_NON_PERSISTENT)

Publications matching this subscription are delivered to the subscriber as non-persistent messages.

`PersAsPub` (MQPSC_PERSISTENT_AS_PUBLISH)

Publications matching this subscription are delivered to the subscriber with the persistence specified by the publisher. This is the default behavior.

`PersAsQueue` (MQPSC_PERSISTENT_AS_Q)

Publications matching this subscription are delivered to the subscriber with the persistence specified on the subscriber queue.

`None` (MQPSC_NONE)

All registration options take their default values.

If the subscriber is already registered, its options are reset to their default values (this is *not* the same effect as omitting the registration options property), and the subscription expiry is updated from the MQMD of the **Register Subscriber** message.

If other registration options are specified at the same time, `None` is ignored.

If the registration options property is omitted and the subscriber is already registered, its registration options are unchanged and the subscription expiry is updated from the MQMD of the **Register Subscriber** message.

If the subscriber is not already registered, a new subscription is created with all registration options taking their default values.

The default values are `PersAsPub` and no other options set.

Register Subscriber

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is the queue manager name for the subscriber queue, to which matching publications are sent by the broker.

If this property is omitted, the default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the broker's *QMgrName*.

<QName> (MQPSC_Q_NAME)

The value is the queue name for the subscriber queue, to which matching publications are sent by the broker.

If this property is omitted, the default is the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

If the queue is a temporary dynamic queue, non-persistent delivery of publications (*NonPers*) must be specified in the <RegOpt> property.

If the queue is a temporary dynamic queue, the subscription will be deregistered automatically by the broker when the queue is closed.

Example

Here is an example of *NameValueData* for a **Register Subscriber** command message. In the sample application, the results service uses this message to register a subscription to the topics containing the latest scores in all matches, with the 'Publish on Request Only' option set. The subscriber's identity, including the *CorrelId*, is taken from the defaults in the MQMD.

```
<psc>
  <Command>RegSub</Command>
  <RegOpt>PubOnReqOnly</RegOpt>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```

Request Update

The **Request Update** command message is sent from a subscriber to a broker, to request the current retained publications for the specified topic and subscription point that match the given (optional) filter.

This message is sent to `SYSTEM.BROKER.CONTROL.QUEUE`, the broker's control queue. Authority to put a message to this queue is required, in addition to access authority for the topic in the request update (set by the broker's system administrator).

See page 55 for details of the message descriptor (MQMD) parameters needed when sending a command message to the broker.

This command is normally used if the subscriber specified the option `PubOnReqOnly` (publish on request only) when it registered. If the broker has matching retained publications, they are sent to the subscriber. If not, the request fails (with an `MQRCCF_NO_RETAINED_MSG`). The requester must have previously registered a subscription with the same `Topic`, `SubPoint`, and `Filter` values.

Properties

<Command> (MQPSC_COMMAND)

The value is `ReqUpdate` (MQPSC_REQUEST_UPDATE). This property is required, and must be present.

<Topic> (MQPSC_TOPIC)

The value is the topic the subscriber is requesting. Wildcards are allowed (see "Topics and wildcards" on page 17).

This property is required, but only one occurrence is allowed in this message.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is the subscription point to which the subscription is attached.

If this property is omitted, the default subscription point is used.

<Filter> (MQPSC_FILTER)

The value is an SQL expression that is used as a filter on the contents of publication messages (see "Filters" on page 18). If a publication on the specified topic matches the filter, it is sent to the subscriber.

The <Filter> property should have the same value as that specified on the original subscription for which you are now requesting an update.

If this property is omitted, no content filtering takes place.

<RegOpt> (MQPSC_REGISTRATION_OPTION)

The registration options property can take the following value:

`CorrelAsId` (MQPSC_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor (MQMD), which must not be zero, is used when sending matching publications to the subscriber queue.

Request Update

None (MQPSC_NONE)

All options take their default values. This has the same effect as omitting the registration options property. If other options are specified at the same time, None is ignored.

The default, if this property is omitted, is that no registration options are set.

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is the queue manager name for the subscriber queue, to which the matching retained publication is sent by the broker.

If this property is omitted, the default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the broker's *QMgrName*.

<QName> (MQPSC_Q_NAME)

The value is the queue name for the subscriber queue, to which the matching retained publication is sent by the broker.

If this property is omitted, the default is the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

Example

Here is an example of *NameValueData* for a **Request Update** command message. In the sample application, the results service uses this message to request retained publications containing the latest scores for all teams. The subscriber's identity, including the *CorrelId*, is taken from the defaults in the MQMD.

```
<psc>
  <Command>ReqUpdate</Command>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```


Broker Response

A **Broker Response** message is sent from a broker to the *ReplyToQ* of a publisher or a subscriber, to indicate the success or failure of a command message received by the broker if the command message descriptor specified that a response is required.

See “MQMD for broker response messages” on page 57 for details of the message descriptor (MQMD) parameters used when a broker sends a response to a publisher or a subscriber.

The broker response message is contained within the *NameValueData* field of the MQRFH2 header, in a `<pscr>` folder.

In the case of a warning or error, the response message contains the `<pscr>` folder from the command message in addition to the `<pscr>` folder. The message data (if any) is not contained in the broker response message. None of the message that caused an error will be processed but a warning indicates that some of the message may have been processed successfully.

Properties

`<Completion>` (MQPSCR_COMPLETION)

The completion code, which can take one of three values:

ok	Command completed successfully
warning	Command completed with warning
error	Command failed

`<Response>` (MQPSCR_RESPONSE)

The response to a command message, if that command produced a completion code of warning or error. It contains a `<Reason>` property, and might contain other properties indicating the cause of the warning or error.

In the case of one or more errors, there will be a single response folder, indicating the cause of the first error only. In the case of one or more warnings, there will be a Response folder for each warning.

`<Reason>` (MQPSCR_REASON)

The reason code qualifying the completion code, if the completion code is warning or error. It is set to one of the error codes listed on page 58. The `<Reason>` property is contained within a `<Response>` folder. The reason code may be followed by any valid property from the `<pscr>` folder (for example, a topic name), indicating the cause of the error or warning.

Examples

Here are some examples of *NameValueData* in a **Broker Response** message. A successful response will be as follows:

```
<pscr>
  <Completion>ok</Completion>
</pscr>
```

Here is an example of a failure response (due to a filter error). The first *NameValueData* string contains the response; the second contains the original command.

Broker Response

```
<pscr>
  <Completion>error</Completion>
  <Response>
    <Reason>3150</Reason>
  </Reponse>
</pscr>
```

```
<psc>
  ...
  command message (to which
  the broker is responding)
  ...
</psc>
```

Here is an example of a warning response (due to unauthorized topics). The first NameValueData string contains the response; the second contains the original command.

```
<pscr>
  <Completion>warning</Completion>
  <Response>
    <Reason>3081</Reason>
    <Topic>topic1</Topic>
  </Reponse>
  <Response>
    <Reason>3081</Reason>
    <Topic>topic2</Topic>
  </Reponse>
</pscr>
```

```
<psc>
  ...
  command message (to which
  the broker is responding)
  ...
</psc>
```

If there is a failure while attempting to send a response:

- MQSeries Integrator tries to send the response to the MQSeries dead-letter queue if the MQPUT fails.
- If the response cannot be sent to the dead-letter queue, an error is logged and the command message is normally rolled back. This depends on how the MQInput node has been configured.

Message descriptor

The MQSeries message descriptor (MQMD) is fully documented in the *MQSeries Application Programming Reference* book. This section summarizes the fields used by MQRFH2 publish/subscribe command and response messages.

MQMD for command messages

This section describes the message descriptor as set by applications that send command messages to the broker.

Fields that are left as the default value, or can be set to any valid value in the usual way, are not listed here.

Report

See *MsgType* and *CorrelId* (below).

MsgType

Can be set to MQMT_REQUEST for a command message if a response is always required. The MQRO_PAN and MQRO_NAN flags in the *Report* field are not significant in this case.

Can be set to MQMT_DATAGRAM, in which case responses depend on the setting of the MQRO_PAN and MQRO_NAN flags in the *Report* field:

- MQRO_PAN alone means that the broker is to send a response only if the command succeeds.
- MQRO_NAN alone means that the broker is to send a response only if the command fails.
- If a command completes with a warning, a response is sent if either MQRO_PAN or MQRO_NAN is set.
- MQRO_PAN + MQRO_NAN means that the broker is to send a response whether the command succeeds or fails. This has the same effect from the broker's perspective as setting *MsgType* to MQMT_REQUEST.
- If neither MQRO_PAN nor MQRO_NAN is set, no response will ever be sent.

Format

Set to MQFMT_RF_HEADER_2

MsgId

Normally set to MQMI_NONE, so that the queue manager generates a unique value.

CorrelId

Set to any value. If the sender's identity includes a *CorrelId*, specify this value, together with MQRO_PASS_CORREL_ID in the *Report* field, to ensure that it will be set in all response messages sent by the broker to the sender.

ReplyToQ

This is the queue to which responses, if any, are to be sent. This can be the sender's queue which has the advantage that the *QName* parameter can

Message descriptor

be omitted from the message. If, however, responses are to be sent to a different queue, the *QName* parameter will be needed.

ReplyToQMgr

Queue manager for responses. If you leave this field blank (the default value), the local queue manager puts its own name in this field.

MQMD for publications forwarded by a broker

This section describes the message descriptor for messages sent by the broker to subscribers.

The fields are set to their default values, except for the following:

Report

Will be set to MQRO_NONE.

MsgType

Will be set to MQMT_DATAGRAM.

Expiry

Will be set to the value in the **Publish** message received from the publisher. In the case of a retained message, the time outstanding is reduced by the approximate time the message has been at the broker.

Format

Will be set to MQFMT_RF_HEADER_2

MsgId

Will be set to a unique value.

CorrelId

If *CorrelId* is part of the subscriber's identity, this is the value specified by the subscriber when registering. Otherwise, it is a non-zero value chosen by the broker.

Priority

As set by the publisher (or as resolved if the publisher specified MQPRI_PRIORITY_AS_Q_DEF).

Persistence

As set by the publisher (or as resolved if the publisher specified MQPER_PERSISTENCE_AS_Q_DEF), unless specified otherwise in the **Register Subscriber** message for the subscriber to which this publication is being sent.

ReplyToQ

Will be set to blanks.

ReplyToQMgr

Broker's queue manager name.

UserIdentifier

Subscriber's user identifier (as set when the subscriber registered).

AccountingToken

Subscriber's accounting token (as set when the subscriber first registered).

ApplIdentityData

Subscriber's application identity data (as set when the subscriber first registered).

PutApplType

Will be set to MQAT_BROKER.

PutApplName

Will be set to the first 28 characters of the broker's queue manager name.

PutDate

Timestamp when the broker puts the message.

PutTime

Timestamp when the broker puts the message.

ApplOriginData

Will be set to blanks.

MQMD for broker response messages

This section describes the message descriptor for response messages sent by the broker.

The fields are set to their default values, except for the following:

Report Will be set to all zeroes.

MsgType Will be set to MQMT_REPLY.

Format Will be set to MQFMT_RF_HEADER_2

MsgId Will be set according to the *Report* options in the original command message. By default, this means that it is set to MQMI_NONE, so that the queue manager generates a unique value.

CorrelId Will be set according to the *Report* options in the original command message. By default, this means that the *CorrelId* is set to the same value as the *MsgId* of the command message. This can be used to correlate commands with their responses.

Priority The same value as in the original command message.

Persistence The same value as in the original command message.

Expiry The same value as in the original command message received by the broker.

PutApplType Will be set to MQAT_BROKER.

PutApplName Will be set to the first 28 characters of the queue manager name.

Other context fields are set as if generated with MQPMO_PASS_IDENTITY_CONTEXT.

Reason codes

The following reason codes might be returned in the *Reason* field of a publish/subscribe response <psc> folder. Constants that can be used to represent these codes in the C or C++ programming languages are also given. The MQRC_ constants require the MQSeries cmqc.h header file. The MQRCCF_ constants require the MQSeries cmqcf.h header file (apart from MQRCCF_FILTER_ERROR and MQRCCF_WRONG_USER, which are specific to MQSeries Integrator Version 2 and require the MQSeries Integrator Version 2 BipRfh2.h header file).

Reason code and text	Explanation	Issued by
2336 MQRC_RFH_COMMAND_ERROR	Valid values for the <Command> field of a <psc> folder are: RegSub, DeregSub, Publish, DeletePub, and ReqUpdate. Any other values result in this error code being issued.	Any command
2337 MQRC_RFH_PARM_ERROR	The <psc> and <mcd> folders both have a set of valid parameters that can be specified within them. Check the descriptions of these folders and ensure that you have not specified incorrect parameters.	Any command
2338 MQRC_RFH_DUPLICATE_PARM	Some parameters (for example, Topic) within a <psc> folder can be repeated, while others (for example, Command) can not. Check that you have not duplicated a non-repeatable parameter.	Any command
2339 MQRC_RFH_PARM_MISSING	Some parameters within <psc> or <mcd> folders are optional and can be omitted; some are mandatory and must not be omitted. Check that you have included all mandatory parameters within your <psc> and <mcd> folders.	Any command
3072 MQRCCF_TOPIC_ERROR	One or more of the values you supplied for the Topic parameter are incorrect. Check that your values for Topic conform to the specified restrictions.	Any command
3073 MQRCCF_NOT_REGISTERED	The combination of SubPoint, Topic, and Filter that you specified on your DeregSub or ReqUpdate command was either not a combination with which you had previously registered or, for the DeregSub command if the DeregAll option was specified, one of the SubPoint, Topic, or Filter properties was not used to deregister any subscription.	Deregister Subscriber and Request Update commands
3074 MQRCCF_Q_MGR_NAME_ERROR	The specified queue manager was not valid or the queue manager was not available or did not exist.	Deregister Subscriber, Publish, Register Subscriber, and Request Update commands

Reason code and text	Explanation	Issued by
3076 MQRCCF_Q_NAME_ERROR	The specified queue name was not valid or the queue did not exist on the specified queue manager.	Deregister Subscriber, Publish, Register Subscriber, and Request Update commands
3077 MQRCCF_NO_RETAINED_MSG	There were no retained messages for the topic you specified. (This may or may not be an error, depending on the design of your application program.)	Request Update command
3079 MQRCCF_INCORRECT_Q	RegSub, DeregSub, and ReqUpdate commands are always sent to the SYSTEM.BROKER.CONTROL.QUEUE queue of the broker for which they are intended. Publish and Delete Publication commands are sent to the input queue for the particular publish/subscribe message flow for which they are intended (determined when the message flow is designed). This error code will be returned if a command is sent to the wrong queue.	Any command
3080 MQRCCF_CORREL_ID_ERROR	You have specified CorrelAsId as one of your RegOpt parameters. However, the CorrelId field of the MQMD does not contain a valid correlation identifier (that is, it is set to MQCI_NONE).	Deregister Subscriber and Register Subscriber commands
3081 MQRCCF_NOT_AUTHORIZED	You are not authorized to perform the requested action. Authorization settings for the broker are handled from the Control Center. Contact your system administrator.	Publish and Register Subscriber commands
3083 MQRCCF_REG_OPTIONS_ERROR	You have specified an unrecognized RegOpt parameter in the <psc> folder that contains your DeregSub command.	Deregister Subscriber and Register Subscriber commands
3084 MQRCCF_PUB_OPTIONS_ERROR	You have specified an unrecognized PubOpt parameter in the <psc> folder that contains your Publish command.	Publish command
3087 MQRCCF_DEL_OPTIONS_ERROR	You have specified an unrecognized DelOpt parameter in the <psc> folder that contains your DeletePub command.	Delete Publication command
3150 MQRCCF_FILTER_ERROR	The value specified for the Filter parameter is not valid. Check the section that describes valid syntax for filter expressions and ensure that your expression conforms.	Deregister Subscriber, Register Subscriber, and Request Update commands

Reason codes

Reason code and text	Explanation	Issued by
3151 MQRCCF_WRONG_USER	A subscription that matches the one specified already exists; however, it was registered by a different user. A subscription can only be changed or deregistered by the same user who originally registered it.	Deregister Subscriber, Register Subscriber, and Request Update commands

Part 2. Programming a plug-in node or parser

This part contains:

- Chapter 6, “Implementing a plug-in node or parser” on page 63
- Chapter 7, “Node implementation and utility functions” on page 103
- Chapter 8, “Parser implementation and utility functions” on page 135

Chapter 6. Implementing a plug-in node or parser

The function of the MQSeries Integrator Version 2.0 message broker can be enhanced by:

- Providing additional message processing nodes to perform a variety of tasks, either superseding existing function or introducing new function. Plug-in nodes can provide generic function that is of value in a wide range of environments, or highly specialized function tailored and customized to a specific environment.
- Providing 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.

This chapter contains:

- “Introduction” on page 64
- “Implementing a message processing node” on page 66
- “Implementing a message parser” on page 87
- “Accessing the message content” on page 89
- “General development considerations” on page 95
- “Errors and exception handling” on page 98

+
+
+
+
References are made throughout this chapter to the MQSeries Integrator home directory: that is, the directory into which MQSeries Integrator is installed. The default is C:\Program Files\IBM MQSeries Integrator 2.0. The label <mqs_i_root> is used to refer to this home directory.

Introduction

This section contains an overview of how to implement a message processing node or parser to enhance the functionality of MQSeries Integrator.

A broker extension, or *plug-in*, is written in C and distributed as a dynamic link library (DLL) on Windows NT. The DLL must be named with a filetype of '.lib' (loadable implementation library).

If the plug-in implements a message processing node, it must also have a *signature* defined as an XML representation in a file. This signature enables the MQSeries Integrator Control Center to display a representation of the plug-in on its palette of message processing nodes, and for these nodes to be wired correctly in the design pane. If a node requires more sophisticated customization than setting property values using a standard property editor, a Java implementation of a customizer can be provided. National language considerations can also be addressed by providing resource property files with locale-specific renderings of the node's attribute names and description.

A plug-in implements a node or parser factory that can support multiple nodes or parser types.

The plug-in is loaded when the broker is initialized. Registration functions in the plug-in are invoked so that the broker understands what nodes or parsers are supported by the plug-in.

The C interfaces consist of:

1. A set of *implementation functions* (or *callback functions*), which provide the functionality of the plug-in. These implementation functions must be written by the developer, and are invoked by the message broker.
2. A set of *utility functions*, the purpose of which is to create or manipulate resources in the message broker or to request a service of the broker. These utility functions are intended to be invoked by the plug-in.

Header files

The C interfaces are defined by the following header files, which are distributed with MQSeries Integrator:

BipCni.h	Message processing nodes
BipCpi.h	Message parsers
BipCci.h	Interfaces common to both nodes and parsers
BipCos.h	Platform specific definitions

+

Sample code

Sample code is provided with the product, to help you understand how to write nodes and parsers. The samples are located in the following directory:

```
<mqs_i_root>\examples\plugin
```

The file `BipSampPluginNode.c` contains sample implementations of:

- A message processing node that routes a message to one of five output terminals depending on content
- A simple fixed transformation of an input message into an output message

The file `BipSampPluginParser.c` contains an implementation of a simple pseudo-XML parser.

Installing a plug-in on the broker

The first step is to create the plug-in factory. Move to the directory where the plug-in code is located. For example, if you are creating a factory for the sample plug-in node:

```
cd <mqsi_root>\examples\plugin
```

Compile the plug-in node (assuming the Microsoft 32-bit C/C++ Compiler) as follows:

```
+ cl /VERBOSE /LD /MT /I..\plugin /I..\..\include\plugin BipSampPluginNode.c
+ -link /DLL ..\..\lib\imbdfp1g.lib
```

Then create the 'lil':

```
rename BipSampPluginNode.dll BipSampPluginNode.lil
```

Having created the factory DLL, and renamed it with a file extension of 'lil', the file can be installed by copying or moving it to the following directory:

```
<mqsi_root>\bin
```

This directory (or its equivalent if you chose to override the default location) is created during installation of MQSeries Integrator Version 2.0.

```
+ You must copy the 'lil' file to each system that requires the functionality of the
+ plug-in node or parser that you have created.
```

```
+ You must stop and restart each broker to enable it to detect the existence of the
+ new 'lil'.
```

To remove or uninstall a 'lil', you must shutdown the broker first. The file can then be removed. The uninstall procedure for MQSeries Integrator will *not* remove any files that were not installed by the MQSeries Integrator installation procedure. This means that a third-party 'lil' must be removed with an uninstall program supplied by that third-party, or removed manually if necessary.

```
+ In addition to installing the code for your message processing node on the broker,
+ you must install the interface definition for the node in the Control Center. This task
+ is described in detail in "Integrate the message processing node into the Control
+ Center" on page 68.
```

Implementing a message processing node

The following tasks must be performed when implementing a message processing node:

1. Determine the configuration attributes required by the node
2. Develop a context creation function
3. Develop the node attribute functions
4. Develop a function to perform the actual processing of the node
5. Write a plug-in initialization function
6. Build an output message (optional)
7. Integrate the new node into the Control Center

Determine configuration attributes

The first step is to decide what configuration attributes the node requires, and what the allowable values are for each attribute. There is no limit to the number of configuration attributes that a node can have. However, they must not conflict with the base configuration attributes that are inherited by all message processing nodes. These are:

- label
- userTraceLevel
- traceLevel
- userTraceFilter
- traceFilter

Develop a context creation function

When an instance of a plug-in node object is created, the context creation implementation function **cniCreateNodeContext** is invoked by the message broker. This allows the plug-in node to allocate instance data associated with the node, such as data areas for attributes.

A message flow node has a number of input terminals and output terminals associated with it. Simple nodes have one input terminal, and either zero or a fixed number of output terminals. The utility functions **cniCreateInputTerminal** and **cniCreateOutputTerminal** are used to add terminals to a node when the node is instantiated. They must be invoked within the **cniCreateNodeContext** implementation function. If a plug-in attempts to create a terminal at any other time, the results will be unpredictable.

Note that, by convention within the broker, the supplied message processing nodes that have a terminal named 'failure' behave in a particular way. If the failure terminal is not attached to a connector to another node and a processing failure occurs, an exception is thrown to terminate the message flow. It is recommended that a plug-in node supports a failure terminal where appropriate that respects this convention. To provide similar behavior, you can use **cnilsTerminalAttached** to check if the named terminal is attached before attempting to propagate a message to it.

Develop attribute functions

The message flow node implementation must respond to requests to get and set its configuration attributes. Functions which assign values should perform appropriate verification. Attribute values are passed between the broker and the plug-in using wide character strings.

A request to set a configuration attribute causes **cniSetAttribute** to be invoked. This function receives the attribute name and attribute value as parameters. The broker ensures that no messages are being processed while this function is called. That is, there are no thread issues to deal with between the updating of a configuration attribute and the referencing of the attribute by the evaluate function. **cniGetAttributeName** allows the node to describe its configuration attributes.

The sample message flow node implementation gives an example of how these functions can be implemented.

Develop the node processing function

The main processing of a message flow node is performed inside the **cniEvaluate** implementation function. A message can be passed to other connected nodes by invoking **cniPropagate** on an output terminal of the node. The message broker runs on multiple threads, and it is possible that two threads might be executing **cniEvaluate** on the same node object at the same time to process different messages. Therefore the code contained within this implementation function, and any that it calls, must be thread safe and fully re-entrant.

It is advisable not to use extra threads in the implementation of a node, and it is essential that any calls to **cniPropagate** are performed on the same thread that **cniEvaluate** was called on.

Develop a plug-in initialization function

This is invoked when the plug-in is loaded during message broker initialization. The plug-in is responsible for:

- Creating and naming the node factory that is implemented by the plug-in. The node factory is simply a container for related node implementations. Node factory names must be unique within a broker.
- Defining the name of each node, and supplying a pointer to a virtual function table that contains pointers to the plug-in implementation functions. Node names must be unique within a broker.

The plug-in initialization function defines the name of the factory that the plug-in supports and the classes of objects supported by the factory. Each DLL (or .lib) that implements a plug-in must export a function called **bipGetMessageflowNodeFactory** as its initialization function.

The initialization function must create the factory object and define the names of all nodes supported by the plug-in. A factory can support any number of object classes (nodes). When a node is defined, a list of pointers to the implementation functions for that node is passed to the broker. If a node of the same name already exists, the request is rejected. The plug-in initialization function is invoked automatically during broker initialization.

Implementing a node

The initialization function must create a node factory by invoking **cniCreateNodeFactory**. The node classes supported by the factory are defined by calling **cniDefineNodeClass**. The address of the factory object (returned by **cniCreateNodeFactory**) must be returned to the broker as the return value from the initialization function.

The broker will ensure that all message flows that have nodes of the relevant class are quiesced while this function is called.

Build an output message (optional)

A plug-in message processing node might need to create an output message; for example, to derive a new message based on the content of the existing input message. The **cniCreateMessage** function is used to do this.

When an output message is created, a root element is created automatically as the root of the (initially empty) syntax element tree. Syntax elements can then be added as required to the element tree, using the root element as the initial insertion point.

When an element is created an *owning parser* can be specified, determined by whether the function that accepts a parser class name is used (for example, **cniCreateElementBeforeUsingParser** instead of **cniCreateElementBefore**). When the message is propagated to an **MQOutput** node, the broker instructs the parsers owning the elements at the first generation (that is, immediate children of the root element) to serialize their part of the element tree to the output message buffer. If the parser is implemented as a plug-in, this causes the broker to invoke the implementation function **cniWriteBuffer**. If an element at the first generation is created so that it is not owned by a parser (using **cniCreateElementBefore**, for example), it is owned by a notional *root parser*. This means that this branch of the element tree will not be serialized to the output message buffer. This feature could be used, for example, to store temporary elements that persist for the duration of the flow of that message through the message flow.

+ Integrate the message processing node into the Control Center

+ When you have created the code that provides the function of your new node, you
+ must define the node to MQSeries Integrator. You can then use it in the Control
+ Center, include it in a message flow, and deploy that message flow to one or more
+ brokers.

+ This section describes the following tasks:

- + • “Defining the interface of the node” on page 69.
- + • “Defining optional node resources” on page 77.
- + • “Installing a new message processing node in the Control Center” on page 84.

+ These tasks involve the creation of a number of files: some of these are required,
+ and others are optional. The files are:

- + • The XML interface definition file (required)
- + • The WDP file (required)
- + • The icon files (optional)
- + • The help file (optional)
- + • The properties file (optional)
- + • A property editor (optional)
- + • A customizer (optional)

These tasks, and the files created to complete them, are illustrated using the sample plug-in node shipped with MQSeries Integrator. A set of files for this sample plug-in node is supplied, and you can copy and update these files for your own new node, or you can create your own files. The files are in directory `<mqsi_root>\examples\plugin`.

Note: The file locations used within the example files use the forward slash (/) character to identify directory structure. You are recommended to use this character in the files you create. However, on the Windows NT system the use of the backslash character (\) is also supported.

Note: The sample files provided for the plug-in node do not conform to the recommendations documented here. In particular:

- package is set to `com.isv` in several files.
- `displayname` in the sample WDP file is incorrect.

However, the information here is consistent with the values and names used in the samples supplied, and provides a working example. Where the names and values differ from the recommendations given, this is noted in the text.

Defining the interface of the node

This subtask is required. You must create two files to define the interface of the node. These are:

- The XML interface definition file.
- The WDP file.

Both these files are created in XML. You can use your favorite text editor to create new files, or to copy and update the samples supplied. You do not have to have any programming skills to complete this subtask. See “Storing the files in the MQSeries Integrator directory structure” on page 84 for the required location for these files.

If you have a good understanding of XML and DTD files, you can refer to the file `mqsi.dtd` that defines the rules of the XML used in these files. The file is stored in `<mqsi_root>\Tool`. The rules are summarized here to help you create files that provide the function you require.

The XML interface definition file:

The XML interface definition file defines the attributes of the node, and defines the interfaces that the node presents to other nodes within a message flow. The file name must match the name of the node, but **without** the suffix “Node”. For example, the name of this file for the sample provided is `ComIbmSampleTransform` (shown in Figure 6 on page 70). There is no file extension. You must use the same file name and an extension of `.wdp` for the WDP file for this node (see “The WDP file” on page 76).

When you create the XML interface definition file for your node, you are recommended to copy this sample. Update the contents of the file following the guidance given in this section. You must leave all other content unchanged.

```

+ <?xml version="1.0" encoding="UTF-8"?>
+ <!DOCTYPE MessageProcessingNodeType SYSTEM "mqsi.dtd" >
+ <MessageProcessingNodeType scaleableIcon=""
+ longDescription="longDescription" icon="images/SampleTransform.gif"
+ versionCreator="" package="com.isv" shortDescription="shortDescription"
+ version="" creationTimestamp="" isPrimitive="true" creator=""
+ versionTimestamp="" xmi.uuid="ComIbmSampleTransform"
+ xmi.label="SampleTransform" collectionPath="">
+ <Attribute xmi.label="nodeTraceOutfile" type="String"
+ value="" xmi.uuid="" valueMandatory="false" encoded="false"/>
+ <Attribute xmi.label="nodeTraceSetting" type="Integer"
+ value="0" xmi.uuid="" valueMandatory="false" encoded="false"/>
+ <OutTerminal longDescription="" icon="images/OutTerminal.gif"
+ versionCreator="" shortDescription="" creationTimestamp=""
+ creator="" y="0" x="0" versionTimestamp=""
+ xmi.uuid="0d06000c-0800-0000-0100-090507050000" xmi.label="failure">
+ <OutTerminalTypeRef xmi.label="OutTerminalType"
+ type="OutTerminalType" icon="images/OutTerminal.gif"
+ title="OutTerminal" refType="OutTerminalType"
+ href="OutTerminalType/OutTerminalType" xml:link="simple"/>
+ </OutTerminal>
+ <OutTerminal longDescription="" icon="images/OutTerminal.gif"
+ versionCreator="" shortDescription="" creationTimestamp=""
+ creator="" y="0" x="0" versionTimestamp=""
+ xmi.uuid="020c000c-0800-0000-0100-090507050000" xmi.label="out">
+ <OutTerminalTypeRef xmi.label="OutTerminalType" type="OutTerminalType"
+ icon="images/OutTerminal.gif" title="OutTerminal" refType="OutTerminalType"
+ href="OutTerminalType/OutTerminalType" xml:link="simple"/>
+ </OutTerminal>
+ <InTerminal longDescription="" icon="images/InTerminal.gif"
+ versionCreator="" shortDescription="" creationTimestamp=""
+ creator="" y="0" x="0" versionTimestamp=""
+ xmi.uuid="0505000c-0800-0000-0100-090507050000" xmi.label="in">
+ <InTerminalTypeRef xmi.label="InTerminalType" type="InTerminalType"
+ icon="images/InTerminal.gif" title="InTerminal" refType="InTerminalType"
+ href="InTerminalType/InTerminalType" xml:link="simple"/>
+ </InTerminal>
+ </MessageProcessingNodeType>

```

Figure 6. A sample plug-in XML interface definition file

The tags and their properties are described below:

- **MessageProcessingNodeType**

This tag describes the type of the new node. You must define one and only one tag of this type. If you define more than one, error BIP0056 is generated when you try to add the node to the workspace (see “Defining the node to the configuration repository” on page 85).

The tag properties are:

- longDescription

This field can provide either a textual description of the node, or a value that is used as a key into the node’s properties file (described in “Creating a properties file for the node” on page 80). If a value is provided, the textual description is looked up in the node’s properties file. If a node properties file does not exist, or the key does not exist within the properties file, the value of this property is used as the textual description and appears when the node’s properties are displayed.

The default behavior is for this property to be included on the *Description* tab when the node’s properties are displayed.

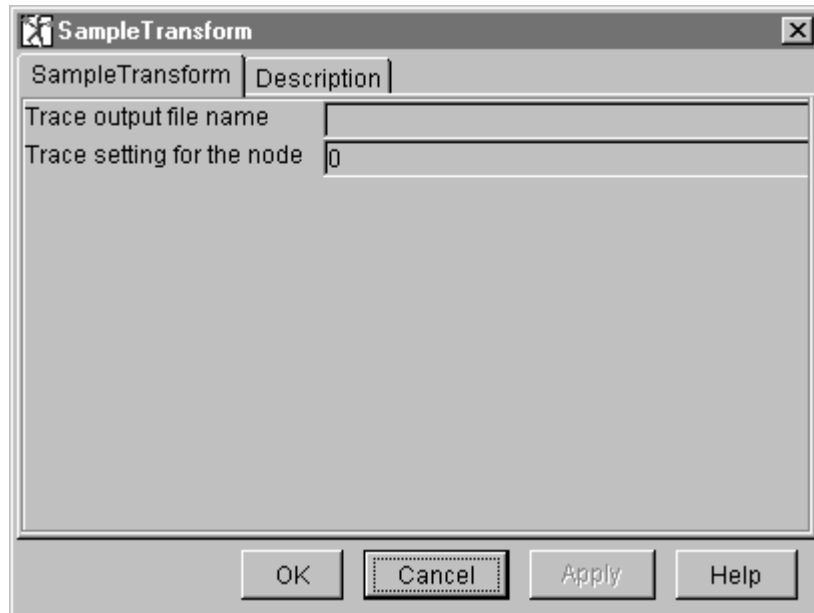
- + – icon
- + This field identifies the path (relative to `<mqsi_root>\Tool\`) and name of
- + the `.gif` file that provides the icon for the node. See “Defining an icon for
- + the node” on page 77 for further details about icons.
- + – package
- + The package attribute identifies where the resources for this node’s
- + customizer, property editor, and properties files are stored on the local file
- + system.
- + The following values are recommended, but not enforced:
- +
 - `com.ibm`. This is used for IBM supplied nodes.
 - `com.isv`. This is used for additional nodes supplied by ISVs and other
 - + companies. In the sample plug-in XML interface definition file supplied,
 - + this value is used instead of `com.ibm`. The value `com.isv` is assumed for
 - + this working example.
- + The value set in `package` defines the directory structure under the directory
- + `<mqsi_root>\Tool`. The period character (`.`) within `package` is replaced with
- + the backslash (`\`) character to form the full directory structure (for example,
- + `<mqsi_root>\Tool\com\isv`).
- + – shortDescription
- + This field can provide either a brief textual description of the node, or a
- + value that is used as a key into the node’s properties file. If a value is
- + provided, the textual description is looked up in the node’s properties file. If
- + a node properties file does not exist, or the key does not exist within the
- + properties file, the value of this property is used as the textual description.
- + The default behavior is for this property to be included on the *Description*
- + tab when the node’s properties are displayed.
- + – isPrimitive
- + This attribute must be set to “true”.
- + – xmi.uuid
- + This must be set to the full name of the node. The suffix “Node” is not part
- + of this attribute value. The identifier specified must be unique within your
- + broker domain, and therefore within your configuration repository. The
- + value you specify for this property can be a maximum of 36 characters in
- + length, and must be used as the file name for this interface file.
- + The following naming convention is recommended, but not enforced, for
- + this value:
- +
 - `ComIbm<nodename>` for all nodes provided by IBM.
 - `Com<YourCompanyName><nodename>` for the nodes you create.
- + Where `<nodename>` excludes the suffix “Node” in both cases.
- + The sample provided conforms to this standard and has `xmi.uuid` set to
- + `ComIbmSampleTransform`.
- + – xmi.label
- + This character field defines the displayed label (name) of the node. The
- + value you specify here must be the same as the value you specify for

Implementing a node

- + xmi.label in the WDP file (see “The WDP file” on page 76). This value is
- + used by the Control Center when this node is displayed, and defines the
- + name that must be used within the file name for the node icon, help,
- + properties, and Customizer files (if defined). The sample provided has
- + xmi.label set to SampleTransform.
- + – The following attributes have no meaning in the current implementation,
- + and must be left blank.
 - + - collectionPath
 - + - scaleableIcon
 - + - versionCreator
 - + - version
 - + - creationTimestamp
 - + - creator
 - + - versionTimestamp
- + • **Attribute**
- + This tag defines an attribute of the node. You can define zero or more
- + <Attribute> tags. The attributes can be specified individually using this tag
- + (illustrated in the SampleTransform interface definition file), or defined together
- + in one or more <AttributeGroup> tags (described below). You can use any
- + combination of <Attribute> and <AttributeGroup> tags to complete the node
- + definition. The attributes that are not defined to a group will appear on the
- + default properties tab when the node properties are displayed.
- + The <Attribute> tag has the following properties:
 - + – xmi.label
 - + This character field defines the displayed label (name) of the attribute. If a
 - + value is provided, it acts as a key into the node’s properties file. If there is
 - + no properties file, or the key does not exist within the properties file, the
 - + value of this field is used and its contents are displayed.
 - + – type
 - + This is the type of the property. It determines which property editor is
 - + invoked to handle the input for this attribute, and therefore how it is
 - + validated. It must be one of:
 - + - A built-in type (handled by the default property editors supplied):
 - + • Boolean
 - + • Double
 - + • Float
 - + • Integer
 - + • Long
 - + • String
 - + • Date
 - + • Time
 - + • Timestamp
 - + - A sequence of possible values "value1 value2 ..." (an enumeration).
 - + - The definition of a property editor called <type>PropertyEditor or
 - + <type>Editor.

- + – value
- + This is the value of the <Attribute>. It is displayed as the initial value for the
- + attribute.
- + If <Attribute> is an enumerated type, the value set is used as a key into the
- + properties file (described in “Creating a properties file for the node” on
- + page 80) and the translated value is displayed as the default value in the
- + drop-down list (that is, the first in the list). If there is no properties file, or
- + the key does not exist within the properties file, the value of this field is
- + used and its contents are displayed as the default value in the drop-down
- + list. If no value is set (that is, value=“”), the drop-down list appears in the
- + order in which it is coded (that is, the first value in the type property is
- + displayed as the default value in the drop-down list).
- + If <Attribute> is not an enumerated type, the value is displayed as entered
- + and is not used as a key into the properties file.
- + – xmi.uuid
- + This provides an identifier for this attribute that is used internally if this
- + attribute is promoted. Set this field to “”. See *MQSeries Integrator Using*
- + *the Control Center* for information about promoting attributes.
- + – valueMandatory
- + This indicates if the <Attribute> must be specified. You must set this
- + property to “true” or “false”:
- + – Set “true” if the <Attribute> must be specified when an instance of the
- + node is created (the <Attribute> is mandatory).
- + – Set “false” if the <Attribute> does not have to be specified when an
- + instance of the node is created (the <Attribute> is optional). If not
- + specified, the <Attribute> takes the default value “”.
- + – encoded
- + This indicates if the value of this <Attribute> must be encoded (the XML
- + parser normalizes all strings: if tabs and line-feeds are to be retained, they
- + must be encoded):
- + – A tab ('\t') is encoded as '\\t'.
- + – A line feed ('\n') is encoded as '\\n'.
- + – A space (' ') is encoded as '+ '.
- + – A plus ('+') is encoded as '\\+'.
- + – The backslash character ('\') is encoded as '\\\'.
- + This normalizes a string to a one line string, and provides instructions to
- + allow the original string to be reconstructed by the Control Center.
- + This property is optional. It can have the value “true” or “false”. If you do
- + not include it, it takes the default value “false”.
- + If a MessageProcessingNodeType has <Attribute> tags, but no
- + <AttributeGroup> tags (like the sample in Figure 6 on page 70), the properties
- + dialog for the node might appear like this:

Implementing a node



- **AttributeGroup**

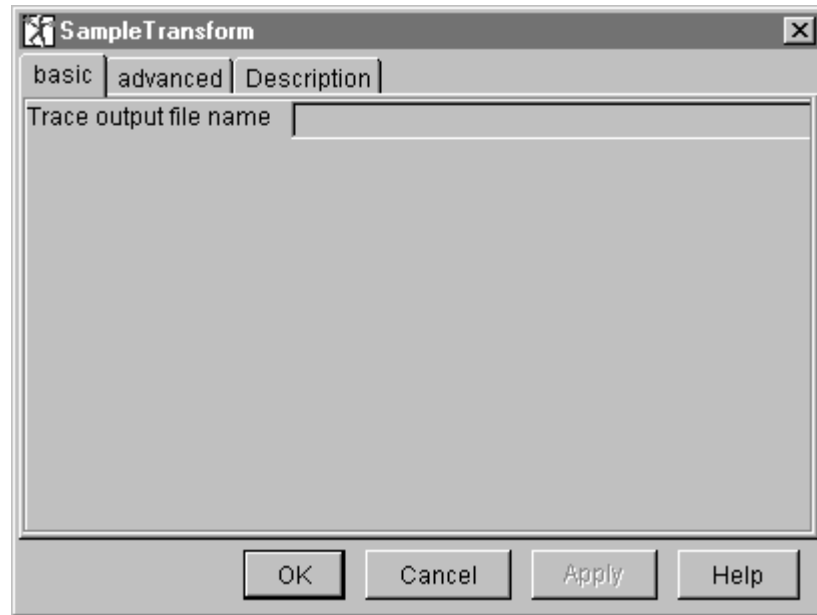
An `<AttributeGroup>` defines a number of related attributes for the node, that are displayed together on a separate tab within the node properties dialog. The use of attribute groups is optional. You can define zero or more `<AttributeGroup>` tags. Each `<AttributeGroup>` tag can include one or more `<Attribute>` tags.

The `<AttributeGroup>` tag has a single property:

- `xmi.label`

This character field defines the displayed label (name) of the tab that contains the group of attributes. If a value is provided, it acts as a key into the node's properties file. If there is no properties file, or the key does not exist within the properties file, the value of this field is used and its contents are displayed.

If a `MessageProcessingNodeType` has attribute groups in addition to, or in place of, attributes, the properties dialog for the node might appear like this:



+ The XML definitions required for the <AttributeGroup> and <Attribute> tags that
 + created this example are:

```
+ <AttributeGroup xmi.label="basic">
+ <Attribute xmi.label="Trace output file name" type="String" value=""
+ xmi.uuid="" valueMandatory="false" encoded="false"/>
+ </AttributeGroup>
+ <AttributeGroup xmi.label = "advanced">
+ <Attribute xmi.label="Trace setting for the node" type="Integer"
+ value="0" xmi.uuid="" valueMandatory="false" encoded="false"/>
+ </AttributeGroup>
```

+ **Note:** You are recommended to ensure that a single attribute is included in a
 + single attribute group. If you do not do so, the results are unpredictable.

+ • **OutTerminal**

+ This tag defines an out terminal of the node. You must code one for each
 + terminal through which your node propagates a message to another node (zero
 + or more). For example, you might have an out terminal and a failure terminal,
 + or a true terminal and a false terminal.

+ If you do not have an out terminal in your node, you can remove this entire tag.

+ You must set the following attributes of this tag:

+ – xmi.uuid

+ You must set this field to "". If you use the sample file supplied, you must
 + modify this property to have this value.

+ – xmi.label

+ This field defines the displayed label of the terminal. It must be unique
 + within this file, and it must be set to the same value as the name specified
 + for the terminal in the plug-in code.

Implementing a node

- **InTerminal**

This tag defines an in terminal of the node. You must code one and only one in terminal for your node.

You must set the following attributes of this tag:

- xmi.uuid

You must set this field to "". If you use the sample file supplied, you must modify this property to have this value.

- xmi.label

This field defines the displayed label of the terminal. It must be unique within this file, and it must be set to the same value as the name specified for the terminal in the plug-in code.

The WDP file:

The WDP (WebDav Properties) file provides information required by the WebDav protocol that is used between the Control Center and the Configuration Manager. The filename must be same as the name of the XML interface definition file, with the extension wdp, for example, ComIbmSampleTransform.wdp

This file specifies several properties required by WebDav. The file ComIbmSampleTransform.wdp, illustrated in Figure 7, is supplied for you to copy and update as indicated below.

```
<?xml version="1.0"?>
<properties xmlns:D="DAV:">
<D:creationdate>1999-10-06T12:54:19-04:00</D:creationdate>
<D:displayname>
  repository\private\MessageProcessingNodeType\ComIbmSampleTransform
</D:displayname>
<D:getcontenttype>text/plain</D:getcontenttype>
<D:xmi.label>SampleTransform</D:xmi.label>
<D:getcontenttype>text/plain</D:getcontenttype>
<D:lockdiscovery xmlns:D="DAV:"/>
<D:getcontenttype>text/plain</D:getcontenttype>
<D:icon>images/SampleTransform.gif</D:icon>
</properties>
```

Figure 7. Sample plug-in WDP file. The figure excludes repeated lines containing <D:getcontenttype>text/plain</D:getcontenttype> that you will see in the sample supplied. The duplicate lines are generated by WebDav but are not required: a single line is sufficient.

You must update the following fields with the appropriate values for your own node within the WDP file:

- displayname

Enter the name of the XML interface definition file (described in “The XML interface definition file” on page 69).

Note: The value set in the sample file includes a partial path in addition to the name of the file. The path is not required.

+ • `xmi.label`
 + Enter the name of the new node type. This is the text string that will appear in
 + the node tree in the Control Center (Message Flows view) to identify the type of
 + the node.

+ This must be identical in value to the `xmi.label` property of the
 + <MessageProcessingNodeType> tag in the XML interface definition file.

+ • `icon`
 + Enter the path and name of the icon defined for this node relative to
 + <mqsi_root>\Tool. This value must identify the minimum size icon. See
 + “Defining an icon for the node” for more information about node icons and
 + naming icon files. For more information about file locations, see “Storing the
 + files in the MQSeries Integrator directory structure” on page 84.

+ You can also update the following field if you choose:

+ • `creationdate`
 + This field can be used to record the date and time at which you created this
 + node. The contents of this field are not referenced within the implementation,
 + but the format of the field is significant and must be maintained in full. If you
 + modify the date, or time, or both, you must ensure that all other parts of this
 + field are unchanged.

+ **Defining optional node resources**

+ In addition to the node interface, you can also provide resources for the node that
 + provide for a new icon, a properties file (for translation of character fields
 + associated with the node), one or more property editors, and one or more
 + customizers. These tasks are all optional.

+ ***Defining an icon for the node:*** You are recommended to provide at least one
 + icon to be used in the Control Center tree view (the minimum size). You can
 + supply up to four additional icons of different sizes that are used for magnification
 + when viewing and zooming the message flow composition pane. If any of the
 + scaled icons do not exist, the closest in size is used, and is scaled to the expected
 + size. If you do not supply an icon, the default node icon is used.

+ The name of the minimum icon file must be the value of `xmi.label` in the
 + <MessageProcessingNodeType> tag in the XML interface definition file.

+ The following set of icons is supplied for the `SampleTransform` node:

- + • `SampleTransform.gif` for the tree view.
- + • `SampleTransform30.gif` for the 25% zoom.
- + • `SampleTransform42.gif` for the 50% zoom.
- + • `SampleTransform58.gif` for the 75% zoom.
- + • `SampleTransform84.gif` for the 100% zoom.

+ These icons are shown in Figure 8 on page 78.

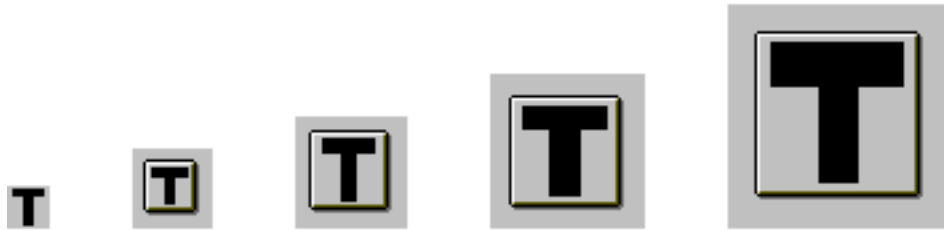


Figure 8. Plug-in node icons

You can reuse one or more of these icons when you create your own node by starting a graphics editor, making your changes, and using *File->Save As* to save the new icon with the appropriate name. You can also use the template icons supplied in `<mqs_i_root>\Tool\images` directory. See “Storing the files in the MQSeries Integrator directory structure” on page 84 for information about where your icon files must be stored.

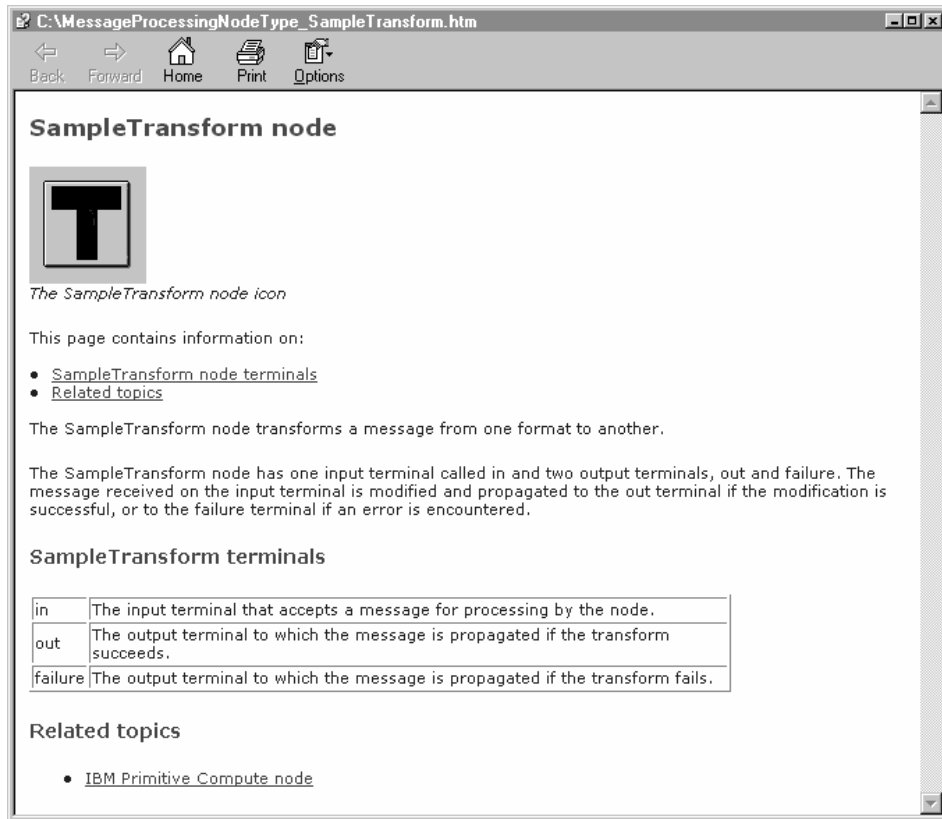
Defining the help text for the node: This subtask is optional, but you are recommended to complete it to provide online help information for your Control Center users. The help you provide will be processed in an identical fashion to the help provided for all the IBM Primitive nodes. This sample uses the style sheet used for the IBM primitives, `bipnt.css`, which also makes the appearance of the help text identical.

The name of the help file must be `MessageProcessingNodeType_<name>`, where `<name>` is the value set for `xmi.label` in the `<MessageProcessingNodeType>` tag in the XML interface definition file. The file extension must be `.htm`. See “Storing the files in the MQSeries Integrator directory structure” on page 84 for information about where your help file must be stored.

An example of a help file for the sample plug-in is shown in Figure 9 on page 79. The file must be called `MessageProcessingNodeType_SampleTransform.htm`. The paths specified for the icon file and the related link included in the help assume that you are using the default directory for `<mqs_i_root>`.

Implementing a node

+ The displayed help might appear as follows:



+ **Creating a properties file for the node:** This subtask is optional. You must
+ create this file if you want to enable translation to other national languages for the
+ properties of your node (for example, the descriptive text) that are displayed at the
+ Control Center interface. The sample plug-in supplied with MQSeries Integrator
+ includes a properties file, SampleTransform.properties.

+ The file must follow the required naming conventions: the name must be
+ <name>_<locale>.properties where:

- + • <name> is the name of the node the value of `xmi.label` in the
+ <MessageProcessingNodeType> tag in the XML interface definition file (for
+ example, SampleTransform).
- + • <locale> is the two character country code and two character language code.

+ This part of the name is optional: it can be omitted if the properties file is in the
+ default locale of the system on which you are running. For example, if you are
+ using the sample plug-in supplied, and your default locale is de_DE, the file
+ SampleTransform_de_DE.properties is searched for. If this does not exist,
+ SampleTransform.properties is referenced. If this latter file exists and its
+ contents have been set up for locale de_DE, the translated values are displayed.
+ If its contents are unchanged, the values defined are displayed regardless of
+ the language in which they are written.

+ If a properties file exists for the node, values can be defined and referenced by the
+ XML interface definition file for the node.

+ The sample node properties file is illustrated in Figure 10 on page 81.

```

+ shortDescription = Sample fixed transformation of an input message.
+ longDescription = The SampleTransform node is an example of a plug-in
+ message processing node.
+ nodeTraceSetting = Trace setting for the node
+ nodeTraceOutfile = Trace output file name

```

+ *Figure 10. Sample plug-in node properties file*

+ The general form of the contents of the properties file is:

```
+ <keyname> = text describing attribute
```

```
+
+ <keyname> is the value assigned to the corresponding property of a tag (for
+ example, the <MessageProcessingNodeType> tag) in the XML interface definition
+ file (described in “The XML interface definition file” on page 69). The keyname must
+ match the value set in the interface definition file. The descriptive text can be
+ translated to the language required by your users.
```

```
+
+ Creating a PropertyEditor: This subtask is optional. The supplied property
+ editors validate and process the following types of properties:
```

- + • String
- + • Integer
- + • Date
- + • Time
- + • Float
- + • Long
- + • Double
- + • Boolean
- + • Timestamp

```
+
+ If you want to define properties of other datatypes, you must create your own
+ property editor. This must follow the java.beans.PropertyEditor convention. The
+ class must implement this interface and must be named <type>PropertyEditor or
+ <type>Editor, where <type> is the value of the type property on the corresponding
+ <Attribute> tag in the XML interface definition file.
```

```
+
+ The three types of property editor (text-based, graphics-based, and
+ component-based) are supported.
```

- + • A property editor named <type>Editor is wrapped with the
+ WrapperPropertyEditor class provided in the tool. This class supports the
+ text-based property editor mode, and handles writing the Attribute value to the
+ XML document as a string (encoded if necessary).
- + • If a property editor is to support the graphics or component-based property
+ editor modes, it must be named <type>PropertyEditor. The property editor then
+ must use the setAttribute() and getAttribute() methods in the
+ org.w3c.dom.Element package to write the Attribute value to the XML
+ document.

```
+
+ The setValue() method on the property editor must be supported by all three types
+ of property editors. It is called with the org.w3c.dom.Element that represents the
+ Attribute that is being edited.
```

Implementing a node

+ The property editor works with a full copy (a clone) of the property. This allows it to
+ support the full range of actions to apply, cancel, and undo any changes made to
+ the properties by the user of the properties dialog. The final action of the user to
+ accept the changes made in the dialog (**Apply** or **OK**) causes the original
+ document to be updated.

+ The property editor has access to the entire XML document of which this node is a
+ part. It is therefore able to access any relevant information defined within that
+ document.

+ Figure 11 illustrates an extract of Java code for a property editor that handles the
+ <Attribute xmi.label="nodeTraceSetting"> from the SampleTransform node with
+ type="TraceSetting". See Appendix B, "Plug-in node sample code" on page 171
+ for a full listing of the source code for this program.

+ The property editor is named <type>PropertyEditor, so it must call the setAttribute()
+ and getAttribute() methods to set and get the Attribute's value. Because it is an
+ enumerated value, this property editor uses the getTags() method to get the valid
+ values, and they are displayed in a comboBox (the call to getTags() and the use of
+ the comboBox is handled by the tool).

```
+ public class TraceSettingPropertyEditor implements
+     java.beans.PropertyEditor {
+     Vector propertyChangeListeners = new Vector();
+     Element element = null;
+
+     public String getAsText() {
+         return element.getAttribute("value");
+     }
+
+     public java.lang.String[] getTags() {
+         String[] tags = {"0","1","2"};
+         return tags;
+     }
+
+     public void setAsText(String value) {
+         String oldValue = element.getAttribute("value");
+         element.setAttribute("value",value);
+         notifyListeners(oldValue);
+     }
+
+     public void setValue(Object value) {
+         String oldValue = null;
+         if(element != null) {
+             oldValue = element.getAttribute("value");
+         }
+         element = (Element)value;
+         notifyListeners(oldValue);
+     }
+
+     public Object getValue() {
+         return element;
+     }
+ }
```

+ *Figure 11. PropertyEditor sample code*

+ When you have created your property editor, you must compile it (using, for
+ example, javac) to create the .class file or files. You must ensure that you add
+ the jar files for any class that you use in the Property editor to the class path (that
+ is, to the CLASSPATH system environment variable) before you compile. For

+ example, if you are using the sample property editor, you must add
 + `<mqsi_root>\classes\xml4j.jar.`

+ When you have successfully compiled your property editor, you must store the
 + compiled files in the correct directory (specified in “Storing the files in the MQSeries
 + Integrator directory structure” on page 84). You must also add this directory to the
 + class path.

+ **Creating a Customizer:** This subtask is optional. If you need to control how the
 + properties of your node are displayed, and how the attributes are written, and
 + neither the default property editors nor a custom property editor support these
 + requirements, you must provide a customizer for your node.

+ The customizer follows the conventions of the `java.beans.Customizer`. The class
 + must implement this interface and must be named `<package>.<name>Customizer`
 + where:

- + • `<package>` is the value set for `package` in the `<MessageProcessingNodeType>`
 + tag in the XML interface definition file.
- + • `<name>` is the value set for `xmi.label` in the `<MessageProcessingNodeType>`
 + tag in the XML interface definition file.

+ A Customizer for the `SampleTransform` node would therefore be called
 + `com.isv.SampleTransformCustomizer`.

+ **Note:** This name is consistent with the content of the sample XML interface
 + definition file supplied, and with the samples java files documented in Appendix B,
 + “Plug-in node sample code” on page 171. It is not consistent with the
 + recommended naming convention.

+ The `setObject()` method on the Customizer is called with the `org.w3c.dom.Element`
 + that represents the `MessageProcessingNode` that is of the appropriate type.

+ The customizer works with a full copy (a clone) of the message processing node
 + document. This allows it to support the full range of actions to apply, cancel, and
 + undo any changes made to the properties by the user of the customizer dialog. The
 + final action of the user to accept the changes made in the dialog (**Apply** or **OK**)
 + causes the original document to be updated.

+ The customizer has access to the entire XML document of which this node is a
 + part. It is therefore able to access any relevant information defined within that
 + document.

+ Figure 12 on page 84 shows an extract of Java code for a customizer for the
 + sample `MessageProcessingNodeType` named “SampleSwitch”. The sample
 + customizer uses a radio button group and a file dialog to get values for attributes
 + called “nodeTraceSetting” and “nodeTraceOutfile”. Only a part of the `setObject()`
 + method (which is called by the tool upon instantiation of the customizer) and the
 + method used to write the “nodeTraceSetting” attribute value to the XML document
 + is shown here. See Appendix B, “Plug-in node sample code” on page 171 for a
 + full listing of the source code for this program.

Implementing a node

```
+ public class SampleSwitchCustomizer extends Panel
+ implements Customizer, ItemListener, ActionListener {
+     private Vector propertyChangeListeners = new Vector();
+     private Element rootNode = null;
+
+     public void setObject(Object bean) {
+         rootNode = (Element) bean;
+
+         // get Attributes for this node
+         NodeList Attributes = rootNode.getElementsByTagName("Attribute");
+         Element typeref = null;
+
+         for(int idx = 0; idx < Attributes.getLength(); idx++) {
+             Element attribute = (Element)Attributes.item(idx);
+             if(attribute.getAttribute("xmi.label").equals(TraceFileAttributeName)) {
+                 TraceFileAttribute = attribute;
+             }
+             if(attribute.getAttribute("xmi.label").equals(TraceSettingAttributeName)) {
+                 TraceSettingAttribute = attribute;
+             }
+         }
+         ...
+     }
+
+     private void setTraceSetting(int setting) {
+         //set Attribute value and notify listeners
+         String oldValue = TraceSettingAttribute.getAttribute("value");
+         String newValue = Integer.toString(setting);
+         if(!oldValue.equals(newValue)) {
+             //only if the value has changed
+             TraceSettingAttribute.setAttribute("value",newValue);
+             notifyListeners(oldValue, newValue);
+         }
+     }
+ }
```

Figure 12. Customizer sample code

When you have created your customizer, you must compile it (using, for example, javac) to create the .class file or files. You must ensure that you add the jar files for any class that you use in the customizer to the class path (that is, to the the CLASSPATH system environment variable) before you compile. For example, if you are using the sample customizer, you must add <mqsi_root>\classes\xml4j.jar and <mqsi_root>\classes\swingall.jar.

When you have successfully compiled your customizer, you must store the compiled files in the correct directory (specified in “Storing the files in the MQSeries Integrator directory structure”). You must also add the directory to the class path.

Installing a new message processing node in the Control Center

To complete this task you must complete two subtasks:

- Placing the files you have created in the directories from which they can be accessed.
- Defining the node to the Configuration Manager’s configuration repository to enable access by all users of the Control Center.

Storing the files in the MQSeries Integrator directory structure: When you have created the files required by your new node, you must install them in specific directories to make them accessible to MQSeries Integrator. These directories are

+ all subdirectories of the MQSeries Integrator home directory (into which MQSeries
+ Integrator was installed on this system), identified by the label <mqsi_root>.

+ You are recommended to stop the Control Center before you copy these files to the
+ appropriate directories.

- + • You must place the XML node interface file and the WDP file in the directory
+ <mqsi_root>\Tool\repository\private*hostname*\
+ *Queue_Manager_name*\MessageProcessingNodeType where:
 - + – *hostname* is the name of the system hosting the Configuration Manager
 - + – *Queue_Manager_name* is the name of the Configuration Manager's queue
+ manager

+ You have to specify these two values when you first establish the connection
+ between the Control Center and the Configuration Manager. The Configuration
+ Manager must have been created and started before the connection can
+ complete.

+ These files are retrieved from disk and stored in the configuration repository
+ when you check in the new node (this task is described in “Defining the node to
+ the configuration repository”). You must therefore store these files on the
+ system on which you will invoke the Control Center to define the new node to
+ the configuration repository.

- + • You must place the remaining files as follows:
 - + – The properties file, the property editor, and the customizer classes must be
+ placed in <mqsi_root>\Tool*<package>* where *<package>* is the location
+ identified by the package attribute on the <MessageProcessingNodeType>
+ tag in the XML interface definition file (with period delimiters replaced by
+ backslash characters \). For example, if *<package>* is set to *com.isv*, you
+ must store these files in <mqsi_root>\Tool\com\isv.
 - + – The help file must be placed in the subdirectory
+ <mqsi_root>\Tool\help*<package>* where *<package>* is the location
+ identified by the package attribute of the <MessageProcessingNodeType>
+ tag in the XML interface definition file (with period delimiters replaced by
+ backslash characters \). For example, if *<package>* is set to *com.isv*, you
+ must store this file in <mqsi_root>\Tool\help\com\isv.
 - + – The icon file or files must be placed in the subdirectory
+ <mqsi_root>\Tool\images.

+ You must store these files in these directories on the local system of every user
+ of the Control Center who needs to access the information about the new node.
+ These files are only accessed locally to each instance of the Control Center:
+ this information is not held centrally in the configuration repository for shared
+ access.

+ **Defining the node to the configuration repository:** When you have completed
+ the definition of the files required by your node, and have installed the files in the
+ appropriate directories, you must make these definitions available to the Control
+ Center. You **must** complete the tasks detailed in “Defining the interface of the
+ node” on page 69 and in “Storing the files in the MQSeries Integrator directory
+ structure” on page 84 before you start this task.

Implementing a node

- + 1. Start the Control Center. The user ID you are using must be a member of the MQSeries Integrator group **mqbrdevt**. You are recommended to use the superuser **IBMMQSI2** to complete this task¹. This causes your new node to be locked under the same user ID as all the supplied IBM primitive nodes. If you do not use this user ID, the definition files in the configuration repository might be accidentally unlocked, and therefore open to unauthorized update.
- + 2. Select the Message Flows view.
- + 3. Select an existing Message Flow Category, or create a new one.
- + 4. Right-click the selected category, and select *Add->Message Flow*.
+ A list box is displayed showing all existing IBM-supplied primitive nodes and
+ any plug-in nodes you have installed following the instructions provided.
- + 5. Select the plug-in node.
+ This node now appears within the message flow category you selected in the
+ tree view in the left-hand pane.
- + 6. Select your new node, and right-click. Select *Check In*.
- + 7. Right-click again, and select *Lock*. The interface and WDP definition files are
+ now stored in the configuration repository, and are available to all users of the
+ Control Center. However, users can only use this new node if they have
+ installed the additional files (icons, properties files, and so on) on their own
+ system.

Updating a message processing node

+ If you need to update the XML interface definition file or any of the support files for
+ a node you have created, you must follow these steps:

- + 1. Start the Control Center. If you locked the node using the MQSeries Integrator
+ superuser ID **IBMMQSI2**, you must be logged on with this user ID to make any
+ changes¹. If not, you can use any user ID that is a member of the MQSeries
+ Integrator group **mqbrdevt**.
- + 2. Select the Message Flows view.
- + 3. Select the node that you want to update.
- + 4. Unlock and delete the selected primitive.
- + 5. End your Control Center session.
- + 6. Install the updated files for this node into the appropriate directories (described
+ in “Storing the files in the MQSeries Integrator directory structure” on page 84).
+ If you have updated files other than the XML interface definition file and the
+ WDP file, you must remember to install the updated files on every system on
+ which the Control Center is used.
- + 7. Add your node back into the workspace following the instructions given in
+ “Defining the node to the configuration repository” on page 85.

+ ¹ You must take care if you change logon IDs to complete this task. Changing logon IDs can affect the operation of the
+ Configuration Manager’s queue manager if it on this system, but it is not running as a Windows NT service. See the *MQSeries
+ Integrator Administration Guide* for more information about queue manager operation (Chapter 2) and the superuser **IBMMQSI2**
+ (Chapter 4).

Implementing a message parser

The following tasks must be performed when implementing a message parser:

1. Write a plug-in initialization function
2. Write a context creation function
3. Implement the parser using appropriate utility functions

In practice, the task of writing a parser will vary considerably according to the complexity of the bit-stream to be parsed. Only the basic steps are described here.

Develop a plug-in initialization function

This is invoked when the plug-in is loaded during message broker initialization. The plug-in is responsible for:

- Creating and naming the message parser factory that is implemented by the plug-in. The parser factory is simply a container for related parser implementations. Parser factory names must be unique within a broker.
- Defining the supported message parser class names, and supplying a pointer to a virtual function table that contains pointers to the plug-in implementation functions. Parser class names must be unique within a broker.

The plug-in initialization function defines the name of the factory that the plug-in supports and the classes of objects supported by the factory. Each DLL (or .lib) that implements a plug-in must export a function called **bipGetParserFactory** as its initialization function.

The initialization function must create the factory object and define the names of all parsers supported by the plug-in. A factory can support any number of object classes (parsers). When a parser is defined, a list of pointers to the implementation functions for that parser is passed to the broker. If a parser of the same name already exists, the request is rejected. The plug-in initialization function is invoked automatically during broker initialization.

The initialization function must create a parser factory by invoking **cpicreateParserFactory**. The parser classes supported by the factory are defined by calling **cpidefineParserClass**. The address of the factory object (returned by **cpicreateParserFactory**) must be returned to the broker as the return value from the initialization function.

Develop a context creation function

Whenever an instance of a plug-in parser object is created, the context creation implementation function **cpicreateContext** is invoked by the message broker. This allows the plug-in parser to allocate instance data associated with the parser.

Implement the parser functions

The implementation functions:

cpiparseBuffer
cpiparseFirstChild
cpiparseLastChild
cpiparsePreviousSibling
cpiparseNextSibling
cpiparseWriteBuffer

Implementing a parser

provide the functionality of the plug-in parser. These implementation functions are invoked by the broker when an operation within the broker (such as a filter expression that specifies a message field name) requires a syntax element tree to be built or extended.

The implementations for these functions in the samples can be used as given, provided that the bit stream is one that is parsed progressively from beginning to end, producing corresponding syntax elements ordered from left to right. This condition is true for most common bit streams.

It will be seen in the samples that these implementations assume that the real parser code resides in a **parseNextItem** function. This function is expected to build the syntax element tree one element at a time, setting names, values and the complete flags appropriately. The implementation of this function is dependent on the nature of the bit stream to be parsed. The sample is an example of a simple pseudo-XML parser.

Messages with multiple message formats

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 be comprised 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 plug-in, the implementation function **cpiNextParserClassName** will be invoked by the broker when it needs to navigate down a chain of parser classes for a message comprising multiple message formats.

If your plug-in parser supports parsing a message format that is part of a multiple message format, then the plug-in *must* implement the **cpiNextParserClassName** function.

Accessing the message content

A message consists of a sequence of bytes. However, an application usually puts a special interpretation on that sequence. For example, the sequence of bytes might be the memory holding a C structure. The broker needs to deal with all messages in a general way, so it does not deal with the sequence of bytes directly, but instead references *syntax elements* that can be navigated to deduce the structure of a message.

Syntax elements

The model of a message presented to a message processing node is that of a parse tree of syntax elements, each of which can be one of three types:

Name elements	A name element has associated with it a string, which is the name of the element.
Value elements	A value element has a value associated with it.
Name-value element	A name-value element is an optimization of the case where a name element contains only a value element and nothing else. The element contains both a name and a value.

The root element is the unique element in the tree that has no parent. The root element is always a name element. Elements with the same parent element are said to be siblings. Sibling elements have a definite order, and iterating over the set of children of an element will always present the elements in the same order.

The element types that are needed depend on the message structure. Here are some examples of how a message can be modelled:

- If the message to be parsed consists of a series of name/scalar-value pairs, the message can be modelled using a tree with a depth of 1, with all of the elements (apart from the root) being name-value elements.
- Suppose the message has name/set-of-values pairs in addition to scalar values. In this case, the name/set-of-values pair can be modelled with a name element to represent the name, and one value element contained within the name element for each value in the set.
- The message might be the series of bytes holding a C structure, which itself contains nested structures. For example:

```
struct SubMessage {
    int field1;
    int field2;
};
struct Message {
    int field1;
    float field2;
    SubMessage field3;
};
```

```
char *messageBytes; // points to the actual message byte stream
Message *message = (Message*)messageBytes;
```

This message can be modelled by name-value elements representing each scalar field, and a name element representing each nested structure.

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 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 Figure 13.

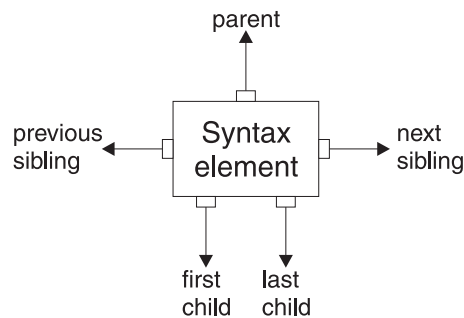


Figure 13. Tree representation of a message

Other functions support the manipulation of the elements themselves, with functions to create elements, to set or query their values, to insert new elements into the tree and to remove elements from the tree.

An element's type is set when it is created and it cannot be changed subsequently.

Updating a message buffer directly is not supported. You must modify a message by updating the syntax element tree. However, modifying an input message is not supported.

Figure 14 on page 91 describes a simple syntax element tree that shows a full range of interconnections between the elements.

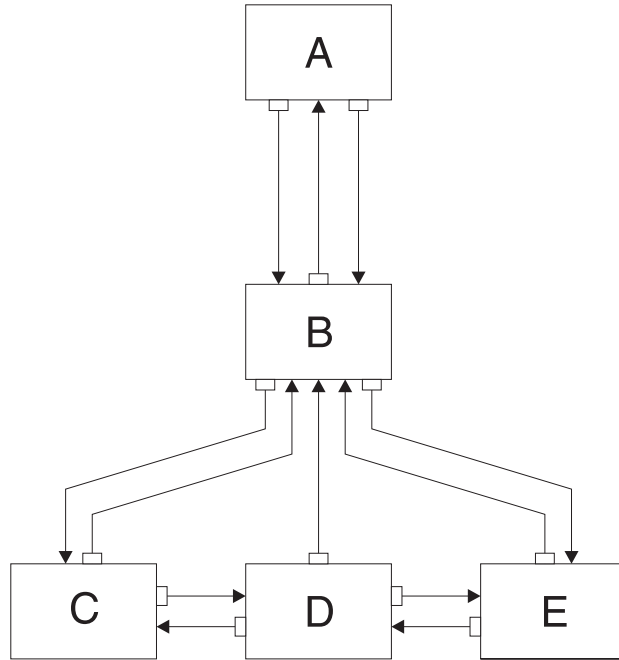


Figure 14. Syntax element tree

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**.

Figure 15 shows the typical first generation of syntax elements of a typical message received by MQSeries Integrator (similar to the example shown in Figure 14).

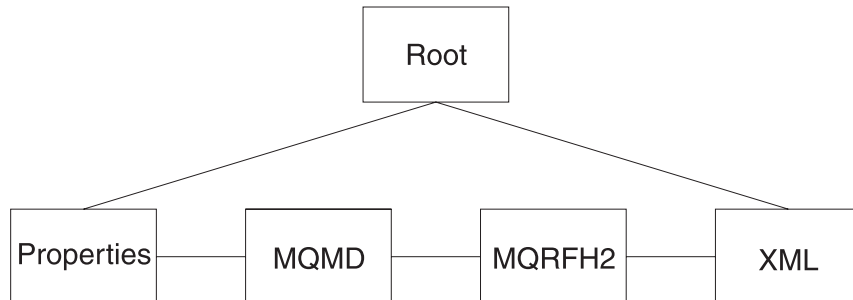


Figure 15. First generation of syntax elements

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. (For more information about standard properties, see *MQSeries Integrator Using the Control Center*.) The next sibling of *Properties* is the folder for the MQMD of the incoming

Accessing the message content

MQSeries 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.

Example

These concepts can be illustrated by an example. Suppose we have the following XML message:

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

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

Name	Business, Product, Company, Title, Location, Department
Value	IBM, Hursley, MQSeries
Name-value	type='messaging'

Figure 16 on page 93 shows the tree that represents the XML shown above.

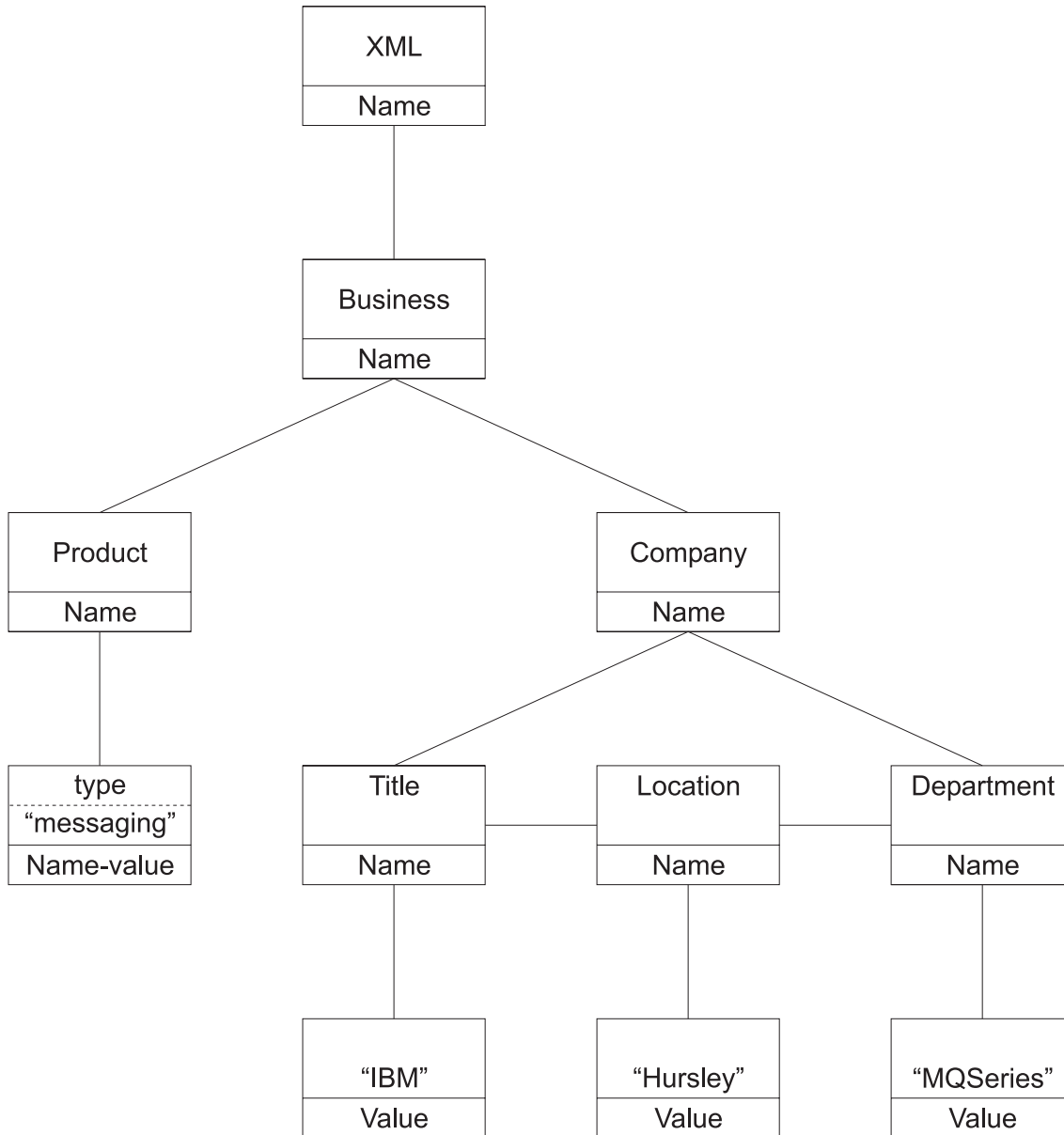


Figure 16. Tree representation of an XML message

How can you use these 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.

For a simple XML format message, the last child of the root element will be 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).

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

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

Accessing the message content

<Product> has an attribute (type='messaging'), which is a child element. Use **cniFirstChild** again to navigate to this element, and **cniElementType** to return its type (which is name-value). Use **cniElementName** 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**.

<Company> has three children, each one having a child that is a value element (IBM, Hursley, and MQSeries). 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.

Syntax element type definition

The element type is stored as a 32-bit integer. It is set using **cniSetElementType** and **cpiSetElementType** functions, and retrieved using the **cniElementType** and **cpiElementType** functions. As discussed previously, syntax elements are of three basic types: Name, Value, and Name/Value. This basic type is known as the *generic type*, and it is stored in the high-order byte of the element type. The low-order two bytes can be used to save parser-specific type information about the element; this is known as the *specific type*. For example, it can be used to denote an element of a special type, which needs to be handled differently when serialized to an output message by a parser. The remaining byte is reserved and must not be used.

If a message flow causes part of an element tree to be copied to another location, the specific type information is set to zeroes in the target elements of the copy, if the elements are owned by different parsers. This is because the bit values of the specific type are not meaningful to a different parser.

The **cniSearchElement** group allows a plug-in node to search from a given point in the element tree for an element of a particular type or name. These functions accept a search mode of type **CciCompareMode**; this mode allows the plug-in to search on all combinations of generic type, specific type, and element name.

Syntax element modification

All messages received on an input terminal of a message processing node are, by implication, *read only*. You must **not** attempt to modify the syntax element tree that belongs to an input message by adding or deleting syntax elements or by changing the attributes of any elements. To do so can cause unpredictable behavior in the message broker, and is not supported.

A plug-in node can *only* modify the syntax element tree of a message that it has created by using the **cniCreateMessage()** utility function.

General development considerations

There are a number of general development considerations and guidelines which should be addressed when implementing a plug-in function for MQSeries Integrator Version 2.0.

Runtime node behavior

A plug-in node must perform its processing in its **cniEvaluate** implementation function and return promptly. You must not design a plug-in node which suspends processing in any way, because this might prevent a message flow that contains such a node from being reconfigured or shutdown.

Portability

Message processing nodes and parsers should be coded in ANSI standard C, and avoid any use of operating system specific functions. This enables them to work on a variety of platforms with re-compilation only (without any source code changes).

Threading issues

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 plug-in 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.

Plug-in implementation functions must be re-entrant, and any function they invoke must also be re-entrant. All plug-in utility functions are fully re-entrant.

Although a plug-in 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.

It should be noted that it is not possible to develop an input node using the plug-in interface, because the internal thread management and allocation mechanisms are not available at this level.

Logging and trace

Message processing nodes and parsers are unlikely to need to write directly to the system event log, because it is recommended that a plug-in reports errors using exceptions (see “Errors and exception handling” on page 98). However, the **cciLog** utility function is provided so that an entry can be written to the event or trace log, if required.

Storage management

All memory allocated by a plug-in must be released by the plug-in. The construction of a node at run-time causes **cniCreateNodeContext** to be invoked, part of the intention of which is to allow the plug-in 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 plug-in function to be invoked; thus, the C plug-in can locate and use the correct context for the function processing. The message broker will pass addresses of C++ objects to the plug-in. These are simply intended to be used as a handle to be passed back on subsequent function calls. The C plug-in should never attempt to manipulate or use this pointer in any way, for example, attempting to release storage using the **free** function. 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 utilizes a plug-in 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.

String handling

Wide character strings are used throughout the interface to pass and return character data. Attributes are received in XML configuration messages as character strings, regardless of datatype. If the true datatype of an attribute is not a string, the **cniSetAttribute** function must perform the necessary verification and conversion prior to storing the attribute value. Similarly, when an attribute value is retrieved using **cniGetAttribute**, conversion must be performed to a wide character string prior to returning the result.

Configuration

Message processing nodes and parsers are required to work in a remotely administered server environment. Plug-in nodes should make provision for any configuration information they require to be passed to them using set methods, and should also provide corresponding methods for such attributes to be read back. If they read any information from the file system, the registry, environment variables or any other such local system resource, this will inhibit the ease of operation in a remote or distributed environment, because the message broker provides no features to administer such resources.

Other considerations

The broker uses ODBC as an interface to its own internal database, and to your own databases accessed in filter, compute and database processing nodes. The ODBC environment cannot be accessed using the plug-in node interface. It is therefore suggested that database access be done using the API functions supplied for that purpose, or by using the built-in processing nodes.

The broker internally maintains MQSeries connection and open queue handles on a thread-by-thread basis, and these are cached to optimize performance. It is

therefore suggested that the built-in **MQOutput** node is used when writing to MQSeries queues.

Using Windows NT event logging from a plug-in

You might choose to write significant events, error or otherwise, for problem determination and operational purposes to the Windows NT Event Log in the same manner as MQSeries Integrator Version 2.

The plug-in utility function CciLog can be used to do this. This function accepts two arguments, messageSource and messageNumber, which define the event source and the actual integer representation of a message within that source, respectively.

The message catalog is delivered as a Windows NT 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 NT Event Log Service:

1. Create a message compiler input (.mc) file with the source for your event messages. Refer to the Windows NT documentation 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 <inputdir> -r <outputdir> <filename>
```

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 plug-in 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 #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 NT Event Log Service:

1. Start the Windows NT Registry Editor by issuing the command:
regedit
2. Create a new registry subkey for your plug-in application under the existing structure defined in:

Error handling

```
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 MQSeries Integrator key
+ “MQSeriesIntegrator2”). You must give the key the name that you specify on
+ the messageSource parameter of the CciLog() invocation.

+ You must then 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”.

See “Adding a Source to the Registry” in the Windows NT documentation for more details about this task.

Errors and exception handling

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 will, by default, bypass any C plug-in 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 will sometimes indicate that a failure 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 will contain CCI_SUCCESS. Otherwise, it will contain 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 will be visible to the plug-in 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 plug-in will not be aware of that exception
- The utility function will not return to the plug-in
- Execution control will pass to higher layers in the broker stack to process the exception

This means that a plug-in 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** can be used to obtain diagnostic information on the type

of exception that occurred, returning this data in the CCI_EXCEPTION_ST structure.

Message inserts can be returned in the CCI_STRING_ST members of the CCI_EXCEPTION_ST structure. The CCI_STRING_ST allows the plug-in to provide a buffer to receive any required inserts. The broker will copy the data into this buffer and will return 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 plug-in can then perform any error recovery, if required. If CCI_EXCEPTION is returned, all exceptions must be passed back to the message broker for additional error recovery to be performed. This is done by invoking **cciRethrowLastException**, which causes the C interface to re-throw the last exception so that it can be handled by other layers in the message broker.

If an exception occurs and is caught by a plug-in, the plug-in must not call any utility functions except **cciGetLastExceptionData** or **cciRethrowLastException**. An attempt to call other utility functions will result in unpredictable behavior which could comprise the integrity of the broker.

If a plug-in encounters a serious error, **cciThrowException** can be used to generate an exception that will be processed by the message broker in the correct manner. The generation of such an exception causes the supplied information to be written into the broker event log.

Types of exception and broker behavior

The broker generates a set of exceptions that can be advised to a plug-in. These exceptions can also be generated by a plug-in 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. |

Error handling

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 attempting to convert to another datatype. A conversion exception is treated as a recoverable exception by the broker.
User	These are generated when a ThrowNode throws a user-defined exception.

Return codes

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 **BipCci.h**.

Table 3 (Page 1 of 2). 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_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_INV_DATA_BUFLen	A data buffer length of zero was specified.
CCI_INV_DATA_POINTER	A null pointer was specified for the address of an output data area.
CCI_INV_DATASOURCE_NAME	A datasource name was not specified.
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_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_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 plug-in 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.

Table 3 (Page 2 of 2). Utility function return codes and values

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.

Note:

1 This return code is returned only by **cniGetLastExceptionData** to indicate the type of the last exception.

Error handling

Chapter 7. Node implementation and utility functions

The plug-in interface for a message flow node consists of:

1. A set of implementation functions that provide the functionality of the plug-in node. These functions are invoked by the message broker. The implementation functions are mandatory, and if not supplied by the developer will cause an exception at runtime.
2. A set of utility functions, the purpose of which is to create resources in the message broker or to request a service of the broker. These utility functions can be invoked by a plug-in node.
3. A set of utility functions for exception handling and logging that can be invoked by a plug-in node or parser.

This chapter contains:

- “Node implementation function overview” on page 104.
- “Node utility function overview” on page 105.
- “Exception handling and logging overview” on page 107.
- “Node implementation function interface” on page 108.
- “Node utility function interface” on page 112.

Node implementation function overview

The plug-in needs to implement 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, the setting and retrieval of attribute values, the function to actually perform the processing of the node itself and functions to examine messages.

The following functions are mandatory, and must be implemented by the developer.

Follow the page references to see the detailed descriptions of each implementation function.

cniCreateNodeContext	page 108
cniDeleteNodeContext	page 108
cniEvaluate	page 109
cniGetAttribute	page 109
cniGetAttributeName	page 110
cniSetAttribute	page 110

Node utility function overview

The following system-provided functions allow the C plug-in to 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.

Follow the page references to see the detailed descriptions of each utility function.

Initialization and resource creation

cniCreateNodeFactory	page 118
cniDefineNodeClass	page 119
cniCreateInputTerminal	page 118
cniCreateOutputTerminal	page 119
cniGetBrokerInfo	page 124

Message management

cniCreateMessage	page 118
cniDeleteMessage	page 120
cniFinalize	page 123
cniGetMessageContext	page 124
cniIsTerminalAttached	page 125
cniPropagate	page 127

Message buffer access

cniBufferByte	page 113
cniBufferPointer	page 114
cniBufferSize	page 114
cniWriteBuffer	page 133

Syntax element navigation

cniRootElement	page 128
cniParent	page 126
cniPreviousSibling	page 127
cniNextSibling	page 126
cniFirstChild	page 124
cniLastChild	page 126
cniSearchElement group	page 128

Syntax element access

cniAddAfter	page 112
cniAddBefore	page 113
cniAddAsFirstChild	page 112
cniAddAsLastChild	page 113
cniCopyElementTree	page 114
cniCreateElementAfter	page 115
cniCreateElementAfterUsingParser	page 115
cniCreateElementBefore	page 117
cniCreateElementBeforeUsingParser	page 117
cniCreateElementAsFirstChild	page 115
cniCreateElementAsFirstChildUsingParser	page 116
cniCreateElementAsLastChild	page 116
cniCreateElementAsLastChildUsingParser	page 116
cniDetach	page 120
cniElementName	page 120
cniElementType	page 121
cniElementValue group	page 121
cniElementValueState	page 122
cniElementValueType	page 122
cniElementValueValue	page 123
cniGetParserClassName	page 125
cniSetElementName	page 129
cniSetElementType	page 129
cniSetElementValue group	page 130
cniSetElementValueValue	page 131

SQL statement handling

cniSqlCreateStatement	page 131
cniSqlExecute	page 132
cniSqlSelect	page 133
cniSqlDeleteStatement	page 132

Exception handling and logging overview

The following exception handling and logging functions are provided for use by a plug-in node or a plug-in parser.

Follow the page references to see the detailed descriptions of each function.

cciGetLastExceptionData	page 159
cciLog	page 159
cciThrowException	page 160
cciRethrowLastException	page 160

Node implementation function interface

The plug-in needs to implement 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, the setting and retrieval of attribute values, the function to actually perform the processing of the node itself and functions to examine messages.

The following functions have to be implemented, using the prototypes as described.

cniCreateNodeContext

Creates any plug-in 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.

The responsibilities of the plug-in 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 (such as context) that might be required (for attribute data and terminals, for example).
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 a plug-in implementation function for this node instance is invoked, the appropriate context is passed as an argument to that function. This means that a plug-in node developed in C need not maintain its own static pointers to per-instance data areas.

```
CciContext* cniCreateNodeContext(  
    CciFactory* factoryObject,  
    CciChar*    nodeName,  
    CciNode*   nodeObject);
```

`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 plug-in context is returned. Otherwise, a value of zero (`CCI_NULL_ADDR`) is returned.

cniDeleteNodeContext

Deletes any plug-in context for an instance of a node object. It is invoked by the message broker whenever an instance of a node object is destroyed. A message flow node may be deleted when reconfiguring or redeploying a broker.

The responsibilities of the plug-in 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.


```
void cniDeleteNodeContext(CciContext* context);
```

`context` The address of the plug-in context for the instance of the node, as created and returned by the **cniCreateNodeContext** function (input).

Return values: None.

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 node.

The responsibilities of the plug-in 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.

```
void cniEvaluate(
    CciContext *context,
    CciMessage *destinationList,
    CciMessage *exceptionList,
    CciMessage *message);
```

`context` The address of the plug-in context for the instance of the node, as created by the plug-in 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).

Return values: None.

cniGetAttribute

Gets the value of an attribute on a specific node instance. It is invoked by the message broker when a report request is received that causes a retrieval of the value of a node attribute. The broker will have verified that the attribute name is valid for the node.

The responsibilities of the plug-in are to:

1. Return a character representation of the attribute value.
2. Return a null string if the data is sensitive and should not be displayed in reports.
3. Throw an exception if an error occurs.

```
int cniGetAttribute(
    CciContext* context,
    CciChar* attrName,
    CciChar* buffer,
    int bufsize);
```

Node implementation functions

context	The address of the plug-in context for the instance of the node, as created by the plug-in 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 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 plug-in, a non-zero value is returned.

cniGetAttributeName

Returns the name of a node attribute by an index. It is invoked by the message broker when it 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.

```
int cniGetAttributeName(  
    CciContext* context,  
    int index,  
    CciChar* buffer,  
    int bufsize);
```

context	The address of the plug-in context for the instance of the node, as created by the plug-in 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 will be copied (output).
bufsize	The length of the buffer specified in the buffer argument (input).

Return values: If successful, zero is returned, and the name of the attribute is returned in the supplied buffer. If the end of the list of attributes is reached, a non-zero value is returned.

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. A plug-in will receive requests to set attributes for the base. If an unknown attribute value is received, this function **must** return a non-zero value; this causes the broker to process the request correctly.

The responsibilities of the plug-in are to:

1. Verify that the value of the attribute is correctly specified. If not, a configuration exception should be thrown using the **cniThrowException** 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 **cniThrowException** function.

```
int cniSetAttribute(  
    CciContext* context,  
    CciChar* attrName,  
    CciChar* attrValue);
```

context The address of the plug-in context for the instance of the node, as created by the plug-in and returned by the **cniCreateNodeContext** function (input).

attrName The name of the attribute for which its 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 plug-in, a non-zero value is returned.

Node utility function interface

The following system-provided functions allow the C plug-in to 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.

The node utility functions are defined in the header file **BipCni.h**.

cniAddAfter

Adds an unattached syntax element after a specified syntax element. The currently unattached syntax element, and any child elements it might possess, 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).

```
void cniAddAfter(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElement*  newElement);
```

returnCode This parameter receives the return code from the function (output).
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, 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 might possess, is connected to the syntax element tree as the **first child** of the specified target element. The target element need not be attached.

```
void cniAddAsFirstChild(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElement*  newElement);
```

returnCode This parameter receives the return code from the function (output).
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, 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 might possess, is connected to the syntax element tree as the **last child** of the specified target element. The new element need not be attached.

```
void cniAddAsLastChild(
    int*         returnCode,
    CciElement* targetElement,
    CciElement* newElement);
```

returnCode This parameter receives the return code from the function (output).

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, 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 might possess, 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).

```
void cniAddBefore(
    int*         returnCode,
    CciElement* targetElement,
    CciElement* newElement);
```

returnCode This parameter receives the return code from the function (output).

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, 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.

```
CciByte cniBufferByte(
    int*         returnCode,
    CciMessage* message,
    CciSize     index);
```

returnCode This parameter receives the return code from the function (output).

Node utility functions

message Specifies 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

Get a pointer to the data buffer associated with (and owned by) the message object specified in the message argument.

```
const CciByte* cniBufferPointer(  
    int*         returnCode,  
    CciMessage* message);
```

returnCode This parameter receives the return code from the function (output).

message Specifies 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.

```
CciSize cniBufferSize(  
    int*         returnCode,  
    CciMessage* message);
```

returnCode This parameter receives the return code from the function (output).

message Specifies 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. If an error occurred, zero (`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. Before the copy is performed, all existing child elements of the target element are deleted, to be replaced by the child elements of the source element.

```
void cniCopyElementTree(  
    int*         returnCode,  
    CciElement* sourceElement,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

sourceElement Specifies the address of the source syntax element object (input).

targetElement Specifies 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.

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.

```
CciElement* cniCreateElementAfter(
    int*      returnCode,
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

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.

In MQSeries Integrator Version 2, a portion of the syntax element tree that is owned by a parser may **only** have its effective root at the first generation of elements (that is, as *immediate children of root*). The plug-in 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 bitstream when outputting a message.

```
CciElement* cniCreateElementAfterUsingParser(
    int*      returnCode,
    CciElement* targetElement,
    const CciChar* parserClassName);
```

returnCode This parameter receives the return code from the function (output).

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.

```
CciElement* cniCreateElementAsFirstChild(
    int*      returnCode,
    CciElement* targetElement);
```

Node utility functions

returnCode This parameter receives the return code from the function (output).
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.

In MQSeries Integrator Version 2, a portion of the syntax element tree that is owned by a parser may **only** have its effective root at the first generation of elements (that is, as *immediate children of root*). The plug-in 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 bitstream when outputting a message.

```
CciElement* cniCreateElementAsFirstChildUsingParser(  
    int*          returnCode,  
    CciElement*  targetElement,  
    const CciChar* parserClassName);
```

returnCode This parameter receives the return code from the function (output).
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

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

```
CciElement* cniCreateElementAsLastChild(  
    int*          returnCode,  
    CciElement*  targetElement);
```

returnCode This parameter receives the return code from the function (output).
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.

cniCreateElementAsLastChildUsingParser

Creates a new syntax element as the last child of the specified syntax element, and associates it with the specified parser class name.

In MQSeries Integrator Version 2, a portion of the syntax element tree that is owned by a parser may **only** have its effective root at the first generation of elements (that is, as *immediate children of root*). The plug-in 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 bitstream when outputting a message.

```
CciElement* cniCreateElementAsLastChildUsingParser(
    int*          returnCode,
    CciElement*  targetElement,
    const CciChar* parserClassName);
```

returnCode This parameter receives the return code from the function (output).

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.

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.

```
CciElement* cniCreateElementBefore(
    int*          returnCode,
    CciElement*  targetElement);
```

returnCode This parameter receives the return code from the function (output).

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.

In MQSeries Integrator Version 2, a portion of the syntax element tree that is owned by a parser may **only** have its effective root at the first generation of elements (that is, as *immediate children of root*). The plug-in 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 bitstream when outputting a message.

```
CciElement* cniCreateElementBeforeUsingParser(
    int*          returnCode,
    CciElement*  targetElement,
    const CciChar* parserClassName);
```

returnCode This parameter receives the return code from the function (output).

targetElement The address of the element object (input).

Node utility functions

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. Note that this function must be called only from within the implementation function **cniCreateNodeContext**.

```
CciTerminal* cniCreateInputTerminal(  
    int*      returnCode,  
    CciNode*  nodeObject,  
    CciChar*  name);
```

returnCode This parameter receives the return code from the function (output).

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.

cniCreateMessage

Creates a new output message object.

```
CciMessage* cniCreateMessage(  
    int*          returnCode,  
    CciMessageContext* messageContext);
```

returnCode This parameter receives the return code from the function (output).

messageContext The address of the context for the message (input). Use **cniGetMessageContext** to get the context from an incoming message (one received in the **cniEvaluate** function, for instance).

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.

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.

```
CciFactory* cniCreateNodeFactory(
    int*      returnCode,
    CciChar*  name);
```

`returnCode` This parameter receives the return code from the function (output).
`name` Specifies 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.

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. Note that this function must be called from within the implementation function **cniCreateNodeContext**.

```
CciTerminal* cniCreateOutputTerminal(
    int*      returnCode,
    CciNode*  nodeObject,
    CciChar*  name);
```

`returnCode` This parameter receives the return code from the function (output).
`nodeObject` Specifies 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` 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.

cniDefineNodeClass

Defines a node class, as specified by the `name` parameter, which is supported by the node factory specified as the `factoryObject` parameter. The parameter `functbl` is a pointer to a CNI_VFT structure that contains pointers to the C plug-in implementation functions (those functions that provide the function of the node itself).

```
void cniDefineNodeClass(
    int*      returnCode,
    CciFactory* factoryObject,
    CciChar*  name,
    CNI_VFT*  functbl);
```

`returnCode` This parameter receives the return code from the function (output).
`factoryObject` Specifies the address of the factory object which is to support the named node (input). The address is returned from **cniCreateNodeFactory**.
`name` The name of the node to be defined (input). The name of the node must end with the characters "Node".

Node utility functions

`funcTbl` The address of the CNI_VFT structure that contains pointers to the plug-in implementation functions (input).

Return values: None. If an error occurs, the `returnCode` parameter indicates the reason for the error.

cniDeleteMessage

Deletes the specified message object.

```
void cniDeleteMessage(  
    int*          returnCode,  
    CciMessage*  message);
```

`returnCode` This parameter receives the return code from the function (output).

`message` Specifies 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.

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.

```
void cniDetach(  
    int*          returnCode,  
    CciElement*  targetElement);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies 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.

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**.

```
CciSize          cniElementName(  
    int*          returnCode,  
    CciElement*  targetElement,  
    const CciChar* value,  
    Ccysize      length);
```

`returnCode` This parameter receives the return code from the function (output).

`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` Specifies the length of the buffer specified by the `buffer` parameter (input).

Return values: If successful, the element name is copied into the supplied buffer and the number of bytes 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 bytes required is returned. For any other failures, `CCI_FAILURE` is returned and `returnCode` indicates the reason for the error.

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**.

```
CciElementType cniElementType(
    int*         returnCode,
    CciElement* targetElement);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies 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.

```
CciSize cniElementBitArrayValue(
    int*         returnCode,
    CciElement* targetElement,
    const struct CciBitArray* value);
```

```
CciBool cniElementBooleanValue(
    int*         returnCode,
    CciElement* targetElement);
```

```
CciSize cniElementByteArrayValue(
    int*         returnCode,
    CciElement* targetElement),
    const struct CciBitArray* value);
```

```
CciSize cniElementCharacterValue(
    int*         returnCode,
    CciElement* targetElement,
    const CciChar* value,
    CciSize      length);
```

```
struct CciDate cniElementDateValue(
    int*         returnCode,
    CciElement* targetElement);
```

```
void cniElementDecimalValue(
    int*         returnCode,
    CciElement* targetElement,
    const CciChar* value,
    CciSize      length);
```

```
struct CciTimestamp cniElementGmtTimestampValue(
    int*         returnCode,
    CciElement* targetElement);
```

Node utility functions

```
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);
```

returnCode This parameter receives the return code from the function (output).

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 specified by the **value** parameter (input). Used on relevant function calls only.

Return values: The value of the target element is returned. If an error occurs, the **returnCode** parameter indicates the reason for the error. In cases where the size of an element's data can vary, the correct data size is returned. Also, if the specified length is too small, the error code is set to **CCI_BUFFER_TOO_SMALL**.

cniElementValueState

Gets the state of the value of the specified syntax element.

```
CciValueState cniElementValueState(  
    int*      returnCode,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

targetElement Specifies 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.

```
CciValueType cniElementValueType(  
    int*      returnCode,  
    CciElement* targetElement);
```

`returnCode` This parameter receives the return code from the function (output).
`targetElement` Specifies 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.

```
CciElementValue* cniElementValueValue(
    int*          returnCode,
    CciElement*  targetElement);
```

`returnCode` This parameter receives the return code from the function (output).
`targetElement` Specifies 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.

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 prior to propagating a message.

```
void cniFinalize(
    int*          returnCode,
    CciMessage*  message,
    int          options);
```

`returnCode` This parameter receives the return code from the function (output).
`message` Specifies 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). Can be omitted, or set to `CCI_FINALIZE_VALIDATE`.

Return values: None. If an error occurs, the `returnCode` parameter indicates the reason for the error.

cniFirstChild

Returns the address of the syntax element object that is the first child of the specified syntax element.

```
CciElement* cniFirstChild(  
    int*      returnCode,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

targetElement Specifies 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.

cniGetBrokerInfo

Queries the current broker environment (for example, for information about broker name and message flow name) and returns this information in CNI_BROKER_INFO_ST.

```
void CniGetBrokerInfo(  
    int*      returnCode,  
    CciNode*  nodeObject,  
    CNI_BROKER_INFO_ST* broker_info_st
```

returnCode This parameter receives the return code from the function (output).

nodeObject Specifies the message flow processing node for which broker environment information is being requested.

broker_info_st Specifies the address of a CNI_BROKER_INFO_ST structure that will be 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.

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.

```
CciMessageContext* cniGetMessageContext(  
    int*      returnCode,  
    CciMessage* message);
```

returnCode This parameter receives the return code from the function (output).

message Specifies 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.

cniGetParserClassName

Gets the parser class name associated with the specified syntax element.

```
CciSize ImportExportPrefix ImportExportSuffix cniGetParserClassName(
    int*          returnCode,
    CciElement*  targetElement
    const CciChar* value,
    CciSize      length);
```

returnCode This parameter receives the return code from the function (output).

targetElement Specifies the address of the element for which the parser class name is to be returned (input).

value Specifies the address of an output buffer into which the parser class name will be stored (input).

length Specifies the length of the output buffer 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.

cniIsTerminalAttached

Checks whether a terminal is attached to another node via 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. It can be used to test whether a message can be propagated to a terminal. However, note that it is *not* necessary to call this function prior to propagating a message with the **cniPropagate** utility function. The intention of this function is to allow a node to modify its behavior when a terminal is not connected.

```
int cniIsTerminalAttached(
    int*          returnCode,
    CciTerminal* terminalObject);
```

returnCode This parameter receives the return code from the function (output).

terminalObject Specifies 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 via a connector, a value of 1 is returned. If the terminal is not attached, or a failure occurred, a value of 0 is returned. If a failure occurred, the value of the **returnCode** parameter indicates the reason for the error.

cniLastChild

Returns the address of the syntax element object that is the last child of the specified syntax element.

```
CciElement* cniLastChild(  
    int*      returnCode,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

targetElement Specifies 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.

cniNextSibling

Returns the address of the syntax element object that is the next sibling (right sibling) of the specified syntax element.

```
CciElement* cniNextSibling(  
    int*      returnCode,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

targetElement Specifies 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.

cniParent

Returns the address of the syntax element object that is the parent of the specified syntax element.

```
CciElement* cpiParent(  
    int*      returnCode,  
    CciElement* targetElement);
```

returnCode This parameter receives the return code from the function (output).

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.

cniPreviousSibling

Returns the address of the syntax element object that is the previous sibling (left sibling) of the specified syntax element.

```
CciElement* cniPreviousSibling(
    int*      returnCode,
    CciElement* targetElement);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies 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 simply not propagated, and the function is regarded as a no-op. Therefore, it is not necessary to check whether the terminal is attached prior to propagating, unless the action that the node takes would be different (in which case **`cnilsTerminalAttached`** can be used to check if the terminal is connected).

```
int cniPropagate(
    int*      returnCode,
    CciTerminal* terminalObject,
    CciMessage* destinationList,
    CciMessage* exceptionList,
    CciMessage* message);
```

`returnCode` This parameter receives the return code from the function (output).

`terminalObject` Specifies the address of the output terminal to receive the message (input). The address is returned by **`cniCreateOutputTerminal`**.

`destinationList` Specifies the address of the destination list object to be sent with the message (input). Note: 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` Specifies the address of the message object to be sent (input). If the message being sent is the same as the input message, then this address will be the one passed on the **`evaluate`** 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.

cniRootElement

Get the root syntax element associated with a specified message. It returns the root element that is associated with (and owned by) the message object identified by the message parameter. When a message object is constructed by the broker, a root element is automatically created.

```
CciElement* cniRootElement(  
    int*      returnCode,  
    CciMessage* message);
```

returnCode This parameter receives the return code from the function (output).
message Specifies 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.

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 element argument, 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 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 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 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 a match is found or the start of the sibling chain is reached.

```
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);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies the address of the syntax element object from which the search is started (input).

`mode` The search mode to use (input). This indicates what combination of element type and element name is to be searched for.

`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.

Return values: The address of the requested syntax element object is returned, unless there is no matching element, in which case zero is returned. If an error occurs, zero (CCI_NULL_ADDR) is returned and the `returnCode` parameter indicates the reason for the error.

cniSetElementName

Sets the name of the specified syntax element.

```
void cniSetElementName(
    int*          returnCode,
    CciElement*  targetElement,
    const CciChar* name);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies the address of the target syntax element object (input).

`name` Specifies the name of the element (input).

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.

```
void cniSetElementType(
    int*          returnCode,
    CciElement*  targetElement,
    CciElementType type);
```

`returnCode` This parameter receives the return code from the function (output).

`targetElement` Specifies the address of the target syntax element object (input).

`type` Specifies the type of the element (input).

Return values: None. If an error occurs, the `returnCode` parameter indicates the reason for the error.

cniSetElementValue group

Functions to set a value into the specified syntax element.

```
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);
```

returnCode	This parameter receives the return code from the function (output).
targetElement	Specifies 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.

cniSetElementValueValue

Sets the value object of the specified syntax element.

```
void cniSetElementValueValue(
    int*          returnCode,
    CciElement*  targetElement,
    CciElementValue* value);
```

returnCode	This parameter receives the return code from the function (output).
targetElement	Specifies 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.

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. 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**. Multiple SQL expression objects can be created in a single message flow processing node. Although these objects can be created at any time, they will typically be created when the message flow processing node is instantiated, within the implementation function **cniCreateNodeContext**.

```
CciSqlExpression* cniSqlCreateStatement(
    int*          returnCode,
    CciNode*     nodeObject,
    CciChar*     dataSourceName,
    CciSqlTransaction transaction,
    CciChar*     statement);
```

returnCode	This parameter receives the return code from the function (output).
nodeObject	Specifies the message flow processing node the SQL expression object will be owned by (input). This pointer is passed to the cniCreateNodeContext implementation function.
dataSourceName	The ODBC data source name to be used if the statement references data in an external database (input).

Node utility functions

transaction	Specifies whether a database commit will be performed after the statement is executed (input). Valid values are CCI_SQL_TRANSACTION_AUTO (the default) and CCI_SQL_TRANSACTION_COMMIT. The former value specifies that a database commit will be performed at the completion of the message flow (that is, as a fully globally coordinated or partially globally coordinated transaction). The latter value specifies that a commit will be performed after execution of the statement, and within the cniSqlExecute or cniSqlSelect function (that is, the message flow is partially broker coordinated).
statement	Specifies 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 the SQL statement previously created using the **cniSqlCreateStatement** utility function, as defined by the sqlExpression argument.

```
void cniSqlDeleteStatement(  
    int*          returnCode,  
    CciSqlExpression* sqlExpression);
```

returnCode	This parameter receives the return code from the function (output).
sqlExpression	Specifies 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 argument. This function is to be used when the statement does not return data, for example, when a PASSTHRU function is used.

```
void cniSqlExecute(  
    int*          returnCode,  
    CciSqlExpression* sqlExpression,  
    CciMessage*   destinationList,  
    CciMessage*   exceptionList,  
    CciMessage*   message);
```

returnCode	This parameter receives the return code from the function (output).
sqlExpression	Specifies 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` argument. If the statement returns data, then it is written into the message specified by the `outputMessage` argument.

```
void cniSqlSelect(
    int*          returnCode,
    CciSqlExpression* sqlExpression,
    CciMessage*   destinationList,
    CciMessage*   exceptionList,
    CciMessage*   message,
    CciMessage*   outputMessage);
```

`returnCode` This parameter receives the return code from the function (output).

`sqlExpression` Specifies 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 will be written (output).

Return values: None. If an error occurs, the `returnCode` parameter indicates the reason for the error.

cniWriteBuffer

Causes the syntax element tree associated with the specified message to be written to the data buffer owned by that message object. This operation serializes the element tree into a bitstream, which can then be processed as a sequence of contiguous bytes. This function should be used when writing the bitstream to a target that is outside the broker (that is, when writing a plug-in output node).

```
void cniWriteBuffer(
    int*          returnCode,
    CciMessage*   message);
```

`returnCode` This parameter receives the return code from the function (output).

`message` Specifies 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.

Node utility functions

Chapter 8. Parser implementation and utility functions

The plug-in interface for a message parser consists of:

1. A set of implementation functions, which provide the functionality of the plug-in parser. These functions are invoked by the message broker. Most implementation functions are mandatory, and if not supplied by the developer will cause an exception at runtime.
2. A set of utility functions, the purpose of which is to create resources in the message broker or to request a service of the broker. These utility functions can be invoked by a plug-in parser.
3. A set of utility functions for exception handling and logging that can be invoked by a plug-in node or parser.

This chapter contains:

- “Parser implementation function overview” on page 136.
- “Parser utility function overview” on page 137.
- “Exception handling and logging overview” on page 139.
- “Parser implementation function interface” on page 140.
- “Parser utility function interface” on page 146.

See also Chapter 9, “Exception handling and logging” on page 159.

Parser implementation function overview

A message parser plug-in implements its capability through a function interface that 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.

Most of the following functions are mandatory, and must be implemented by the developer.

Follow the page references to see the detailed descriptions of each implementation function.

cpiCreateContext	page 140
cpiDeleteContext	page 140
cpiElementValue	page 143
cpiNextParserClassName	page 143
cpiNextParserCodedCharSetId	page 144
cpiNextParserEncoding	page 144
cpiParseBuffer	page 141
cpiParseFirstChild	page 141
cpiParseLastChild	page 142
cpiParseNextSibling	page 142
cpiParsePreviousSibling	page 142
cpiParserType	page 144
cpiSetElementValue	page 145
cpiSetNextParserClassName	page 145
cpiWriteBuffer	page 143

Parser utility function overview

The following functions allow the C plug-in to create or define message broker objects, such as message parser factories.

Follow the page references to see the detailed descriptions of each utility function.

Initialization and resource creation

<code>cpiCreateParserFactory</code>	page 149
<code>cpiDefineParserClass</code>	page 150

Message buffer access

<code>cpiAppendToBuffer</code>	page 147
<code>cpiBufferByte</code>	page 147
<code>cpiBufferPointer</code>	page 148
<code>cpiBufferSize</code>	page 148

Syntax element navigation

<code>cpiFirstChild</code>	page 153
<code>cpiLastChild</code>	page 153
<code>cpiNextSibling</code>	page 154
<code>cpiParent</code>	page 154
<code>cpiPreviousSibling</code>	page 154
<code>cpiRootElement</code>	page 155

Syntax element access

<code>cpiAddAfter</code>	page 146
<code>cpiAddBefore</code>	page 147
<code>cpiAddAsFirstChild</code>	page 146
<code>cpiAddAsLastChild</code>	page 146
<code>cpiCreateAndInitializeElement</code>	page 148
<code>cpiCreateElement</code>	page 149
<code>cpiElementCompleteNext</code>	page 150
<code>cpiElementCompletePrevious</code>	page 150
<code>cpiElementName</code>	page 151
<code>cpiElementType</code>	page 151
<code>cpiElementValue</code> group	page 151
<code>cpiElementValueValue</code>	page 153
<code>cpiSetCharacterValueFromBuffer</code>	page 155
<code>cpiSetElementCompleteNext</code>	page 155
<code>cpiSetElementCompletePrevious</code>	page 156

Parser function overview

cpisetElementName	page 156
cpisetElementType	page 156
cpisetElementValue group	page 157
cpisetElementValueValue	page 158
cpisetNameFromBuffer	page 158

Exception handling and logging overview

The following exception handling and logging functions are provided for use by a plug-in node or a plug-in parser.

Follow the page references to see the detailed descriptions of each function.

cciGetLastExceptionData	page 159
cciLog	page 159
cciThrowException	page 160
cciRethrowLastException	page 160

Parser implementation function interface

A message parser plug-in implements its capability through a function interface that 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.

The following functions have to be implemented, using the prototypes as described, except those functions that are specified as optional.

cpiCreateContext

Creates a plug-in 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 plug-in 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 a plug-in implementation function for this parser instance is invoked, the appropriate context is passed as an argument to that function. This means that a plug-in parser developed in C need not maintain its own static pointers to per-instance data areas.

```
CciContext* cpiCreateContext(CciParser* parser);
```

parser The address of the parser object that has been constructed (input).

Return values: If successful, the address of the plug-in context is returned. Otherwise, a value of zero is returned.

cpiDeleteContext

Deletes the plug-in context associated with a parser object. It is invoked by the message broker when an instance of a parser object is destroyed.

The responsibilities of the plug-in 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.

```
void cpiDeleteContext(  
    CciParser* parser,  
    CciContext* context);
```


parser The address of the parser object (input).
context The address of the plug-in context (input).

Return values: None.

cpiParseBuffer

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 plug-in 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 plug-in 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.

```
int cpiParseBuffer(
    CciParser*  parser,
    CciContext* context,
    int         offset);
```

parser The address of the parser object (input).
context The address of the plug-in context (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.

cpiParseFirstChild

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.

```
void cpiParseFirstChild(
    CciParser*  parser,
    CciContext* context,
    CciElement* currentElement);
```

parser The address of the parser object (input).

Parser implementation functions

context The address of the plug-in context (input).
currentElement The address of the current syntax element (input).

Return values: None.

cpiParseLastChild

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.

```
void cpiParseLastChild(  
    CciParser*  parser,  
    CciContext* context,  
    CciElement* currentElement);
```

parser The address of the parser object (input).
context The address of the plug-in context (input).
currentElement The address of the current syntax element (input).

Return values: None.

cpiParseNextSibling

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.

```
void cpiParseNextSibling(  
    CciParser*  parser,  
    CciContext* context,  
    CciElement* currentElement);
```

parser The address of the parser object (input).
context The address of the plug-in context (input).
currentElement The address of the current syntax element (input).

Return values: None.

cpiParsePreviousSibling

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.

```
void cpiParsePreviousSibling(  
    CciParser*  parser,  
    CciContext* context,  
    CciElement* currentElement);
```

parser The address of the parser object (input).
context The address of the plug-in context (input).
currentElement The address of the current syntax element (input).

Return values: None.

cpiWriteBuffer

Writes a syntax element tree to the message buffer associated with a parser. It appends data to the bitstream 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 (bitstream) with data from the element tree.

```
void cpiWriteBuffer(
    CciParser*  parser,
    CciContext* context);
```

parser The address of the parser object (input).

context The address of the plug-in context (input).

Return values: None.

cpiElementValue

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 plug-in parser to override the behavior for retrieving element values.

```
const CciElementValue* cpiElementValue(
    CciParser*  parser,
    CciElement* currentElement);
```

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.

cpiNextParserClassName

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.

```
void cpiNextParserClassName(
    CciParser*  parser,
    CciContext* context,
    CciChar*    buffer,
    int         size);
```

parser The address of the parser object (input).

context The address of the plug-in context (input).

Parser implementation functions

buffer The address of a buffer into which the parser class name should be put (input).

size The length of the buffer provided by the broker (input).

Return values: None.

cppNextParserCodedCharSetId

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

```
int cpiNextParserCodedCharSetId(
    CciParser*  parser,
    CciContext* context);
```

parser The address of the parser object (input).

context The address of the plug-in context (input).

Return values: The CCSID of the data is returned. If it is not known, zero may be returned and a default CCSID will apply.

cppNextParserEncoding

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

```
int cpiNextParserEncoding(
    CciParser*  parser,
    CciContext* context);
```

parser The address of the parser object (input).

context The address of the plug-in context (input).

Return values: The encoding of the data is returned. If it is not known, zero may be returned and default encoding will apply.

cpiParamType

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 **cpiParamType** implementation function is not provided, the message broker assumes that the parser is of the *standard* type.

```
CciBool cpiParamType(
    CciParser*  parser,
    CciContext* context);
```

parser The address of the parser object (input).

context The address of the plug-in context (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.

cciSetElementValue

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 plug-in parser to override the behavior for setting element values.

```
void cciSetElementValue(
    CciParser*      parser,
    CciElement*    currentElement,
    CciElementValue* value);
```

parser The address of the parser object (input).

currentElement The address of the current syntax element (input).

value The value (input).

Return values: None.

cciSetNextParserClassName

Optional function to advise a parser of the next parser in the chain. It is called during finalize processing, and returns to the plug-in 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 prior to the phase that causes serialization of the bit stream.

```
void cciSetNextParserClassName(
    CciParser*      parser,
    CciContext*    context,
    CciChar*       name,
    CciBool*       parserType);
```

parser The address of the parser object (input).

context The address of the plug-in context (input).

name A pointer to a string containing the parser class name (output).

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.

Parser utility function interface

The following functions allow the C plug-in to create or define message broker objects, such as message parser factories.

The parser utility functions are defined in the header file **BipCpi.h**.

cpiAddAfter

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.

```
void cpiAddAfter(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElement*  newElement);
```

returnCode Receives the return code from the function (output).

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.

cpiAddAsFirstChild

Adds a new (and currently unattached) syntax element to the syntax element tree as the first child of the specified target element.

```
void cpiAddAsFirstChild(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElement*  newElement);
```

returnCode Receives the return code from the function (output).

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.

cpiAddAsLastChild

Adds a new (and currently unattached) syntax element to the syntax element tree as the last child of the specified target element.

```
void cpiAddAsLastChild(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElement*  newElement);
```

returnCode Receives the return code from the function (output).
 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.

cpiAddBefore

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.

```
void cpiAddBefore(
    int*          returnCode,
    CciElement*  targetElement,
    CciElement*  newElement);
```

returnCode Receives the return code from the function (output).
 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.

cpiAppendToBuffer

Appends data to the buffer containing the bit stream representation of a message, for the specified parser object.

```
void cpiAppendToBuffer(
    int*          returnCode,
    CciParser*   parser,
    CciByte*     data,
    CciSize      length);
```

returnCode Receives the return code from the function (output).
 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.

cpiBufferByte

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.

Parser utility functions

```
CciByte cpiBufferByte(  
    int*      returnCode,  
    CciParser* parser,  
    CciSize   index);
```

returnCode Receives the return code from the function (output).

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.

cpiBufferPointer

Gets a pointer to the buffer containing the bit stream representation of the input message, for the specified parser object.

```
const CciByte* cpiBufferPointer(  
    int*      returnCode,  
    CciParser* parser);
```

returnCode Receives the return code from the function (output).

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.

cpiBufferSize

Gets the size of the buffer containing the bit stream representation of the input message, for the specified parser object.

```
CciSize cpiBufferSize(  
    int*      returnCode,  
    CciParser* parser);
```

returnCode Receives the return code from the function (output).

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.

cpiCreateAndInitializeElement

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.


```
CciElement* cpiCreateAndInitializeElement(
    int*          returnCode
    CciParser*   parser,
    CciElementType type,
    const CciChar* name,
    CciBool      firstChildComplete,
    CciBool      lastChildComplete);
```

returnCode Receives the return code from the function (output).

parser Specifies the address of the parser object (input). This address is passed to the plug-in 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.

cpiCreateElement

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.

```
CciElement* cpiCreateElement(
    int*          returnCode,
    CciParser*   parser);
```

returnCode Receives the return code from the function (output).

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.

cpiCreateParserFactory

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 'lib' is loaded by the message broker. If **cpiCreateParserFactory** is invoked at any other time, the results are unpredictable.

```
CciFactory* cpiCreateParserFactory(
    int*          returnCode,
    CciChar*     name);
```

returnCode Receives the return code from the function (output).

name Specifies the name of the factory being created (input).

Parser utility functions

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.

cciDefineParserClass

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 plug-in implementation functions, that is, those functions that provide the function of the parser itself.

```
void cciDefineParserClass(  
    int*          returnCode,  
    CciFactory*  factoryObject,  
    CciChar*     name,  
    CPI_VFT*    functbl);
```

returnCode Receives the return code from the function (output).

factoryObject Specifies the address of the factory object that supports the named parser (input). The address is returned from **cciCreateParserFactory**.

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 plug-in implementation functions (input).

Return values: None. If an error occurs, returnCode indicates the reason for the error.

cciElementCompleteNext

Gets the value of the 'next child complete' flag from the target syntax element. This attribute indicates whether the element tree is complete.

```
CciBool cciElementCompleteNext(  
    int*          returnCode,  
    CciElement*  targetElement);
```

returnCode Receives the return code from the function (output).

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.

cciElementCompletePrevious

Gets the value of the 'previous child complete' flag from the target syntax element. This attribute indicates whether the element tree is complete.

```
CciBool cciElementCompletePrevious(  
    int*          returnCode,  
    CciElement*  targetElement);
```

returnCode Receives the return code from the function (output).

`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.

cpiElementName

Gets the name of the target syntax element. The syntax element name will have been set previously using **`cniSetElementName`** or **`cpiSetElementName`**.

```
CcSize      cpiElementName(
  int*      returnCode,
  CciElement* targetElement),
const CciChar* value,
CcSize      length);
```

`returnCode` Receives the return code from the function (output).

`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` Specifies the length of the buffer specified by the `buffer` parameter (input).

Return values: If successful, the element name is copied into the supplied buffer and the number of bytes 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 bytes required is returned. For any other failures, `CCI_FAILURE` is returned and `returnCode` indicates the reason for the error.

cpiElementType

Gets the type of the target syntax element. The syntax element type will have been set previously using **`cniSetElementType`** or **`cpiSetElementType`**.

```
CciElementType cpiElementType(
  int*      returnCode,
  CciElement* targetElement);
```

`returnCode` Receives the return code from the function (output).

`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.

cpiElementValue group

Functions to get the value of the specified syntax element.

```
CcSize cpiElementBitArrayValue(
  int*      returnCode,
  CciElement* targetElement);
const struct CciBitArray* value);
```

```
CciBool cpiElementBooleanValue(
  int*      returnCode,
  CciElement* targetElement);
```

Parser utility functions

```
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);
```

returnCode Receives the return code from the function (output).

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 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.

cciElementValueValue

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 **cciSetElementValueValue** function. This can be used by parsers that override behavior by invoking the implementation functions **cciElementValue** and **cciSetElementValue**.

```
const CciElementValue* cciElementValueValue(
    int*          returnCode,
    CciElement*  targetElement);
```

returnCode Receives the return code from the function (output).

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.

cciFirstChild

Returns the address of the syntax element object that is the first child of the specified target element.

```
CciElement* cciFirstChild(
    int*          returnCode,
    const CciElement* targetElement);
```

returnCode Receives the return code from the function (output).

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.

cciLastChild

Returns the address of the syntax element object that is the last child of the specified target element.

```
CciElement* cciLastChild(
    int*          returnCode,
    const CciElement* targetElement);
```

returnCode Receives the return code from the function (output).

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.

cppNextSibling

Returns the address of the syntax element object that is the next (right) sibling of the specified target element.

```
CciElement* cppNextSibling(  
    int*          returnCode,  
    const CciElement* targetElement);
```

returnCode Receives the return code from the function (output).

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.

cpParent

Returns the address of the syntax element object that is the parent of the specified target element.

```
CciElement* cpParent(  
    int*          returnCode,  
    const CciElement* targetElement);
```

returnCode Receives the return code from the function (output).

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.

cpPreviousSibling

Returns the address of the syntax element object that is the previous (left) sibling of the specified target element.

```
CciElement* cpPreviousSibling(  
    int*          returnCode,  
    const CciElement* targetElement);
```

returnCode Receives the return code from the function (output).

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 previous 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.

cpiRootElement

Gets the address of the root syntax element of the specified parser object.

```
CciElement* cpiRootElement(
    int*      returnCode,
    CciParser* parser);
```

returnCode Receives the return code from the function (output).

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.

cpiSetCharacterValueFromBuffer

Sets the value of the specified syntax element.

```
void cpiSetCharacterValueFromBuffer(
    int*      returnCode,
    CciElement* targetElement,
    const CciChar* value,
    CciSize   length);
```

returnCode Receives the return code from the function (output).

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 specified by the value parameter (input).

Return values: None. If an error occurs, returnCode indicates the reason for the error.

cpiSetElementCompleteNext

Sets the 'next child complete' flag in the target syntax element to the specified value.

```
void cpiSetElementCompleteNext(
    int*      returnCode,
    CciElement* targetElement,
    CciBool   value);
```

returnCode Receives the return code from the function (output).

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.

cciSetElementCompletePrevious

Sets the 'previous child complete' flag in the target syntax element to the specified value.

```
void cciSetElementCompletePrevious(  
    int*         returnCode,  
    CciElement* targetElement,  
    CciBool      value);
```

returnCode Receives the return code from the function (output).

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.

cciSetElementName

Sets the name of the specified syntax element.

```
void cciSetElementName(  
    int*         returnCode,  
    CciElement* targetElement,  
    const CciChar* name);
```

returnCode Receives the return code from the function (output).

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.

cciSetElementType

Sets the type of the specified syntax element.

```
void cciSetElementType(  
    int*         returnCode,  
    CciElement* targetElement,  
    CciElementType type);
```

returnCode Receives the return code from the function (output).

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.

cpisetElementValue group

Functions to set a value in the specified syntax element.

```

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);

```

Parser utility functions

returnCode	This argument receives the return code from the function (output).
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 (input). Used on relevant function calls only.

Return values: None. If an error occurs, returnCode indicates the reason for the error.

cpisetElementValueValue

Sets the value of the specified syntax element. See **cpisetElementValueValue** on page 153.

```
void ImportExportPrefix ImportExportSuffix cpisetElementValueValue(  
    int*          returnCode,  
    CciElement*  targetElement,  
    CciElementValue* value);
```

returnCode	Receives the return code from the function (output).
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.

cpisetNameFromBuffer

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.

```
void cpisetNameFromBuffer(  
    int*          returnCode,  
    CciElement*  targetElement,  
    const CciChar* name,  
    CciSize      length);
```

returnCode	Receives the return code from the function (output).
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 name (input).

Return values: None. If an error occurs, returnCode indicates the reason for the error.

Chapter 9. Exception handling and logging

The following exception handling and logging functions are provided for use by a plug-in node or a plug-in parser.

They are defined in the header file **BipCci.h**.

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. It can be used by the plug-in to determine whether any recovery is required when a utility function returns an error code of CCI_EXCEPTION.

This function may be called when a utility function has indicated that an exception occurred by setting returnCode to CCI_EXCEPTION.

```
void* cciGetLastExceptionData(
    int*          returnCode,
    CCI_EXCEPTION_ST* exception_st);
```

returnCode This parameter receives the return code from the function (output).

exception_st Specifies the address of a CCI_EXCEPTION_ST structure to receive data about the last exception (output).

Return values: None. If an error occurs, the returnCode parameter indicates the reason for the error.

cciLog

Logs an error, warning or informational event. The event is logged by the message broker interface using the specified arguments as log data.

```
void cciLog(
    int*          returnCode,
    CCI_LOG_TYPE type,
    char*        file,
    int          line,
    char*        function,
    CciChar*    messageSource,
    int          messageNumber,
    char*        traceText,
    ...);
```

Exception handling

returnCode	This parameter receives the return code from the function (output).
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 can be useful for debugging purposes.
line	The line number in the source file where the function was invoked (input). The value is optional, but it can be useful for debugging purposes.
function	The function name which invoked the log function (input). The value is optional, but it can be useful for debugging purposes.
messageSource	A string that identifies the Windows NT message source.
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 trace log (input). The value is optional, but it can be 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. 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.

cciRethrowLastException

Rethrows the last exception generated on the current thread. It is used to pass the exception back to the message broker for further handling.

```
void cciRethrowLastException(int* returnCode);
```

returnCode This parameter receives the return code from the function (output).

Return values: None. If an error occurs, the returnCode parameter indicates the reason for the error.

cciThrowException

Throws an exception. The exception is thrown by the message broker interface using the specified arguments as exception data.

```

void cciThrowException(
    int*          returnCode,
    CCI_EXCEPTION_TYPE type,
    char*        file,
    int         line,
    char*        function,
    CciChar*    messageSource,
    int         messageNumber,
    char*        traceText,
    ...);

```

returnCode	This parameter receives the return code from the function (output).
type	The type of exception (input). Valid values are: CCI_FATAL_EXCEPTION CCI_RECOVERABLE_EXCEPTION CCI_CONFIGURATION_EXCEPTION CCI_PARSER_EXCEPTION CCI_CONVERSION_EXCEPTION
file	The source file name where the exception was generated (input). The value is optional, but it can be useful for debugging purposes.
line	The line number in the source file where the exception was generated (input). The value is optional, but it can be useful for debugging purposes.
function	The function name which generated the exception (input). The value is optional, but it can be useful for debugging purposes..
messageSource	A string that identifies the Windows NT message source.
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 which will be written into the broker trace log (input). The value is optional, but it can be 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 pointer to char. 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.

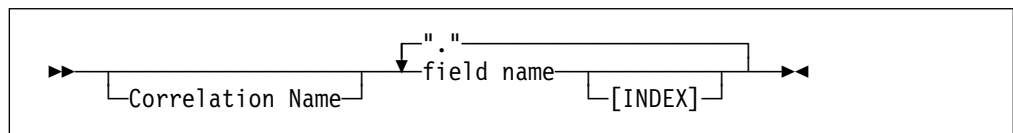
Exception handling

Part 3. Appendixes

Appendix A. Using filters in content-based routing

The language used in the specification of filters for content-based routing (CBR) forms a proper subset of the Filter node's language (See *MQSeries Integrator Using the Control Centre*.) A subset of the language is provided because the processing involved in the filtering of message content when a publication is made differs from that used in the Filter node. This processing is performed by a component of the Publish/Subscribe function in the broker, named the Matching Engine. At publication time, many filters must be tested to see if they match the publication and the Matching Engine has been optimized to perform this task.

Field references



The field references that may be used in CBR filters form a subset of those supported by the Filter node. As with the Filter node, it is necessary to specify a path in order to reference a field in a filter. Each element of the path comprises a (possibly indexed) field name.

The syntax of a field reference is shown above. `field name` and `Correlation Name` are identifiers. MQSeries Integrator represents all messages as a hierarchical syntax element tree. Each path identifies a route through that tree, which leads to a particular syntax element, starting from one of the predefined correlation names that refer to fixed points that every message has. The following correlation names are supported for CBR:

Root	Identifies the root of a published message.
Properties	Identifies the portion of the message in which the standard properties of a message lie.
Body	Identifies the last child of the root of the message, which is usually, but not always, the application data that follows any headers.
None of Root, Properties or Body	An implicit <code>Body.</code> is prepended to the field reference.

Here are some examples of field references, together with their meanings:

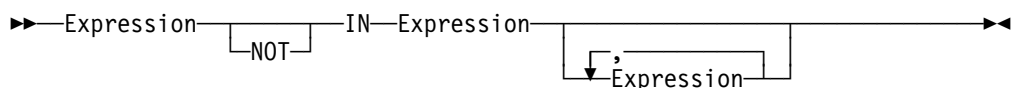
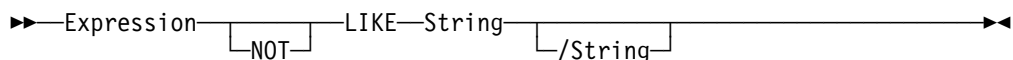
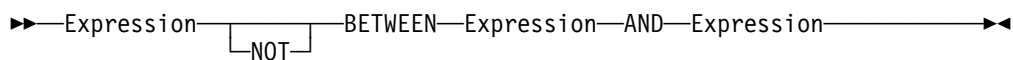
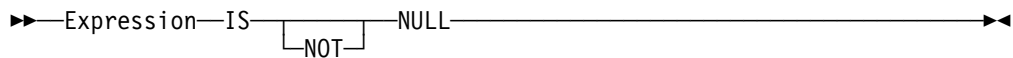
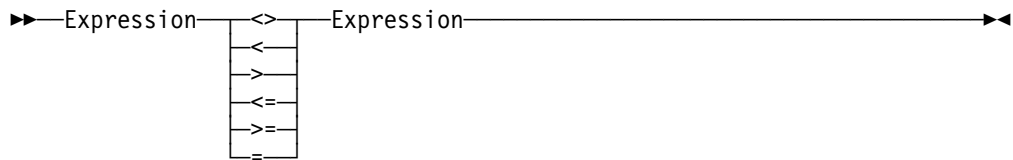
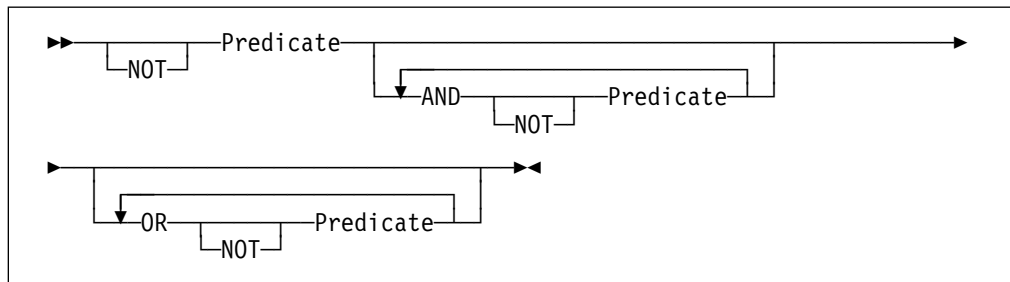
Person.Salary	Refers to the Salary field in the Person entity in the body of the message.
Body.Person.Address[0]	Refers to the first Address field in the Person entity in the body of the message.
Properties.Topic	Refers to the "Topic" field in the standard properties of a message.

Root.MQMD.UserIdentifier	Refers to the UserIdentifier field in the MQMD of the message.
--------------------------	--

Note: Path elements of “*” and the array index “LAST” are not supported in CBR filters.

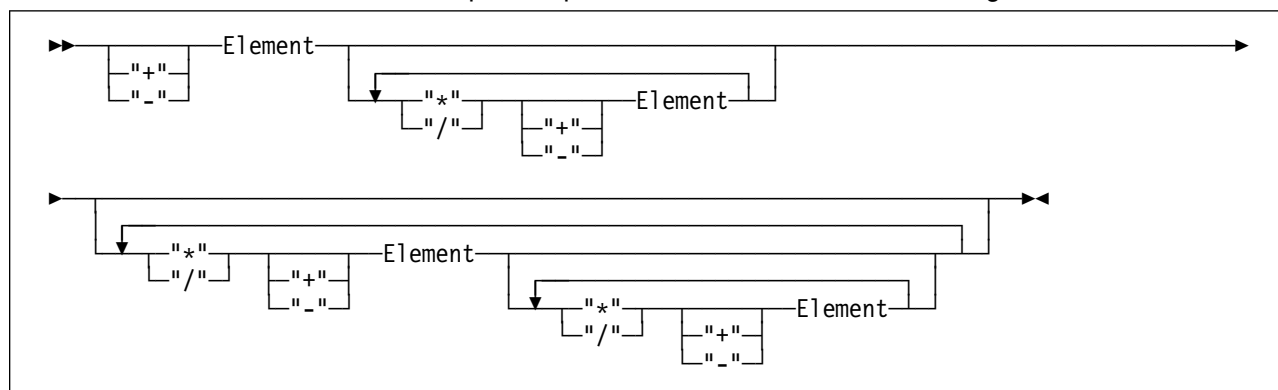
Specifying a filter

A filter is specified through a combination of predicates as shown here:



Note: The Between predicate is Asymmetric only (that is, the lowest value must be specified as the first operand).

Predicates comprise expressions that can take the following form:



An Element can be one of:

A Field Reference	As described above
A String Literal	A character string. The length of the string is the number of characters in the sequence. If the length is zero, the value is called the empty string. The latter is not a NULL value. The String may include double quote characters. Single quotes or backslashes should be preceded by a backslash, so that: Field1='Howard\'s "Bubble" Car\\Stephanie\'s Cycle' expresses a single valid string literal that will be matched against a message with Field1 set to the string: 'Howard's "Bubble" Car\Stephanie's Cycle'
An Integer Literal	Signed or unsigned, 64 bit, for example: Field1 = -123
A Floating Point Literal	Represented by two numbers separated by an "E". The first number may include a sign and a decimal point. The second may include a sign but not a decimal point. For example: Field1=-1.79E+23
A Boolean Literal	May be TRUE or FALSE.
A Filter specification	Allowing the nesting of filter expressions.

Some filter examples

Person.Salary>10000	Filtering against an integer literal
"Person.Address"[1]NOT LIKE 'Blen%'AND"Person.Salary">15000	A more complex filter. Note that field identifiers may optionally be surrounded by double quotes.
Body.Date1='2000-02-14'	Filtering against a date. The date is matched as a string and care must be taken with its layout (see below).
Person.ApprovalFlag	Filtering against a Boolean field.
Person.Salary+Person.Bonus>Person.Limit	An arithmetic filter.
Properties.Topic='employees/marketing'	Filtering on a message property.
Root.MQMD.UserIdentifier='Blair'	Filtering on a message attribute.

Person.HourlyRate = 10.24	Filtering against a float literal
Planet.DistanceFromSun = 0.93E8	Filtering against a float literal in exponential format

Datatypes and type mappings

CBR exploits a set of four internal datatypes. These are:

- String
- Integer
- Float
- Boolean

The set of supported MQSeries Integrator datatypes are mapped to these types in the following way.

Character

A value of the CHARACTER datatype is mapped to a CBR String representation comprising two-byte Unicode characters.

Boolean

A value of the BOOLEAN datatype is mapped to the CBR Boolean type, which may be true or false. An implicit cast is supported between string literals 'TRUE' and 'FALSE' and the corresponding boolean values (for example, if message field Body.Field1 is set to 'True', it will match a boolean value of true).

Integer

A value of the INTEGER datatype is mapped to a CBR integer value, which is an exact numeric number stored with 64-bit binary precision.

Float

A value of the FLOAT datatype is mapped to a CBR float value, which is a 64-bit binary approximation of a real number.

The number can be zero or can range from -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

Decimal

A value of the DECIMAL datatype is mapped to a CBR float value which is a 64 bit binary approximation of a real number (see above).

Date

A value of the DATE datatype is mapped to a string representation of the form 'YYYY-MM-DD', allowing filters such as Body.Date1='2000-02-14' to be specified.

Time and GMTTime

A value of the TIME datatype is mapped to a string representation of the form 'HH:MM:SS[.UUUUUU]', allowing filters such as Body.Time1='10:36:11' to be specified. (UUUUUU indicates a number of microseconds.)

GMTTIMESTAMP

A value of the DATE datatype is mapped to a string representation of the form 'YYYY-MM-DD HH:MM:SS[.UUUUUU]'. (UUUUUU indicates a number of microseconds.)

Interval

A value of the Interval datatype is mapped to a string representation, allowing filters such as Body.Date1='2002-01 YEAR TO MONTH' to be specified.

Bit Array

A value of the Bit Array datatype is mapped to a String of 0's and 1's which may be matched lexicographically against filters that include equivalent strings of 0's and 1's. There is no restriction that the number of 0's and 1's in the sequence.

Byte Arrays

A value of the Byte Array datatype is mapped to a string representation, so that, for example, a Byte Array with value 0x0102 is mapped into the ASCII string '0102'). This allows Byte Array values to be matched against filter expression string literals (for example, a filter that specified Body.Field1='0102' will match a Byte Array field value of 0x0102 in the message).

Implicit type casting

No explicit CAST operation is provided for use in CBR filters but a limited implicit casting scheme is provided. This is illustrated in the table below.

Filter Expression	Publication Content	Implicit Cast
Integer Literal, for example: Field1=100 or Field2>100	String, for example: Field1 is '100'	The publication content will be cast to an Integer, provided it is numeric and contains no decimal points or exponent.
Float Literal, for example: Field1=1.78E+11 or Field2>1.78E+11	String, for example: Field1 is '1.78E+11'	The publication content may be cast to a Float, provided it is numeric.
Float Literal, for example: Field1=1.78E+2 or Field2<1.78E+2	Integer, for example: Field1 is 178	The publication content may be cast to a Float.

Filter Expression	Publication Content	Implicit Cast
Integer Literal, for example: Field1=178 or Field2>178	Float, for example: Field1 is 1.78E+2	The publication content may be cast to an Integer. An integer comparison is always performed in this case.
A boolean literal or expression, for example: Field1=TRUE or Field2	String, for example: Field1 is 'True' or Field2 is 'True'	The publication content may be cast to an Boolean, provided it is a string value of 'True' or 'False' (irrespective of case).
A String literal	Another type	The publication content may be cast to a string (a lexicographical comparison is made).

Error reporting and logging

The CBR Matching Engine will report errors found in a publication by returning a Response message with code `MQRCCF_FILTER_ERROR` to the publisher. For example, a message format error that is found when parsing a message for specific fields to match will be reported in this way.

Similarly, where the CBR Matching engine determines that a subscriber's filter is in error, a Response message with code `MQRCCF_FILTER_ERROR` will be returned to the subscriber.

There are, however, a class of errors that will be found in subscriptions only at publication time. Such errors will be reported in the event log. One example of this kind of error would be where a filter sought to apply an arithmetic operation to a field in a message of String type that could not be implicitly cast to a numeric.

Rounding errors and overflows

The CBR Matching Engine will disallow filter expressions that violate the boundaries of the numeric datatypes returning a message with an `MQRCCF_FILTER_ERROR` to the subscriber. It does not, however, check numeric datatype overflows that result from arithmetic expressions (for example, `Field1+Field2*Field3>23E+200`). In such cases, the results are unpredictable.

The CBR Float datatype can support 15 digits precision. Special care should be taken when performing arithmetic on floats because individual rounding errors rapidly become compounded.

Appendix B. Plug-in node sample code

This appendix documents the following sample Java files referenced in Chapter 6, “Implementing a plug-in node or parser” on page 63:

- “The sample property editor `TraceSettingPropertyEditor.java`”
- “The sample customizer files” on page 173

The sample property editor `TraceSettingPropertyEditor.java`

```
package com.isv;

// *****
// Source File Name: TraceSettingPropertyEditor.java
//
// IBM MQSeries Integrator Version 2.0 - 5639-F61
// (C) Copyright IBM Corp. 2000
// *****

import java.beans.*;
import java.util.*;
import org.w3c.dom.*;
import java.awt.*;

/**
 * This is a sample ISV Property Editor to show the use of the
 * PropertyEditor and org.w3c.dom.Element interfaces
 */
public class TraceSettingPropertyEditor implements PropertyEditor {
    private final static String copyright = "Licensed Material
        - Property of IBM \nIBM MQSeries Integrator Version 2.0
        - 5639-F61 \n(C) Copyright IBM Corp. 2000
        - All Rights Reserved. \nUS Government Users Restricted Rights
        - Use, duplication or disclosure \nrestricted by GSA ADP
        Schedule Contract with IBM Corp.";
    Vector propertyChangeListener = new Vector();
    Element element = null;
    /**
     * TraceSettingPropertyEditor constructor
     */
    public TraceSettingPropertyEditor() {
        super();
    }
    /**
     * addPropertyChangeListener method
     */
    public void
        addPropertyChangeListener(PropertyChangeListener listener) {
        propertyChangeListener.addElement(listener);
    }
    /**
     * getAsText method
     */
    public String getAsText() {
        return element.getAttribute("value");
    }
    /**
     * getCustomEditor method
     */
    public Component getCustomEditor() {
        return null;
    }
    /**
     * getJavaInitializationString method
     */
    public String getJavaInitializationString() {
        return null;
    }
}
```

Property editor sample

```
}
/**
 * getTags method
 */
public java.lang.String[] getTags() {
    String[] tags = {"0","1","2"};
    return tags;
}
/**
 * getValue method
 */
public Object getValue() {
    return element;
}
/**
 * isPaintable method
 */
public boolean isPaintable() {
    return false;
}
protected void notifyListeners(String oldValue)
{
    for(int idx = 0; idx < propertyChangeListeners.size(); idx++) {
        PropertyChangeEvent event = new PropertyChangeEvent(this,
            element.getAttribute("xml.label"),
            oldValue, element.getAttribute("value"));
        ((PropertyChangeListener)propertyChangeListeners.
            elementAt(idx)).propertyChange(event);
    }
}
/**
 * paintValue method
 */
public void
    paintValue(java.awt.Graphics gfx, java.awt.Rectangle box) {}
/**
 * removePropertyChangeListener method
 */
public void
    removePropertyChangeListener(PropertyChangeListener listener) {
        propertyChangeListeners.removeElement(listener);
    }
/**
 */
public void setAsText(String value) {
    String oldValue = element.getAttribute("value");
    element.setAttribute("value",value);
    notifyListeners(oldValue);
}
/**
 * setValue method
 */
public void setValue(Object value) {
    String oldValue = null;
    if(element != null) {
        oldValue = element.getAttribute("value");
    }
    element = (Element)value;
    notifyListeners(oldValue);
}
/**
 * supportsCustomEditor method
 */
public boolean supportsCustomEditor() {
    return false;
}
}
```

The sample customizer files

Sample customizer file SampleSwitchCustomizer.java

```

package com.isv;

// *****
// Source File Name: SampleSwitchCustomizer.java
//
// IBM MQSeries Integrator Version 2.0 - 5639-F61
// (C) Copyright IBM Corp. 2000
// *****

import java.beans.*;
import com.ibm.xml.parser.*;
import org.w3c.dom.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This is an example customizer for the SampleSwitch example
 * MessageProcessingNodeType
 */
public class SampleSwitchCustomizer extends Panel implements
    java.beans.Customizer,
    java.awt.event.ItemListener,
    java.awt.event.ActionListener {
    private final static String copyright = "Licensed Material
    - Property of IBM \nIBM MQSeries Integrator Version 2.0
    - 5639-F61 \n(C) Copyright IBM Corp. 2000
    - All Rights Reserved.
    \nUS Government Users Restricted Rights
    - Use, duplication or disclosure \nrestricted by GSA ADP
    Schedule Contract with IBM Corp.";

    //listeners interested when a value changes
    private Vector propertyChangeListeners = new Vector();

    //group of buttons for setting the nodeTraceSetting
    private ButtonGroup debugButtonGroup = null;
    private JRadioButton debugOff = null;
    private JRadioButton debugNormal = null;
    private JRadioButton debugDebug = null;

    //panels to display controls on
    private JPanel ivjDebugPanel = null;
    private JPanel ivjFilePanel = null;

    //the XML elements that are of interest
    private Element TraceSettingAttribute = null;
    private Element TraceFileAttribute = null;
    private Element rootNode = null;

    //group of objects for setting the nodeTraceOutfile
    private JButton changeFileButton = null;
    private static String currentDir = System.getProperty("user.dir");
    private static Component parentFrame;
    private JLabel currentFile;

    //constants values
    private final static String TraceSettingAttributeName = "nodeTraceSetting";
    private final static String TraceFileAttributeName = "nodeTraceOutfile";
    private final static int OFF = 0;
    private final static int NORMAL = 1;
    private final static int DEBUG = 2;
}

```

Customizer sample

```
* SampleSwitchCustomizer constructor
*/
public SampleSwitchCustomizer() {
    super();
    initialize();
}
/*
 * actions to perform when the user pushes the 'change trace file' button
 * @param java.awt.event.ActionEvent
 */
public void actionPerformed(ActionEvent e) {
    if (e.getSource().equals(changeFileButton)) {
        setTraceFile();
    }
}
/**
 * addPropertyChangeListener method
 */
public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeListeners.addElement(listener);
}
/**
 * one of the debug radio buttons was selected -> setTraceSetting()
 */

private void changeMode() {
    try {
        if(debugOff.isSelected()) setTraceSetting(OFF);
        else if (debugNormal.isSelected()) setTraceSetting(NORMAL);
        else setTraceSetting(DEBUG);
    } catch (Throwable ivjExc) {
        handleException(ivjExc);
    }
}
/*
 * create an Attribute and return it
 * @params String, String, String
 * @return org.w3c.dom.Element
 */
private Element createAttribute(String name, String value, String type)
{
    try {
        Element attribute = rootNode.getOwnerDocument().createElement("Attribute");

        attribute.setAttribute("xml:label", name);
        attribute.setAttribute("value", value);
        attribute.setAttribute("type",type);

        return attribute;

    } catch (Throwable ivjExc) {
        handleException(ivjExc);
        return null;
    }
}
/**
 * Return the DebugPanel property value.
 * @return javax.swing.JPanel
 */
private JPanel getDebugModePanel() {
    if (ivjDebugPanel == null) {
        try {
            ivjDebugPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
            ivjDebugPanel.setName("ivjDebugPanel");

            ivjDebugPanel.add(new JLabel("Trace Mode: "));
            ivjDebugPanel.setBorder(new javax.swing.border.EtchedBorder());

            debugButtonGroup = new ButtonGroup();
            debugOff = new JRadioButton("Off");
            debugOff.setSelected(true);
        }
    }
}
```

```

        ivjDebugPanel.add(debugOff);
        debugButtonGroup.add(debugOff);

        debugNormal = new JRadioButton("Normal");
        debugNormal.setSelected(false);
        ivjDebugPanel.add(debugNormal);
        debugButtonGroup.add(debugNormal);

        debugDebug = new JRadioButton("Debug");
        debugDebug.setSelected(false);
        ivjDebugPanel.add(debugDebug);
        debugButtonGroup.add(debugDebug);

    } catch (Throwable ivjExc) {
        handleException(ivjExc);
    }
}
return ivjDebugPanel;
}
/**
 * Return the DebugFilePanel property value.
 * @return javax.swing.JPanel
 */
private JPanel getTraceFilePanel(Frame parent) {
    if (ivjFilePanel == null) {
        try {
            GridBagConstraints gc = new GridBagConstraints();
            GridBagLayout layout = new GridBagLayout();

            ivjFilePanel = new JPanel();
            ivjFilePanel.setLayout(layout);
            parentFrame = parent;
            changeFileButton = new JButton("Change Trace File");
            currentFile = new JLabel("");
            JLabel title = new JLabel("Trace File:");

            gc.gridx = 0;
            gc.gridy = 0;
            gc.weightx = 0;
            gc.weighty = 0;
            gc.fill = gc.NONE;
            gc.anchor = gc.WEST;
            gc.insets = new Insets(10,0,10,10);
            ivjFilePanel.add(title);
            layout.setConstraints(title, gc);

            gc.gridx = 1;
            gc.weightx = 1;
            gc.insets = new Insets(10,0,10,10);
            ivjFilePanel.add(currentFile);
            layout.setConstraints(currentFile, gc);

            gc.gridx = 0;
            gc.gridy = 1;
            gc.gridwidth = 2;
            gc.insets = new Insets(0,0,10,10);
            ivjFilePanel.add(changeFileButton);
            layout.setConstraints(changeFileButton, gc);

        } catch (Throwable ivjExc) {
            handleException(ivjExc);
        }
    }
    return ivjFilePanel;
}
/**
 * show the current value for the trace file
 * @param setting String
 */
private void getTraceFileSetting(String setting)
{

```

Customizer sample

```
        currentFile.setText(setting);
    }
    /**
     * @param setting String
     */
    private void getTraceSetting(String setting)
    {
        //set the correct radioButton based on the passed in value
        int newSetting = Integer.parseInt(setting);
        try {
            if(newSetting == DEBUG) debugDebug.setSelected(true);
            else if(newSetting == NORMAL) debugNormal.setSelected(true);
            else debugOff.setSelected(true);
        }
        catch (NumberFormatException nfe) {
            //the parameter passed in is not a valid integer, so just set
            //the default value
            debugOff.setSelected(true);
        }
    }
    /**
     * Called whenever the part throws an exception.
     * @param exception java.lang.Throwable
     */
    private void handleException(Throwable exception) {

        /* Uncomment the following lines to print uncaught exceptions to stdout */
        // System.out.println("----- UNCAUGHT EXCEPTION -----");
        // exception.printStackTrace(System.out);
    }
    /**
     * Initializes connections
     * @exception java.lang.Exception The exception description.
     */

    private void initConnections() throws Exception
    {

        try {
            debugOff.addItemListener(this);
            debugNormal.addItemListener(this);
            debugDebug.addItemListener(this);
            changeFileButton.addActionListener(this);
        } catch (Throwable ivjExc) {
            handleException(ivjExc);
        }

    }
    /**
     * Initialize the class.
     */

    private void initialize() {
        try {

            setName("SampleSwitchCustomizer");
            GridBagConstraints gc = new GridBagConstraints();
            GridBagLayout layout = new GridBagLayout();
            setLayout(layout);
            setSize(320, 181);
            setEnabled(true);

            gc.gridx = 0;
            gc.gridy = 0;
            gc.weightx = 1;
            gc.weighty = 0;
            gc.fill = gc.NONE;
            gc.anchor = gc.WEST;
            gc.insets = new Insets(0,10,10,0);

            //add the panel containing the trace file controls
            Frame tf = new Frame();
```

```

add(getTraceFilePanel(tf), getTraceFilePanel(tf).getName());
layout.setConstraints(getTraceFilePanel(tf), gc);
gc.gridy = 1;
//add the panel containing the debug mode controls
add(getDebugModePanel(), getDebugModePanel().getName());
layout.setConstraints(getDebugModePanel(), gc);

initConnections();
} catch (Throwable ivjExc) {
    handleException(ivjExc);
}
}
//actions to perform when the user changes the trace mode
public void itemStateChanged(ItemEvent e) {
    if(e.getSource().equals(debugOff) ]]
        e.getSource().equals(debugNormal) ]]
        e.getSource().equals(debugDebug) {
        changeMode();
    }
}
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(String[] args) {
    try {
        Frame frame = new Frame();
        SampleSwitchCustomizer aSampleSwitchCustomizer;
        aSampleSwitchCustomizer = new SampleSwitchCustomizer();
        frame.add("Center", aSampleSwitchCustomizer);
        frame.setSize(aSampleSwitchCustomizer.getSize());
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.setVisible(true);
    } catch (Throwable exception) {
        System.err.println("Exception occurred in main()
            of com.isv.SampleSwitchCustomizer");
        exception.printStackTrace(System.out);
    }
}
//send PropertyChangeEvent to objects interested that a value in this customizer
//has been changed
private void notifyListeners(String oldValue, String newValue)
{
    for(int idx = 0; idx < propertyChangeListenerListners.size(); idx++) {
        PropertyChangeEvent event =
            new PropertyChangeEvent(this, null, oldValue, newValue);
        ((PropertyChangeListener)propertyChangeListenerListners.elementAt(idx)).propertyChange(event);
    }
}
/**
 * removePropertyChangeListener method
 */
public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeListenerListners.removeElement(listener);
}
/**
 * setObject method
 * @param Object - the node that this customizer is being launched from
 */
public void setObject(Object bean) {
    rootNode = (Element) bean;

    // get Attributes for this node
    NodeList Attributes = rootNode.getElementsByTagName("Attribute");
    Element typeref = null;

    for(int idx = 0; idx < Attributes.getLength(); idx++) {
        Element attribute = (Element)Attributes.item(idx);

```

Customizer sample

```
        if(attribute.getAttribute("xmi.label").equals(TraceFileAttributeName)) {
            TraceFileAttribute = attribute;
        }
        if(attribute.getAttribute("xmi.label").equals(TraceSettingAttributeName)) {
            TraceSettingAttribute = attribute;
        }
    }

    //if we did not find any/all of the attributes for this node, create them
    typeref = (Element) rootNode.getLastChild();
    if (TraceSettingAttribute == null) {
        //create TraceSettingAttribute and add it to the node
        TraceSettingAttribute =
            createAttribute(TraceSettingAttributeName,Integer.toString(OFF),"Integer");
        rootNode.insertBefore((Node) TraceSettingAttribute, typeref);
    }
    if (TraceFileAttribute == null) {
        //create TraceFileAttribute and add it to the node
        TraceFileAttribute = createAttribute(TraceFileAttributeName,"","String");
        rootNode.insertBefore((Node) TraceFileAttribute, typeref);
    }

    //it is assumed that at this point, instance variables TraceFileAttribute and
    //TraceSettingAttribute are non-null. The code does not handle them = null

    //get the initial values for the attributes
    getTraceSetting(TraceSettingAttribute.getAttribute("value"));
    getTraceFileSetting(TraceFileAttribute.getAttribute("value"));
}
/**
 * bring up a file dialog and capture the chosen result.
 */
private void setTraceFile()
{
    //create file dialog and show it
    FileDialog dialog = new FileDialog((Frame) parentFrame,
        "Change File", FileDialog.LOAD);
    dialog.setDirectory(currentDir);
    dialog.show();

    //get the chosen file. If it is null, then the user pressed 'Cancel'
    String file = dialog.getFile();
    if(file != null) {
        currentFile.setText(dialog.getDirectory() + dialog.getFile());
    }

    //save the directory setting for the future
    currentDir = dialog.getDirectory();
    setTraceFileSetting(currentFile.getText());
}
/**
 * @param setting String
 */
private void setTraceFileSetting(String setting)
{
    //set Attribute value and notify listeners

    String oldValue = TraceFileAttribute.getAttribute("value");
    if(!oldValue.equals(setting)) {
        //only if the value has changed
        TraceFileAttribute.setAttribute("value",setting);
        notifyListeners(oldValue, setting);
    }
}
/**
 * @param setting int
 */
private void setTraceSetting(int setting)
{
    //set Attribute value and notify listeners
```

```
String oldValue = TraceSettingAttribute.getAttribute("value");
String newValue = Integer.toString(setting);
if(!oldValue.equals(newValue)) {
    //only if the value has changed
    TraceSettingAttribute.setAttribute("value",newValue);
    notifyListeners(oldValue, newValue);
}
}
```

Sample customizer file SampleSwitchCustomizerBeanInfo.java

```

package com.isv;

// *****
// Source File Name: SampleSwitchCustomizerBeanInfo.java
//
// IBM MQSeries Integrator Version 2.0 - 5639-F61
// (C) Copyright IBM Corp. 2000
// *****
/**
 * The bean information class for com.isv.SampleSwitchCustomizer.
 */
public class
    SampleSwitchCustomizerBeanInfo extends java.beans.SimpleBeanInfo {
    private final static String copyright = "Licensed Material
    - Property of IBM \nIBM MQSeries Integrator Version 2.0
    - 5639-F61 \n(C) Copyright IBM Corp. 2000
    - All Rights Reserved.
    \nUS Government Users Restricted Rights
    - Use, duplication or disclosure \nrestricted by GSA ADP
    Schedule Contract with IBM Corp.";
/**
 * Gets the addPropertyChangeListener(java.beans.PropertyChangeListener)
 * method descriptor.
 * @return java.beans.MethodDescriptor
 */
public java.beans.MethodDescriptor
    addPropertyChangeListener_javabeansPropertyChangeListenerMethodDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /**
         * Create and return the
         * addPropertyChangeListener(java.beans.PropertyChangeListener)
         * method descriptor.
         */
        java.lang.reflect.Method aMethod = null;
        try {
            /**
             * Attempt to find the method using getMethod with
             * parameter types.
             */
            java.lang.Class aParameterTypes[] = {
                java.beans.PropertyChangeListener.class
            };
            aMethod = getBeanClass().getMethod("addPropertyChangeListener",
                aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod(getBeanClass(), "addPropertyChangeListener", 1);
        };
        try {
            /* Try creating the method descriptor with parameter descriptors. */
            java.beans.ParameterDescriptor aParameterDescriptor1 =
                new java.beans.ParameterDescriptor();
            aParameterDescriptor1.setName("arg1");
            aParameterDescriptor1.setDisplayName("listener");
            java.beans.ParameterDescriptor aParameterDescriptors[] = {
                aParameterDescriptor1
            };
            aDescriptor = new java.beans.MethodDescriptor(aMethod,aParameterDescriptors);
        } catch (java.lang.Throwable exception) {
            /* Try creating the method descriptor without parameter descriptors. */
            handleException(exception);
            aDescriptor = new java.beans.MethodDescriptor(aMethod);
        };
        /* aDescriptor.setDisplayName("addPropertyChangeListener
        (java.beans.PropertyChangeListener)"); */
        /* aDescriptor.setShortDescription("addPropertyChangeListener
        (java.beans.PropertyChangeListener)"); */
        /* aDescriptor.setExpert(false); */
    }
}

```



```

        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return aDescriptor;
}
/**
 * Gets the checked(java.awt.Checkbox) method descriptor.
 * @return java.beans.MethodDescriptor
 */
public java.beans.MethodDescriptor
    checked_javaawtCheckboxMethodDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /* Create and return the checked(java.awt.Checkbox) method descriptor. */
        java.lang.reflect.Method aMethod = null;
        try {
            /* Attempt to find the method using getMethod with parameter types. */
            java.lang.Class aParameterTypes[] = {
                java.awt.Checkbox.class
            };
            aMethod = getBeanClass().getMethod("checked", aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod(getBeanClass(), "checked", 1);
        };
        try {
            /* Try creating the method descriptor with parameter descriptors. */
            java.beans.ParameterDescriptor aParameterDescriptor1 =
                new java.beans.ParameterDescriptor();
            aParameterDescriptor1.setName("arg1");
            aParameterDescriptor1.setDisplayName("arg1");
            java.beans.ParameterDescriptor aParameterDescriptors[] = {
                aParameterDescriptor1
            };
            aDescriptor = new java.beans.MethodDescriptor(aMethod, aParameterDescriptors);
        } catch (java.lang.Throwable exception) {
            /* Try creating the method descriptor without parameter descriptors. */
            handleException(exception);
            aDescriptor = new java.beans.MethodDescriptor(aMethod);
        };
        /* aDescriptor.setDisplayName("checked(java.awt.Checkbox)"); */
        /* aDescriptor.setShortDescription("checked(java.awt.Checkbox)"); */
        /* aDescriptor.setExpert(false); */
        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return aDescriptor;
}
/**
 * Find the method by comparing (name & parameter size) against
 * the methods in the class.
 * @return java.lang.reflect.Method
 * @param aClass java.lang.Class
 * @param methodName java.lang.String
 * @param parameterCount int
 */
public static java.lang.reflect.Method
    findMethod(java.lang.Class aClass, java.lang.String methodName, int parameterCount) {
    try {
        /**
         * Since this method attempts to find a method by getting all
         * methods from the class, this method should only be called
         * if getMethod cannot find the method.
         */
        java.lang.reflect.Method methods[] = aClass.getMethods();
        for (int index = 0; index < methods.length; index++){
            java.lang.reflect.Method method = methods[index];

```

Customizer sample

```
        if ((method.getParameterTypes().length == parameterCount)
            && (method.getName().equals(methodName))) {
            return method;
        }
    }
} catch (java.lang.Throwable exception) {
    return null;
}
return null;
}
/**
 * Returns the BeanInfo of the superclass of this bean
 * to inherit its features.
 * @return java.beans.BeanInfo£
 */
public java.beans.BeanInfo£ getAdditionalBeanInfo() {
    java.lang.Class superClass;
    java.beans.BeanInfo superBeanInfo = null;

    try {
        superClass = getBeanDescriptor().getBeanClass().getSuperclass();
    } catch (java.lang.Throwable exception) {
        return null;
    }

    try {
        superBeanInfo = java.beans.Introspector.getBeanInfo(superClass);
    } catch (java.beans.IntrospectionException ie) {}

    if (superBeanInfo != null) {
        java.beans.BeanInfo£ ret = new java.beans.BeanInfo£1£;
        ret£0£ = superBeanInfo;
        return ret;
    }
    return null;
}
/**
 * Gets the bean class.
 * @return java.lang.Class
 */
public static java.lang.Class getBeanClass() {
    return com.isv.SampleSwitchCustomizer.class;
}
/**
 * Gets the bean class name.
 * @return java.lang.String
 */
public static java.lang.String getBeanClassName() {
    return "com.isv.SampleSwitchCustomizer";
}
public java.beans.BeanDescriptor getBeanDescriptor() {
    java.beans.BeanDescriptor aDescriptor = null;
    try {
        /**
         * Create and return the SampleSwitchCustomizerBeanInfo
         * bean descriptor.
         */
        aDescriptor =
            new java.beans.BeanDescriptor(com.isv.SampleSwitchCustomizer.class);
        /* aDescriptor.setExpert(false); */
        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("hidden-state", Boolean.FALSE); */
    } catch (Throwable exception) {
    };
    return aDescriptor;
}
/**
 * Return the event set descriptors for this bean.
 * @return java.beans.EventSetDescriptor£
 */
public java.beans.EventSetDescriptor£ getEventSetDescriptors() {
    try {
```

```

        java.beans.EventSetDescriptor aDescriptorList£ = {
            propertyChangeEventSetDescriptor()
        };
    return aDescriptorList;
} catch (java.lang.Throwable exception) {
    handleException(exception);
};
return null;
}
/**
 * Return the method descriptors for this bean.
 * @return java.beans.MethodDescriptor£
 */
public java.beans.MethodDescriptor£ getMethodDescriptors() {
    try {
        java.beans.MethodDescriptor aDescriptorList£ = {
            addPropertyChangeListener_javabeansPropertyChangeListenerMethodDescriptor()
            ,checked_javaawtCheckboxMethodDescriptor()
            ,main_javalangString__MethodDescriptor()
            ,removePropertyChangeListener_javabeansPropertyChangeListenerMethodDescriptor()
            ,setObject_javalangObjectMethodDescriptor()
        };
        return aDescriptorList;
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return null;
}
/**
 * Return the property descriptors for this bean.
 * @return java.beans.PropertyDescriptor£
 */
public java.beans.PropertyDescriptor£ getPropertyDescriptors() {
    try {
        java.beans.PropertyDescriptor aDescriptorList£ = {
            objectPropertyDescriptor()
        };
        return aDescriptorList;
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return null;
}
/**
 * Called whenever the bean information class throws an exception.
 * @param exception java.lang.Throwable
 */
private void handleException(java.lang.Throwable exception) {

    /* Uncomment the following lines to print uncaught exceptions to stdout */
    // System.out.println("----- UNCAUGHT EXCEPTION -----");
    // exception.printStackTrace(System.out);
}
/**
 * Gets the main(java.lang.String£) method descriptor.
 * @return java.beans.MethodDescriptor
 */
public java.beans.MethodDescriptor main_javalangString__MethodDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /* Create and return the main(java.lang.String£) method descriptor. */
        java.lang.reflect.Method aMethod = null;
        try {
            /* Attempt to find the method using getMethod with parameter types. */
            java.lang.Class aParameterTypes£ = {
                java.lang.String£.class
            };
            aMethod = getBeanClass().getMethod("main", aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod(getBeanClass(), "main", 1);
        }
    }
}

```

Customizer sample

```
};
try {
    /* Try creating the method descriptor with parameter descriptors. */
    java.beans.ParameterDescriptor aParameterDescriptor1 =
        new java.beans.ParameterDescriptor();
    aParameterDescriptor1.setName("arg1");
    aParameterDescriptor1.setDisplayName("args");
    java.beans.ParameterDescriptor aParameterDescriptors[] = {
        aParameterDescriptor1
    };
    aDescriptor = new java.beans.MethodDescriptor(aMethod, aParameterDescriptors);
} catch (java.lang.Throwable exception) {
    /* Try creating the method descriptor without parameter descriptors. */
    handleException(exception);
    aDescriptor = new java.beans.MethodDescriptor(aMethod);
};
/* aDescriptor.setDisplayName("main(java.lang.String[])"); */
/* aDescriptor.setShortDescription("main(java.lang.String[])"); */
/* aDescriptor.setExpert(false); */
/* aDescriptor.setHidden(false); */
/* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
} catch (java.lang.Throwable exception) {
    handleException(exception);
};
return aDescriptor;
}
/**
 * Gets the object property descriptor.
 * @return java.beans.PropertyDescriptor
 */
public java.beans.PropertyDescriptor objectPropertyDescriptor() {
    java.beans.PropertyDescriptor aDescriptor = null;
    try {
        try {
            /* Using methods via getMethod is the faster way to
             * create the object property descriptor.
             */
            java.lang.reflect.Method aGetMethod = null;
            java.lang.reflect.Method aSetMethod = null;
            try {
                /* Attempt to find the method using getMethod with parameter types. */
                java.lang.Class aSetMethodParameterTypes[] = {
                    java.lang.Object.class
                };
                aSetMethod = getBeanClass().getMethod("setObject", aSetMethodParameterTypes);
            } catch (java.lang.Throwable exception) {
                /* Since getMethod failed, call findMethod. */
                handleException(exception);
                aSetMethod = findMethod(getBeanClass(), "setObject", 1);
            };
            aDescriptor = new java.beans.PropertyDescriptor("object"
                , aGetMethod, aSetMethod);
        } catch (java.lang.Throwable exception) {
            /* Since we failed using methods, try creating a
             default property descriptor. */
            handleException(exception);
            aDescriptor = new java.beans.PropertyDescriptor("object"
                , getBeanClass());
        };
        aDescriptor.setBound(true);
        /* aDescriptor.setConstrained(false); */
        /* aDescriptor.setDisplayName("object"); */
        /* aDescriptor.setShortDescription("object"); */
        /* aDescriptor.setExpert(false); */
        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
        /* aDescriptor.setValue("ivjDesignTimeProperty",
            new java.lang.Boolean(true)); */
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
};
```

```

return aDescriptor;
}
/**
 * Gets the propertyChange event set descriptor.
 * @return java.beans.EventSetDescriptor
 */
public java.beans.EventSetDescriptor propertyChangeEventSetDescriptor() {
    java.beans.EventSetDescriptor aDescriptor = null;
    try {
        try {
            /* Try using method descriptors to create the propertyChange
            event set descriptor. */
            java.beans.MethodDescriptor eventMethodDescriptors£ = {
                propertyChangePropertyChange_javabeansPropertyChangeEventMethodEventDescriptor()
            };
            java.lang.reflect.Method anAddMethod = null;
            try {
                /* Attempt to find the method using getMethod with parameter types. */
                java.lang.Class anAddMethodParameterTypes£ = {
                    java.beans.PropertyChangeListener.class
                }
                anAddMethod = getBeanClass().getMethod("addPropertyChangeListener",
                    anAddMethodParameterTypes);
            } catch (java.lang.Throwable exception) {
                /* Since getMethod failed, call findMethod. */
                handleException(exception);
                anAddMethod = findMethod(getBeanClass(),
                    "addPropertyChangeListener", 1);
            };
            java.lang.reflect.Method aRemoveMethod = null;
            try {
                /* Attempt to find the method using getMethod with parameter types. */
                java.lang.Class aRemoveMethodParameterTypes£ = {
                    java.beans.PropertyChangeListener.class
                };
                aRemoveMethod = getBeanClass().getMethod("removePropertyChangeListener",
                    aRemoveMethodParameterTypes);
            } catch (java.lang.Throwable exception) {
                /* Since getMethod failed, call findMethod. */
                handleException(exception);
                aRemoveMethod = findMethod(getBeanClass(),
                    "removePropertyChangeListener", 1);
            };
            aDescriptor = new java.beans.EventSetDescriptor(
                "propertyChange",
                java.beans.PropertyChangeListener.class,
                eventMethodDescriptors, anAddMethod, aRemoveMethod);
        } catch (java.lang.Throwable exception) {
            /* Using method descriptors failed, try using the methods names. */
            handleException(exception);
            java.lang.String eventMethodNames£ = {
                "propertyChange"
            };
            aDescriptor = new java.beans.EventSetDescriptor(getBeanClass(),
                "propertyChange",
                java.beans.PropertyChangeListener.class,
                eventMethodNames,
                "addPropertyChangeListener",
                "removePropertyChangeListener");
        };
        /* aDescriptor.setUnicast(false); */
        /* aDescriptor.setDisplayName("propertyChange"); */
        /* aDescriptor.setShortDescription("propertyChange"); */
        /* aDescriptor.setExpert(false); */
        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return aDescriptor;
}
/**

```

Customizer sample

```
* Gets the propertyChange.propertyChange(java.beans.PropertyChangeEvent)
* method descriptor.
* @return java.beans.MethodDescriptor
*/
public java.beans.MethodDescriptor
propertyChangePropertyChange_javabeansPropertyChangeEventMethodEventDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /* Create and return the propertyChange.propertyChange(java.beans.PropertyChangeEvent)
        method descriptor. */
        java.lang.reflect.Method aMethod = null;
        try {
            /* Attempt to find the method using getMethod with parameter types. */
            java.lang.Class aParameterTypes[] = {
                java.beans.PropertyChangeEvent.class
            };
            aMethod = (java.beans.PropertyChangeListener.class).getMethod("propertyChange",
                aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod((java.beans.PropertyChangeListener.class),
                "propertyChange", 1);
        };
        try {
            /* Try creating the method descriptor with parameter descriptors. */
            java.beans.ParameterDescriptor aParameterDescriptor1 =
                new java.beans.ParameterDescriptor();
            aParameterDescriptor1.setName("arg1");
            aParameterDescriptor1.setDisplayName("evt");
            java.beans.ParameterDescriptor aParameterDescriptors[] = {
                aParameterDescriptor1
            };
            aDescriptor = new java.beans.MethodDescriptor(aMethod, aParameterDescriptors);
        } catch (java.lang.Throwable exception) {
            /* Try creating the method descriptor without parameter descriptors. */
            handleException(exception);
            aDescriptor = new java.beans.MethodDescriptor(aMethod);
        };
        /* aDescriptor.setDisplayName("propertyChange"); */
        /* aDescriptor.setShortDescription("propertyChange"); */
        /* aDescriptor.setExpert(false); */
        /* aDescriptor.setHidden(false); */
        /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
    } catch (java.lang.Throwable exception) {
        handleException(exception);
    };
    return aDescriptor;
}
/**
* Gets the removePropertyChangeListener(java.beans.PropertyChangeListener)
* method descriptor.
* @return java.beans.MethodDescriptor
*/
public java.beans.MethodDescriptor
removePropertyChangeListener_javabeansPropertyChangeListenerMethodDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /* Create and return the removePropertyChangeListener
        (java.beans.PropertyChangeListener) method descriptor. */
        java.lang.reflect.Method aMethod = null;
        try {
            /* Attempt to find the method using getMethod with parameter types. */
            java.lang.Class aParameterTypes[] = {
                java.beans.PropertyChangeListener.class
            };
            aMethod = getBeanClass().getMethod("removePropertyChangeListener",
                aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod(getBeanClass(), "removePropertyChangeListener", 1);
        };
    }
}
```

```

};
try {
    /* Try creating the method descriptor with parameter descriptors. */
    java.beans.ParameterDescriptor aParameterDescriptor1 =
        new java.beans.ParameterDescriptor();
    aParameterDescriptor1.setName("arg1");
    aParameterDescriptor1.setDisplayName("listener");
    java.beans.ParameterDescriptor aParameterDescriptors[] = {
        aParameterDescriptor1
    };
    aDescriptor = new java.beans.MethodDescriptor(aMethod, aParameterDescriptors);
} catch (java.lang.Throwable exception) {
    /* Try creating the method descriptor without parameter descriptors. */
    handleException(exception);
    aDescriptor = new java.beans.MethodDescriptor(aMethod);
};
/* aDescriptor.setDisplayName("removePropertyChangeListener
    (java.beans.PropertyChangeListener)"); */
/* aDescriptor.setShortDescription("removePropertyChangeListener
    (java.beans.PropertyChangeListener)"); */
/* aDescriptor.setExpert(false); */
/* aDescriptor.setHidden(false); */
/* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
} catch (java.lang.Throwable exception) {
    handleException(exception);
};
return aDescriptor;
}
/**
 * Gets the setObject(java.lang.Object) method descriptor.
 * @return java.beans.MethodDescriptor
 */
public java.beans.MethodDescriptor setObject_javaLangObjectMethodDescriptor() {
    java.beans.MethodDescriptor aDescriptor = null;
    try {
        /* Create and return the setObject(java.lang.Object) method descriptor. */
        java.lang.reflect.Method aMethod = null;
        try {
            /* Attempt to find the method using getMethod with parameter types. */
            java.lang.Class aParameterTypes[] = {
                java.lang.Object.class
            };
            aMethod = getBeanClass().getMethod("setObject", aParameterTypes);
        } catch (java.lang.Throwable exception) {
            /* Since getMethod failed, call findMethod. */
            handleException(exception);
            aMethod = findMethod(getBeanClass(), "setObject", 1);
        };
    }
    try {
        /* Try creating the method descriptor with parameter descriptors. */
        java.beans.ParameterDescriptor aParameterDescriptor1 =
            new java.beans.ParameterDescriptor();
        aParameterDescriptor1.setName("arg1");
        aParameterDescriptor1.setDisplayName("bean");
        java.beans.ParameterDescriptor aParameterDescriptors[] = {
            aParameterDescriptor1
        };
        aDescriptor = new java.beans.MethodDescriptor(aMethod, aParameterDescriptors);
    } catch (java.lang.Throwable exception) {
        /* Try creating the method descriptor without parameter descriptors. */
        handleException(exception);
        aDescriptor = new java.beans.MethodDescriptor(aMethod);
    };
    /* aDescriptor.setDisplayName("setObject(java.lang.Object)"); */
    /* aDescriptor.setShortDescription("setObject(java.lang.Object)"); */
    /* aDescriptor.setExpert(false); */
    /* aDescriptor.setHidden(false); */
    /* aDescriptor.setValue("preferred", new java.lang.Boolean(false)); */
} catch (java.lang.Throwable exception) {
    handleException(exception);
};
};

```

Customizer sample

```
    return aDescriptor;  
  }  
}
```

Appendix C. Notices

This information was developed for products and services offered in the United States. 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, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

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

CICS
MQSeries

IBM
RETAIN

IMS
SupportPac

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Other company, product, and service names may be trademarks or service marks of others.

Part 4. Glossary and index

Glossary of terms and abbreviations

This glossary defines MQSeries Integrator terms and abbreviations used in this book. If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute. Copies may be ordered from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

Access Control List (ACL). The list of principals that have explicit permissions (to publish, to subscribe to, and to request persistent delivery of a publication message) against a topic in the topic tree. The ACLs define the implementation of topic-based security.

ACL. Access Control List.

AMI. Application Messaging Interface.

Application Messaging Interface (AMI). The programming interface provided by MQSeries that defines a high level interface to message queuing services. See also **MQI** and **JMS**.

B

blob. Binary Large Object. A block of bytes of data (for example, the body of a message) that has no discernible meaning, but is treated as one solid entity that cannot be interpreted. Also written as BLOB.

broker. See **message broker**.

broker domain. A collection of brokers that share a common configuration, together with the single Configuration Manager that controls them.

C

callback function. See *implementation function*.

category. An optional grouping of messages that are related in some way. For example, messages that relate to a particular application.

collective. A hyperconnected (totally connected) set of brokers forming part of a multi-broker network for publish/subscribe applications.

configuration. In the broker domain, the brokers, execution groups, message flows and message sets assigned to them, topics and access control specifications.

Configuration Manager. A component of MQSeries Integrator that acts as the interface between the configuration repository and an executing set of brokers. It provides brokers with their initial configuration, and updates them with any subsequent changes. It maintains the broker domain configuration.

configuration repository. Persistent storage for broker configuration and topology definition.

connector. See **message processing node connector**.

content-based filter. An expression that is applied to the content of a message to determine how the message is to be processed.

context tag. A tag that is applied to an element within a message to enable that element to be treated differently in different contexts. For example, an element could be mandatory in one context and optional in another.

Control Center. The graphical interface that provides facilities for defining, configuring, deploying, and monitoring resources of the MQSeries Integrator network.

D

datagram. The simplest form of message that MQSeries supports. Also known as *send-and-forget*. This type of message does not require a reply. Compare with *request/reply*.

deploy. Make operational the configuration and topology of the broker domain.

distribution list. A list of MQSeries queues to which a message can be put using a single statement.

E

e-business. A term describing the commercial use of the Internet and World Wide Web to conduct business (short for electronic-business).

element. A unit of data within a message that has business meaning, for example, street name

element qualifier • message queue interface (MQI)

element qualifier. See **context tag**.

execution group. A named grouping of message flows that have been assigned to a broker. The broker is guaranteed to enforce some degree of isolation between message flows in distinct execution groups by ensuring that they execute in separate address spaces, or as unique processes.

Extensible Markup Language (XML). A W3C standard for the representation of data.

F

filter. An expression that is applied to the content of a message to determine how the message is to be processed.

format. A format defines the internal structure of a message, in terms of the fields and order of those fields. A format can be self-defining, in which case the message is interpreted dynamically when read.

G

graphical user interface (GUI). An interface to a software product that is graphical rather than textual. It refers to window-based operational characteristics.

I

implementation function. Function written by a third-party developer for a plug-in node or parser. Also known as a *callback function*.

input node. A message flow node that represents a source of messages for the message flow.

installation mode. The installation mode can be Full, Custom, or Broker only. The mode defines the components of the product installed by the installation process.

J

Java Database Connectivity (JDBC). An application programming interface that has the same characteristics as **ODBC** but is specifically designed for use by Java database applications.

Java Development Kit (JDK). A software package that can be used to write, compile, debug, and run Java applets and applications.

Java Message Service (JMS). An application programming interface that provides Java language functions for handling messages.

Java Runtime Environment. A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes and supporting files.

JDBC. Java Database Connectivity.

JDK. Java Development Kit.

JMS. Java Message Service. See also **AMI** and **MQI**.

JRE. Java Runtime Environment.

M

message broker. A set of execution processes hosting one or more message flows.

messages. Entities exchanged between a broker and its clients.

message dictionary. A repository for (predefined) message type specifications.

message domain. The source of a message definition. For example, a domain of MRM identifies messages defined using the Control Center, a domain of NEON identifies messages created using the NEON user interfaces.

message flow. A directed graph that represents the set of activities performed on a message or event as it passes through a broker. A message flow consists of a set of message processing nodes and message processing node connectors.

message flow component. See **message flow**.

message parser. A program that interprets a message bitstream.

message processing node. A node in the message flow, representing a well defined processing stage. A message processing node can be one of several primitive types or can represent a subflow.

message processing node connector. An entity that connects the output terminal of one message processing node to the input terminal of another. A message processing node connector represents the flow of control and data between two message flow nodes.

message queue interface (MQI). The programming interface provided by MQSeries queue managers. The programming interface allows application programs to access message queuing services. See also **AMI** and **JMS**.

message repository. A database holding message template definitions.

message set. A grouping of related messages.

message template. A named and managed entity that represents the format of a particular message. Message templates represent a business asset of an organization.

message type. The logical structure of the data within a message. For example, the number and location of character strings.

metadata. Data that describes the characteristic of stored data.

MQI. Message queue interface.

MQRFH. An architected message header that is used to provide metadata for the processing of a message. This header is supported by MQSeries Publish/Subscribe.

MQRFH2. An extended version of MQRFH, providing enhanced function in message processing.

multi-level wildcard. A wildcard that can be specified in subscriptions to match any number of levels in a topic.

N

node. See **message processing node**.

O

ODBC. Open Database Connectivity.

Open Database Connectivity. A standard application programming interface (API) for accessing data in both relational and non-relational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group.

output node. A message processing node that represents a point at which messages flow out of the message flow.

P

plug-in. An extension to the broker, written by a third-party developer, to provide a new message processing node or message parser in addition to those supplied with the product. See also *implementation function* and *utility function*.

point-to-point. Style of messaging application in which the sending application knows the destination of the message. Compare with *publish/subscribe*.

predefined message. A message with a structure that is defined before the message is created or referenced. Compare with *self-defining message*.

primitive. A message processing node that is supplied with the product.

principal. An individual user ID (for example, a log-in ID) or a group. A group can contain individual user IDs and other groups, to the level of nesting supported by the underlying facility.

property. One of a set of characteristics that define the values and behaviors of objects in the Control Center. For example, message processing nodes and deployed message flows have properties.

publication node. An end point of a specific path through a message flow to which a client application subscribes. A publication node has an attribute, subscription point. If this is not specified, the publication node represents the default subscription point for the message flow.

publish/subscribe. Style of messaging application in which the providers of information (publishers) are decoupled from the consumers of that information (subscribers) using a broker. Compare with *point-to-point*. See also *topic*.

publisher. An application that makes information about a specified topic available to a broker in a publish/subscribe system.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that

programs can access messages on the queues that the queue manager owns.

R

retained publication. A published message that is kept at the broker for propagation to clients that subscribe at some point in the future.

request/reply. Type of messaging application in which a request message is used to request a reply from another application. Compare with *datagram*.

rule. A rule is a definition of a process, or set of processes, applied to a message on receipt by the broker. Rules are defined on a message format basis, so any message of a particular format will be subjected to the same set of rules.

S

self-defining message. A message that defines its structure within its content. For example, a message coded in XML is self-defining. Compare with *pre-defined message*.

send and forget. See *datagram*.

setup type. The definition of the type of installation requested. This can be one of **Full**, **Broker only**, or **Custom**.

shared. All configuration data that is shared by users of the Control Center. This data is not operational until it has been deployed.

signature. The definition of the external characteristics of a message processing node.

single-level wildcard. A wildcard that can be specified in subscriptions to match a single level in a topic.

subscriber. An application that requests information about a specified topic from a publish/subscribe broker.

subscription. Information held within a publication node, that records the details of a subscriber application, including the identity of the queue on which that subscriber wants to receive relevant publications.

subscription filter. A predicate that specifies a subset of messages to be delivered to a particular subscriber.

subscription point. An attribute of a publication node that differentiates it from other publication nodes on the same message flow and therefore represents a specific path through the message flow. An unnamed publication node (that is, one without a specific

subscription point) is known as the default publication node.

T

terminal. The point at which one node in a message flow is connected to another node. Terminals enable you to control the route that a message takes, depending whether the operation performed by a node on that message is successful.

topic. A character string that describes the nature of the data that is being published in a publish/subscribe system.

topology. In the broker domain, the brokers, collectives, and connections between them.

transform. A defined way in which a message of one format is converted into one or more messages of another format.

U

User Name Server. The MQSeries Integrator component that interfaces with operating system facilities to determine valid users and groups.

utility function. Function provided by MQSeries Integrator for the benefit of third-party developers writing plug-in nodes or parsers.

W

warehouse. A persistent, historical datastore for events (or messages). The **Warehouse** node within a message flow supports the recording of information in a database for subsequent retrieval and processing by other applications.

wildcard. A character that can be specified in subscriptions to match a range of topics. See also *multilevel wildcard* and *single-level wildcard*.

wire format. This describes the physical representation of a message within the bit-stream.

W3C. World Wide Web Consortium. An international industry consortium set up to develop common protocols to promote evolution and interoperability of the World Wide Web.

X

XML. Extensible Markup Language.

Index

Special Characters

(multi-level wildcard) 13

+ (single-level wildcard) 13

A

access, message buffer

 plug-in node 105

 plug-in parser 137

access, syntax element

 plug-in node 106

 plug-in parser 137

AccountingToken parameter 56

AMI (Application Messaging Interface) 9, 27

AMI publish/subscribe functions 27

Application Messaging Interface 9

 using in publish/subscribe applications 27

application program 7

applications 5

 point-to-point 9

 publish/subscribe 11

ApplIdentityData parameter 57

ApplOriginData parameter 57

attribute functions, plug-in node 67

B

BipCci.h header file 64, 159

BipCni.h header file 64, 112

BipCos.h header file 64

BipCpi.h header file 64

bipGetMessageflowNodeFactory 67

bipGetParserFactory 87

BipRfc.h header file 35

BipRfh2.h 58

BipRfh2.h header file 58

BipSampPluginParser.c file 65

broker

 point-to-point applications 9

 publish/subscribe applications 22

 response message 22, 53

broker response message 22, 53

 message descriptor 57

broker restarts 23

C

CciCompareMode 94

cciGetLastExceptionData 98, 159

cciLog 159

CciLog plug-in utility function 97

cciRethrowLastException 99, 160

cciThrowException 99, 160

character set and encoding for MQRFH2 30

cmqc.h header file 35, 58

cmqfc.h header file 58

cniAddAfter 112

cniAddAsFirstChild 112

cniAddAsLastChild 113

cniAddBefore 113

cniBufferByte 113

cniBufferPointer 114

cniBufferSize 114

cniCopyElementTree 114

cniCreateElementAfter 115

cniCreateElementAfterUsingParser 115

cniCreateElementAsFirstChild 115

cniCreateElementAsFirstChildUsingParser 116

cniCreateElementAsLastChild 116

cniCreateElementAsLastChildUsingParser 116

cniCreateElementBefore 117

cniCreateElementBeforeUsingParser 68, 117

cniCreateInputTerminal 66, 118

cniCreateMessage 68, 94, 118

cniCreateNodeContext 66, 96, 108

cniCreateNodeFactory 68, 118

cniCreateOutputTerminal 66, 119

cniDefineNodeClass 68, 119

cniDeleteMessage 120

cniDeleteNodeContext 108

cniDetach 120

cniElementName 120

cniElementType 94, 121

cniElementValue group 121

cniElementValueState 122

cniElementValueType 122

cniElementValueValue 123

cniEvaluate 67, 95, 109

cniFinalize 123

cniFirstChild 124

cniGetAttribute 96, 109

cniGetAttributeName 110

cniGetBrokerInfo 124

cniGetMessageContext 124

cniGetParserClassName 125

cniIsTerminalAttached 66, 125

cniLastChild 126

cniNextSibling 126

cniParent 126

cniPreviousSibling 127

cniPropagate 67, 127

cniRootElement 93, 128

Index

- cnisearchElement 94
 - cnisearchElement group 128
 - cnisetAttribute 67, 96, 110
 - cnisetElementName 129
 - cnisetElementType 94, 129
 - cnisetElementValue group 130
 - cnisetElementValueValue 131
 - cnisqlCreateStatement 131
 - cnisqlDeleteStatement 132
 - cnisqlExecute 132
 - cnisqlSelect 133
 - cnisqlWriteBuffer 133
 - CodedCharSetId field 31
 - command message
 - Delete Publication 14, 40
 - Deregister Subscriber 19, 42
 - message descriptor 55
 - Publish 13, 45
 - Register Subscriber 16, 48
 - Request Update 20, 51
 - Command property
 - Delete Publication command 40
 - Deregister Subscriber command 42
 - Publish command 45
 - Register Subscriber command 48
 - Request Update command 51
 - Completion property 53
 - conference-type applications 15
 - configuration
 - plug-in node 96
 - plug-in parser 96
 - configuration attributes for plug-in node 66
 - content-based routing, using filters 165
 - datatypes and type mappings 168
 - error reporting and logging 170
 - examples 167
 - field references 165
 - implicit type casting 169
 - rounding errors and overflows 170
 - specifying a filter 166
 - context creation
 - plug-in node 66
 - plug-in parser 87
 - Control Center 5
 - control queue 16
 - CorrelId parameter
 - message sent to broker 55
 - publications forwarded by broker 56
 - response messages 57
 - cpiAddAfter 146
 - cpiAddAsFirstChild 146
 - cpiAddAsLastChild 146
 - cpiAddBefore 147
 - cpiAppendToBuffer 147
 - cpiBufferByte 147
 - cpiBufferPointer 148
 - cpiBufferSize 148
 - cpiCreateAndInitializeElement 148
 - cpiCreateContext 140
 - cpiCreateElement 149
 - cpiCreateParserFactory 87, 149
 - cpiDefineParserClass 87, 150
 - cpiDeleteContext 140
 - cpiElementCompleteNext 150
 - cpiElementCompletePrevious 150
 - cpiElementName 151
 - cpiElementType 94, 151
 - cpiElementValue 143
 - cpiElementValue group 151
 - cpiElementValueValue 153
 - cpiFirstChild 153
 - cpiLastChild 153
 - cpiNextParserClassName 88, 143
 - cpiNextParserCodedCharSetId 144
 - cpiNextParserEncoding 144
 - cpiNextSibling 154
 - cpiParent 154
 - cpiParseBuffer 87, 141
 - cpiParseFirstChild 87, 141
 - cpiParseLastChild 87, 142
 - cpiParseNextSibling 87, 142
 - cpiParsePreviousSibling 87, 142
 - cpiParserType 144
 - cpiPreviousSibling 154
 - cpiRootElement 155
 - cpiSetCharacterValueFromBuffer 155
 - cpiSetElementCompleteNext 155
 - cpiSetElementCompletePrevious 156
 - cpiSetElementName 156
 - cpiSetElementType 94, 156
 - cpiSetElementValue 145
 - cpiSetElementValue group 157
 - cpiSetElementValueValue 158
 - cpiSetNameFromBuffer 158
 - cpiSetNextParserClassName 145
 - cpiWriteBuffer 87, 143
 - create an event source 97
- ## D
- data types, detailed description of MQRFH2 structure 29
 - database access
 - plug-in node 96
 - plug-in parser 96
 - datagram message 9
 - datatypes and type mappings for CBR filters 168
 - defining a node icon 77
 - defining a plug-in node to the Control Center 85
 - Delete Options property 40

Delete Publication command 14, 40
 deleting a local publication 15
 deleting a retained publication 14
 Deregister Subscriber command 19, 42
 Deregister Subscriber command (AMI) 27
 deregistration, publish/subscribe 19

E

element 8
 Encoding field 31
 error codes
 broker response message 58
 publish/subscribe commands 58
 error handling for point-to-point messaging 10
 error reporting and logging for CBR filters 170
 errors
 CciLog utility function 97
 plug-in node 98
 plug-in parser 98
 Windows NT event log 97
 event information 13
 event logging from a plug-in 97
 event source, creating 97
 example
 Broker response message 53
 Delete Publication command 41
 Deregister Subscriber command 44
 Publish command 47
 Register Subscriber command 50
 Request Update command 52
 exception handling functions, node
 interface 159
 overview 107
 exception handling functions, parser
 interface 159
 overview 139
 exceptions
 plug-in node 99
 plug-in parser 99
 expiry of retained publications 14
 Expiry parameter 56

F

field references in content-based routing filters 165
 filter
 content-based routing 165
 publish/subscribe 18
 Filter property
 Deregister Subscriber command 43
 Register Subscriber command 48, 51
 Flags field 32
 flow of messages, publish/subscribe 20
 folder 8
 mcd 36

folder (*continued*)
 message service 36
 psc 37
 pscr 37
 usr 37
 Format field 31
 Format parameter
 message sent to broker 55
 publications forwarded by broker 56
 response messages 57

G

get current broker environment information 124

H

header
 MQRFH 8
 MQRFH2 8
 header files
 BipCci.h 159
 BipCni.h 112
 BipRfc.h 35
 cmqc.h 35, 58
 cmqcfc.h 58
 plug-in node 64
 plug-in parser 64
 help for a plug-in node 78

I

IBMMQSI2 86
 implementation
 plug-in node 66
 plug-in parser 87
 implementation functions, node
 interface 108
 overview 104
 implementation functions, parser
 interface 140
 overview 136
 implicit type casting for CBR filters 169
 initial values of fields in MQRFH2 34
 initialization
 plug-in node 105
 plug-in parser 137
 initialization function
 plug-in node 67
 plug-in parser 87
 installation on a broker
 plug-in node 65
 plug-in parser 65
 installing a plug-in node in the Control Center 84
 integrating a new node 68

Index

L

- LINK /DLL /NOENTRY command 97
- loadable implementation library for plug-in 64
- local publications 15
- local subscription, publish/subscribe 19
- logging
 - plug-in node 95, 159
 - plug-in parser 95, 159

M

- mcd folder 36
- message 4
 - broker response 22
 - datagram 9
 - ordering 15
 - receiving 7
 - request/reply 10
 - send and forget 9
 - sending 7
- message broker 3
- message buffer access
 - plug-in node 105
 - plug-in parser 137
- message catalog 97
- message compiler input (.mc) file 97
- message content, plug-in node 89
- message descriptor
 - broker response message 57
 - command message 55
 - point-to-point message 10
 - publications 56
- message flow 4
- message format, broker response 53
- message headers 8
- message management
 - plug-in node 105
- Message Queue Interface 8
- message service folders 36
- messages, flow of publish/subscribe 20
- modifying a syntax element 94
- MQMD
 - broker response message 57
 - command message 55
 - point-to-point message 10
 - publications 56
- MQRFH header 8
- MQRFH_* values 30, 32
- MQRFH2 definition for C 35
- MQRFH2 definitions
 - Delete Publication 40
 - Deregister Subscriber 42
 - Publish 45
 - Register Subscriber 48
 - Request Update 51

- MQRFH2 header 8
- MQRFH2 structure 29
- MQRFH2_DEFAULT 35
- MQRFH2, multiple headers 37
- MQSeries Integrator
 - applications 5
 - Control Center 5
 - message 4
 - message broker 3
 - message flow 4
 - overview 3
- MQSeries publications viii
- MsgId parameter 57
- MsgType parameter
 - message sent to broker 55
 - publications forwarded by broker 56
 - response messages 57
- multi-level wildcard (#) 13
- multiple message formats, plug-in parser 88
- multiple MQRFH2 headers 37
- multithreading
 - plug-in node 95
 - plug-in parser 95

N

- name element 89
- name-value element 89
- NameValueCCSID field 32
- NameValueData 8
- NameValueData field 32
- NameValueLength field 32
- navigation, syntax element
 - plug-in node 105
 - plug-in parser 137
- node exception handling functions
 - cciGetLastExceptionData 159
 - cciLog 159
 - cciRethrowLastException 160
 - cciThrowException 160
- node implementation and utility functions 103
- node implementation functions
 - cniCreateNodeContext 108
 - cniDeleteNodeContext 108
 - cniEvaluate 109
 - cniGetAttribute 109
 - cniGetAttributeName 110
 - cniSetAttribute 110
- node utility functions
 - cniAddAfter 112
 - cniAddAsFirstChild 112
 - cniAddAsLastChild 113
 - cniAddBefore 113
 - cniBufferByte 113
 - cniBufferPointer 114
 - cniBufferSize 114

- node utility functions (*continued*)
 - cniCopyElementTree 114
 - cniCreateElementAfter 115
 - cniCreateElementAfterUsingParser 115
 - cniCreateElementAsFirstChild 115
 - cniCreateElementAsFirstChildUsingParser 116
 - cniCreateElementAsLastChild 116
 - cniCreateElementAsLastChildUsingParser 116
 - cniCreateElementBefore 117
 - cniCreateElementBeforeUsingParser 117
 - cniCreateInputTerminal 118
 - cniCreateMessage 118
 - cniCreateNodeFactory 118
 - cniCreateOutputTerminal 119
 - cniDefineNodeClass 119
 - cniDeleteMessage 120
 - cniDetach 120
 - cniElementName 120
 - cniElementType 121
 - cniElementValue group 121
 - cniElementValueState 122
 - cniElementValueType 122
 - cniElementValueValue 123
 - cniFinalize 123
 - cniFirstChild 124
 - cniGetBrokerInfo 124
 - cniGetMessageContext 124
 - cniGetParserClassName 125
 - cniIsTerminalAttached 125
 - cniLastChild 126
 - cniNextSibling 126
 - cniParent 126
 - cniPreviousSibling 127
 - cniPropagate 127
 - cniRootElement 128
 - cniSearchElement group 128
 - cniSetElementName 129
 - cniSetElementType 129
 - cniSetElementValue group 130
 - cniSetElementValueValue 131
 - cniSqlCreateStatement 131
 - cniSqlDeleteStatement 132
 - cniSqlExecute 132
 - cniSqlSelect 133
 - cniWriteBuffer 133
 - node, plug-in 63, 171
 - attribute functions 67
 - checking in to the Configuration Manager 86
 - configuration 96
 - configuration attributes 66
 - context creation 66
 - creating a Customizer 83
 - creating a PropertyEditor 81
 - Customizer 83
 - database access 96
 - defining the icon 77
 - node, plug-in (*continued*)
 - defining the interface 69
 - defining the node interface 69
 - defining to the Control Center 85
 - encoded attribute value 73
 - errors 98
 - exception handling interface 159
 - exception handling overview 107
 - exceptions 99
 - file locations 84
 - header files 64
 - help 78
 - implementation 66
 - implementation function interface 108
 - implementation function overview 104
 - initialization and resource creation 105
 - initialization function 67
 - installation in the Control Center 84
 - installation on a broker 65
 - integrating into the Control Center 68
 - introduction 64
 - logging and trace 95
 - message buffer access 105
 - message content 89
 - message management 105
 - output message 68
 - output queues 96
 - portability 95
 - processing function 67
 - properties file 80
 - PropertyEditor 81
 - return codes 100
 - runtime node behavior 95
 - sample code 64
 - SQL statement handling 106
 - storage management 96
 - string handling 96
 - syntax element access 106
 - syntax element modification 94
 - syntax element navigation 90, 105
 - syntax element type 94
 - syntax elements 89
 - threading 95
 - updating 86
 - utility function interface 112
 - utility function overview 105
 - WDP file 76
 - XML interface definition file 69
 - non-retained and retained publications 14
 - non-standard parser 144
- O**
- ordering of messages 15
 - other subscribers only 15

Index

output message, plug-in node 68
overview of MQSeries Integrator 3

P

parser exception handling functions

cciGetLastExceptionData 159
cciLog 159
cciRethrowLastException 160
cciThrowException 160

parser implementation and utility functions 135

parser implementation functions

apiCreateContext 140
apiDeleteContext 140
apiElementValue 143
apiNextParserClassName 143
apiParseBuffer 141
apiParseFirstChild 141
apiParseLastChild 142
apiParseNextSibling 142
apiParsePreviousSibling 142
apiParserType 144
apiSetElementValue 145
apiSetNextParserClassName 145
apiWriteBuffer 143

parser utility functions

apiAddAfter 146
apiAddAsFirstChild 146
apiAddAsLastChild 146
apiAddBefore 147
apiAppendToBuffer 147
apiBufferByte 147
apiBufferPointer 148
apiBufferSize 148
apiCreateAndInitializeElement 148
apiCreateElement 149
apiCreateParserFactory 149
apiDefineParserClass 150
apiElementCompleteNext 150
apiElementCompletePrevious 150
apiElementName 151
apiElementType 151
apiElementValue group 151
apiElementValueValue 153
apiFirstChild 153
apiLastChild 153
apiNextSibling 154
apiParent 154
apiPreviousSibling 154
apiRootElement 155
apiSetCharacterValueFromBuffer 155
apiSetElementCompleteNext 155
apiSetElementCompletePrevious 156
apiSetElementName 156
apiSetElementType 156
apiSetElementValue group 157

parser utility functions (*continued*)

apiSetElementValueValue 158
apiSetNameFromBuffer 158

parser, plug-in 63

configuration 96
context creation 87
database access 96
errors 98
exception handling interface 159
exception handling overview 139
exceptions 99
header files 64
implementation 87
implementation function interface 140
implementation function overview 136
initialization and resource creation 137
initialization function 87
installation on a broker 65
introduction 64
logging and trace 95
message buffer access 137
multiple message formats 88
output queues 96
parser functions 87
portability 95
return codes 100
sample code 64
storage management 96
string handling 96
syntax element access 137
syntax element navigation 137
threading 95
utility function interface 146
utility function overview 137

persistence 23

Persistence parameter

publications forwarded by broker 56
response messages 57

plug-in node 63, 171

attribute functions 67
configuration 96
configuration attributes 66
context creation 66
creating a Customizer 83
creating a PropertyEditor 81
Customizer 83
database access 96
defining the icon 77
defining the interface 69
defining to the Control Center 85
encoded attribute value 73
errors 98
event logging 97
exceptions 99
file locations 84
header files 64

- plug-in node (*continued*)
 - help 78
 - implementation 66
 - implementation function interface 108
 - implementation function overview 104
 - initialization and resource creation 105
 - initialization function 67
 - installation in the Control Center 84
 - installation on a broker 65
 - integrating into the Control Center 68
 - introduction 64
 - locking 86
 - logging and trace 95
 - message buffer access 105
 - message content 89
 - message management 105
 - output message 68
 - output queues 96
 - portability 95
 - processing function 67
 - properties file 80
 - PropertyEditor 81
 - return codes 100
 - runtime node behavior 95
 - sample code 64
 - sample customizer 173, 180
 - SQL statement handling 106
 - storage management 96
 - string handling 96
 - syntax element access 106
 - syntax element modification 94
 - syntax element navigation 90, 105
 - syntax element type 94
 - syntax elements 89
 - threading 95
 - updating 86
 - utility function interface 112
 - utility function overview 105
 - WDP file 76
- plug-in parser 63
 - configuration 96
 - context creation 87
 - database access 96
 - errors 98
 - exceptions 99
 - header files 64
 - implementation 87
 - implementation function interface 140
 - implementation function overview 136
 - initialization and resource creation 137
 - initialization function 87
 - installation on a broker 65
 - introduction 64
 - logging and trace 95
 - message buffer access 137
 - multiple message formats 88
- plug-in parser (*continued*)
 - output queues 96
 - parser functions 87
 - portability 95
 - return codes 100
 - sample code 64
 - storage management 96
 - string handling 96
 - syntax element access 137
 - syntax element navigation 137
 - threading 95
 - utility function interface 146
 - utility function overview 137
- point-to-point applications 9
- portability
 - plug-in node 95
 - plug-in parser 95
- Priority parameter
 - publications forwarded by broker 56
 - response messages 57
- problem determination
 - event logging from a plug-in 97
- problem determination Q&A website ix
- processing function, plug-in node 67
- programming model 7
- pssc folder 37
- pssc folder 37
- Publication Options property 46
- publications
 - local 15
 - on request only 20
 - retained 13, 20
- publications, MQSeries viii
- Publish command 13, 45
- Publish command (AMI) 27
- publish on request only 20
- Publish Timestamp property 46
- publish/subscribe
 - command messages 39
 - filters 18
 - flow of messages 20
 - local 17
 - messaging 12
 - retained publications 13
 - sample application 25
 - state and event information 14
 - subscriber queue 17
 - subscription 17
 - subscription deregistration 19
 - subscription expiry 19
 - subscription registration 19
 - topic 17
 - topics 13
 - using the Application Messaging Interface 27
 - wildcards 17

Index

publish/subscribe applications 11
publisher 13
PutAppName parameter
 publications forwarded by broker 57
 response messages 57
PutApplType parameter 57
PutDate parameter 57
PutTime parameter 57

Q

QMGrName property
 Deregister Subscriber command 43
 Publish command 47
 Register Subscriber command 50
 Request Update command 52
QName property
 Deregister Subscriber command 44
 Publish command 47
 Register Subscriber command 50
 Request Update command 52
query current broker environment 124
queue
 broker control 16
 input (point-to-point) 9
 input (publish/subscribe) 13
 output 9
 subscriber 17
 temporary dynamic 18

R

RC /v command 97
reason codes, publish/subscribe responses 58
Reason property 53
Receive a publication function (AMI) 27
receiver 9
receiving messages 7
Register Subscriber command 16, 48
Register Subscriber command (AMI) 27
Registration Options property
 Deregister Subscriber command 43
 Register Subscriber command 48
 Request Update command 51
registration, publish/subscribe 19
reply message 10
ReplyToQ parameter
 message sent to broker 55
 publications forwarded by broker 56
ReplyToQMGr parameter
 message sent to broker 56
 publications forwarded by broker 56
Report parameter 57
 message sent to broker 55
 publications forwarded by broker 56
 response messages 57

request message 10
Request Update command 20, 51
request/reply messaging 10
resource creation
 plug-in node 105
 plug-in parser 137
Response folder 53
response messages from broker 53
restarting a broker 23
Results Service sample application 25
retained and non-retained publications 14
retained publications 13, 20
return codes
 plug-in node 100
 plug-in parser 100
root element of a message 89
rounding errors and overflows in CBR filters 170
routing messages that are not from an MQSeries
 Integrator publisher 16
rules and formatter header structure for Version 2 29
runtime behavior, plug-in node 95

S

sample code
 plug-in node 64
 plug-in parser 64
sample publish/subscribe application 25
send and forget messaging 9
sender 9
sending messages 7
sequence number 15
Sequence Number property 46
single-level wildcard (+) 13
specifying a CBR filter 166
SQL statement handling for a plug-in node 106
standard parser 144
state and event information, publish/subscribe 14
state information 13
storage management
 plug-in node 96
 plug-in parser 96
string handling
 plug-in node 96
 plug-in parser 96
StrucId field 30
StrucLength field 31
subscriber 16
subscriber queue 18
subscription 17
subscription deregistration 19
subscription expiry 19
subscription point 17
Subscription Point property
 Deregister Subscriber command 42
 Publish command 45

- Subscription Point property (*continued*)
 - Register Subscriber command 48, 51
- subscription registration 19
- syntax element access
 - plug-in node 106
 - plug-in parser 137
- syntax element modification 94
- syntax element navigation
 - plug-in node 90, 105
 - plug-in parser 137
- syntax element type, plug-in node 94
- syntax elements, plug-in node 89

T

- temporary dynamic queue 18
- threading
 - plug-in node 95
 - plug-in parser 95
- timestamp 15
- Topic property
 - Delete Publication command 40
 - Deregister Subscriber command 42
 - Publish command 45
 - Register Subscriber command 48
 - Request Update command 51
- topics
 - publish/subscribe 17
 - wildcards 13
- trace
 - plug-in node 95
 - plug-in parser 95

U

- Unicode 8
- unit of work 23
- updating a plug-in node 86
- UserId parameter 56
- using retained publications 14
- usr folder 37
- utility functions, node
 - interface 112
 - overview 105
- utility functions, parser
 - interface 146
 - overview 137

V

- value element 89
- Version field 30

W

- wildcards 13

- wildcards, publish/subscribe 17
- Windows NT event logging from a plug-in 97

X

- XML 8
- XML interface definition file 69
 - Attribute tag 72
 - AttributeGroup tag 74
 - InTerminal tag 76
 - MessageProcessingNodeType tag 70
 - OutTerminal tag 75

Sending your comments to IBM

MQSeries® Integrator

Programming Guide

SC34-5603-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, use the Readers' Comment Form
- By fax:
 - From outside the U.K., after your international access code use 44 1962 870229
 - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
 - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
 - IBMLink: HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries® Integrator

Programming Guide

SC34-5603-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Telephone

Email



You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States
Belgium	France	Israel	New Zealand	Spain	of America
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

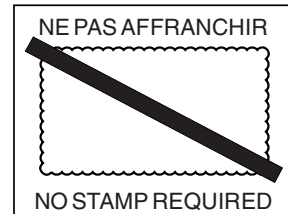
If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO



**REPONSE PAYEE
GRANDE-BRETAGNE**

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____
Company or Organization _____
Address _____

EMAIL _____
Telephone _____

1 Cut along this line

4 Fasten here with adhesive tape



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5603-00

