

WebSphere MQ



Publish/Subscribe User's Guide

Version 6.0

WebSphere MQ



Publish/Subscribe User's Guide

Version 6.0

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices," on page 165.

First edition (May 2005)

This edition of the book applies to the following product:

- IBM WebSphere MQ, Version 6.0

and to any subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1998, 2005. All rights reserved.

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

Contents

Figures	vii
--------------------------	------------

Tables	ix
-------------------------	-----------

About this book	xi
----------------------------------	-----------

Who this book is for	xi
What you need to know to understand this book	xi
How to use this book	xi
Terms used in this book	xi
Appearance of text in this book	xi

Summary of changes	xv
-------------------------------------	-----------

Changes for this edition (SC34-6606-00)	xv
---	----

Part 1. Introduction and system design	1
---	----------

Chapter 1. Introduction	3
--	----------

What is publish/subscribe?	3
What are the components involved?	3
Example of a single broker configuration	4
Example of a multiple broker configuration	4
How does it work?	5
How WebSphere MQ Publish/Subscribe relates to WebSphere MQ	6
How WebSphere MQ Publish/Subscribe relates to WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker	7

Chapter 2. System design	9
---	----------

Topics	9
Matching topic strings	9
Streams	10
Broker networks	11
Passing subscription information between brokers	12
Different types of publication	14
Local and global publications	14
State and event information	14
Retained publications	14
Sample application	15

Part 2. Writing applications.	19
--	-----------

Chapter 3. Introduction to writing applications	21
--	-----------

Message flows	22
Simplified message flow	23
Message ordering	26
Ensuring that messages are retrieved in the correct order	26
Publisher and subscriber identity	27

Subscription name and identity	28
The message descriptor	29
Messages sent to the broker	29
Publications forwarded by the broker	30
Persistence and units of work	31
Limitations	31
Group messages	31
Segmented messages	32
Cluster queues	32
Data conversion of MQRFH structure	32
Using the Application Messaging Interface	32
AMI publish/subscribe functions	32

Chapter 4. Writing publisher applications	35
--	-----------

Registering with the broker	35
Choosing not to register	36
Options you can specify when registering as a publisher	36
Broker restart	37
Changing an application's registration	37
Publishing information	37
Publication data	37
Retained publications	38
Publishing locally and globally	38
Deleting information	38
Deregistering with the broker	39

Chapter 5. Writing subscriber applications	41
---	-----------

Registering as a subscriber	41
Subscriber queues	42
Options you can specify when registering as a subscriber	42
Broker restart	43
Changing an application's registration	43
Requesting information	43
Requesting information from the broker	43
Requesting information from a publisher	44
Deregistering as a subscriber	44

Chapter 6. Format of command messages	47
--	-----------

MQRFH – Rules and formatting header	47
Fields	48
Structure definition in C	50
Publish/Subscribe name/value strings	51
Options using string constants	52
Options using integer constants	52
Sending a command message with the RFH structure	52
Publication data	53
Double-byte character sets	53

Chapter 7. Publish/Subscribe command messages 57

Delete Publication	58
Required parameters	58
Optional parameters	58
Example	58
Error codes	58
Deregister Publisher	60
Required parameters	60
Optional parameters	60
Example	61
Error codes	61
Deregister Subscriber	62
Required parameters	62
Optional parameters	62
Example	64
Error codes	64
Publish	65
Required parameters	65
Optional parameters	65
Example	69
Error codes	69
Register Publisher	70
Required parameters	70
Optional parameters	70
Example	71
Error codes	71
Register Subscriber	72
Required parameters	72
Optional parameters	72
Example	78
Error codes	79
Request Update	80
Required parameters	80
Optional parameters	80
Example	81
Error codes	81

Chapter 8. Error handling and response messages 83

Error handling by the broker	83
Response messages	84
Message descriptor for response messages	84
Types of error response	85
Broker responses	86
Standard parameters	86
Optional parameters	86
Examples	88
Error codes applicable to all commands	88
Problem determination	88

Chapter 9. Sample programs 91

Sample application	92
Running the application	93
Possible extensions	95
Application Messaging Interface samples	96

Part 3. Managing the broker 97

Chapter 10. Setting up a broker 99

Broker queues	99
System queues	99
Other stream queues	100
Internal queues	101
Dead-letter queue	101
Other considerations	101
Access control	101
Backup	101
Broker configuration stanza	102
Broker configuration parameters	102

Chapter 11. Controlling the broker 107

Starting a broker	107
Using triggering to start the broker	107
Stopping a broker	107
Displaying the status of a broker	107
Adding a stream	107
Creating a stream queue	107
Informing other brokers about the stream	108
Deleting a stream	108
Deleting a stream on an isolated broker	108
Deleting a stream on a broker that is part of a network	108
Adding a broker to a network	109
Deleting a broker from the network	109
Problems when deleting brokers	110
Deleting a broker that has a child broker	110
Sequence of commands for adding and deleting brokers	110

Chapter 12. Control commands. 113

clrmqbrk (Clear broker's memory of a neighboring target broker)	114
dlmqbrk (Delete broker)	117
dspmqrk (Display broker status)	119
endmqbrk (End broker function)	121
migmqrk (Migrate broker to WebSphere Business Integration Brokers)	123
strmqbrk (Start broker function)	125

Chapter 13. Message broker exit 129

Publish/subscribe routing exit	129
Parameters	129
Usage notes	129
Publish/subscribe routing exit parameter structure	130
Writing a publish/subscribe routing exit program	136
Limitations on WebSphere MQ work done in the routing exit	136
Security considerations	137
Compiling a publish/subscribe routing exit program	137
Sample routing exit	137

Part 4. System programming 139

Chapter 14. Writing system management applications 141

Format of broker administration messages	141
Subscription deregistered message	142
Stream deleted message	142
Broker deleted message	142
Stream support messages	143
Children messages	143
Parent messages	143
MQCFH - PCF header	143
Reason codes returned from publish/subscribe messages	145
PCF Command Messages	146
Delete Publication	147
Deregister Publisher	147
Deregister Subscriber	147
Publish	148
Register Publisher	148
Register Subscriber	149
Request Update	150

Chapter 15. Finding out about other publishers and subscribers 151

Metatopics	151
Subscribing to metatopics	152

Using wild cards	153
Example requests	153
Authorized metatopics	153
Finding out about brokers	154
Message format for metatopics	154
Parameters	155
Sample program for administration information	157
Operation	158
Example of metatopic information	159

Part 5. Appendixes 161

Appendix A. Header files 163

Appendix B. Notices 165

Trademarks	166
----------------------	-----

Index 167

Sending your comments to IBM . . . 173

Figures

1. Simple publish/subscribe example	4	11. Flow of messages in a single-broker system	24
2. Publish/subscribe example with two brokers	5	12. Flow of messages in a multi-broker system	24
3. Communication between publishers, subscribers, and brokers.	6	13. Flow of messages using retained publications	25
4. Simple broker hierarchy	11	14. Flow of messages using publish on request only	25
5. Propagation of subscriptions through a broker network.	12	15. Message descriptor and RFH structure . . .	53
6. Multiple subscriptions	13	16. Publication data after the RFH structure	54
7. Propagation of publications through a broker network.	13	17. Publishing data within the NameValueString	54
8. The results service application	16	18. User-defined publication data	55
9. Basic flow of messages.	22	19. Inheriting the CCSID	56
10. Simplified flow of messages	23	20. Results service running with four match simulators	95
		21. Sample Broker stanza for qm.ini	102

Tables

1.	How to read syntax diagrams	xii	5.	Sample programs for iSeries	91
2.	Fields in MQRFH	47	6.	Fields in MQPXP	130
3.	Initial values of fields in MQRFH	50	7.	Parameters for publisher and subscriber information messages	155
4.	Sample programs for AIX, HP-UX, Linux, Solaris, and Windows	91			

About this book

This book describes how to use WebSphere® MQ Publish/Subscribe.

Who this book is for

This book is for experienced users of WebSphere MQ who want to use WebSphere MQ Publish/Subscribe. Familiarity with these WebSphere MQ books is assumed:

- *WebSphere MQ Application Programming Reference*
- *WebSphere MQ Application Programming Guide*
- *WebSphere MQ Programmable Command Formats and Administration Interface*
- *WebSphere MQ System Administration Guide*

What you need to know to understand this book

To use WebSphere MQ Publish/Subscribe you need to have a good knowledge of WebSphere MQ in general. All the sample programs and header files are in the C programming language.

How to use this book

This book contains the following parts:

- Part 1, “Introduction and system design,” on page 1 explains what you can do using WebSphere MQ Publish/Subscribe.
- Part 2, “Writing applications,” on page 19 discusses how to write programs to use WebSphere MQ Publish/Subscribe.
- Part 3, “Managing the broker,” on page 97 describes how to set up and manage your brokers.
- Part 4, “System programming,” on page 139 contains information needed to write system management programs.

Terms used in this book

UNIX® system is used as a general term for any of the following platforms:

- AIX®
- HP-UX
- Linux®
- Solaris

Appearance of text in this book

This book uses the following type styles:

CompCode

The name of a parameter of a call, a field in a structure, or an attribute of an object

dltmqbrk

A control command or command message

MQRFH

The name of a data type or structure

MQPS_COMMAND

The name of a constant

MQPSCommand Publish

Examples

"MQPSTopic"

A character string

How to read syntax diagrams

This book contains syntax diagrams (sometimes referred to as “railroad” diagrams).

Each syntax diagram begins with a double right arrow and ends with a right and left arrow pair. Lines beginning with a single right arrow are continuation lines. You read a syntax diagram from left to right and from top to bottom, following the direction of the arrows.

Other conventions used in syntax diagrams are:

Table 1. How to read syntax diagrams

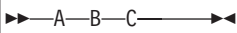
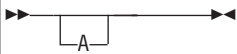

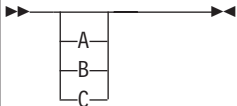
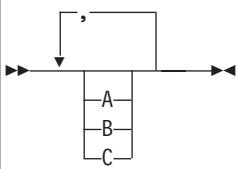
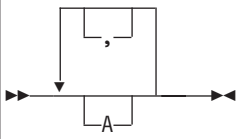
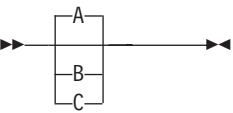
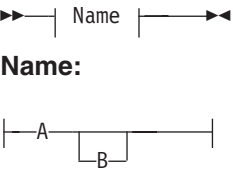
Convention	Meaning
	You must specify values A, B, and C. Required values are shown on the main line of a syntax diagram.
	You may specify value A. Optional values are shown below the main line of a syntax diagram.
	Values A, B, and C are alternatives, one of which you must specify.
	Values A, B, and C are alternatives, one of which you might specify.
	You might specify one or more of the values A, B, and C. Any required separator for multiple or repeated values (in this example, the comma (,)) is shown on the arrow.
	You might specify value A multiple times. The separator in this example is optional.

Table 1. How to read syntax diagrams (continued)

Convention	Meaning
	Values A, B, and C are alternatives, one of which you might specify. If you specify none of the values shown, the default A (the value shown above the main line) is used.
 <p>Name:</p>	The syntax fragment Name is shown separately from the main syntax diagram.
Punctuation and uppercase values	Specify exactly as shown.
Lowercase values (for example, <i>name</i>)	Supply your own text in place of the <i>name</i> variable.

Summary of changes

This section describes changes in this edition of *WebSphere MQ Publish/Subscribe User's Guide*.

Changes for this edition (SC34-6606-00)

- WebSphere MQ Publish/Subscribe is now a part of the main WebSphere MQ product.
- Platform support has been extended to include iSeries.
- The Broker Configuration tool on Windows® (cfgmqbrk) has been replaced by the Broker page in WebSphere MQ Explorer.
- Messages and reason codes that were previously in this book have been moved to the *WebSphere MQ Messages* book.
- Constants that were previously in this book have been moved to the new *WebSphere MQ Constants* book.
- Minor editorial improvements have been made.
- This book is a revision of the *MQSeries Publish/Subscribe User's Guide*, Version 1 Release 0.7, GC34-5269-09

Changes

Part 1. Introduction and system design

Chapter 1. Introduction	3
What is publish/subscribe?	3
What are the components involved?	3
Example of a single broker configuration	4
Example of a multiple broker configuration	4
How does it work?	5
How WebSphere MQ Publish/Subscribe relates to WebSphere MQ	6
How WebSphere MQ Publish/Subscribe relates to WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker	7
Chapter 2. System design	9
Topics	9
Matching topic strings	9
Streams.	10
Broker networks	11
Passing subscription information between brokers	12
Different types of publication	14
Local and global publications	14
State and event information	14
Retained publications	14
Sample application	15

Chapter 1. Introduction

This chapter explains what WebSphere MQ Publish/Subscribe is and introduces the concepts and terminology used in this manual. It contains the following topics:

- “What is publish/subscribe?”
- “How does it work?” on page 5
- “How WebSphere MQ Publish/Subscribe relates to WebSphere MQ” on page 6
- “How WebSphere MQ Publish/Subscribe relates to WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker” on page 7

What is publish/subscribe?

WebSphere MQ Publish/Subscribe allows you to decouple the provider of information from the consumers of that information.

Before a standard WebSphere MQ application can send some information to another application, it needs to know something about that application. For example, it needs to know the name of the queue to which to send the information, and might also specify a queue manager name.

WebSphere MQ Publish/Subscribe removes the need for your application to know anything about the target application. All it has to do is send information it wants to share to a standard destination managed by WebSphere MQ Publish/Subscribe, and let WebSphere MQ Publish/Subscribe deal with the distribution. Similarly, the target application does not have to know anything about the source of the information it receives.

What are the components involved?

The provider of the information is called a *publisher*. Publishers supply information about a subject, without needing to know anything about the applications that are interested in the information.

The consumer of the information is called a *subscriber*. The subscriber decides what information it is interested in, and then waits to receive that information. Subscribers can receive information from many different publishers, and the information they receive can also be sent to other subscribers.

The information is sent in a WebSphere MQ message, and the subject of the information is identified by a *topic*. The publisher specifies the topic when it publishes the information, and the subscriber specifies the topics on which it wants to receive publications. The subscriber is sent information about only those topics it subscribes to.

Interactions between publishers and subscribers are all controlled by a *broker*. The broker receives messages from publishers, and subscription requests from subscribers (to a range of topics). The broker’s job is to route the published data to the target subscribers.

Related topics can be grouped together to form a *stream*. Publishers can choose to use streams, for example to restrict the range of publications and subscriptions that

What is publish/subscribe?

a broker has to support, or to provide access control. The broker has a default stream that is used for all topics that do not belong to another stream.

The broker uses standard WebSphere MQ facilities to do this, so your applications can use all the features that are available to existing WebSphere MQ applications. This means that you can use persistent messages to get once-only assured delivery, and that your messages can be part of a transactional unit-of-work to ensure that messages are delivered to the subscriber only if they are committed by the publisher.

Example of a single broker configuration

Figure 1 illustrates a basic broker configuration. The example shows the configuration for a news service, where information is available from Publishers about several topics within a single stream:

- Publisher 1 is publishing information about sports results using a topic of Sport
- Publisher 2 is publishing information about stock prices using a topic of Stock
- Publisher 3 is publishing information about film reviews using a topic of Films, and about television listings using a topic of TV

Three subscribers have registered an interest in different topics, so the broker sends them the information that they are interested in:

- Subscriber 1 receives the sports results and stock prices
- Subscriber 2 receives the film reviews
- Subscriber 3 receives the sports results

None of the subscribers have registered an interest in the television listings, so these are not distributed.

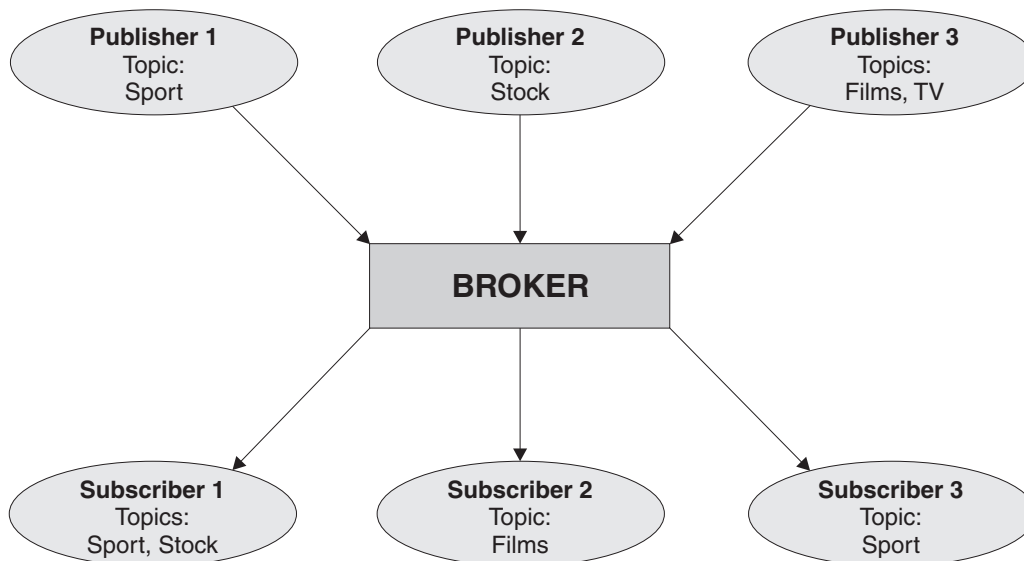


Figure 1. Simple publish/subscribe example. This shows the relationship between publishers, subscribers, and brokers.

Example of a multiple broker configuration

You can have only one broker on each WebSphere MQ queue manager; however, brokers can communicate with other brokers in your WebSphere MQ system, so subscribers can subscribe to one broker and receive messages that were initially published to another broker. This is illustrated in Figure 2 on page 5.

In this example, a second broker has been added.

- Broker 2 is used by Publisher 4 to publish weather forecast information, using a topic of Weather, and information about traffic conditions on major roads, using a topic of Traffic.
- Subscriber 4 also uses this broker, and subscribes to information about traffic conditions using topic Traffic.
- Subscriber 3 also subscribes to information about weather conditions, even though it uses a different broker from the publisher. This is possible because the brokers are linked to each other.

A publication is propagated to another broker only if a subscription to that topic exists on the other broker.

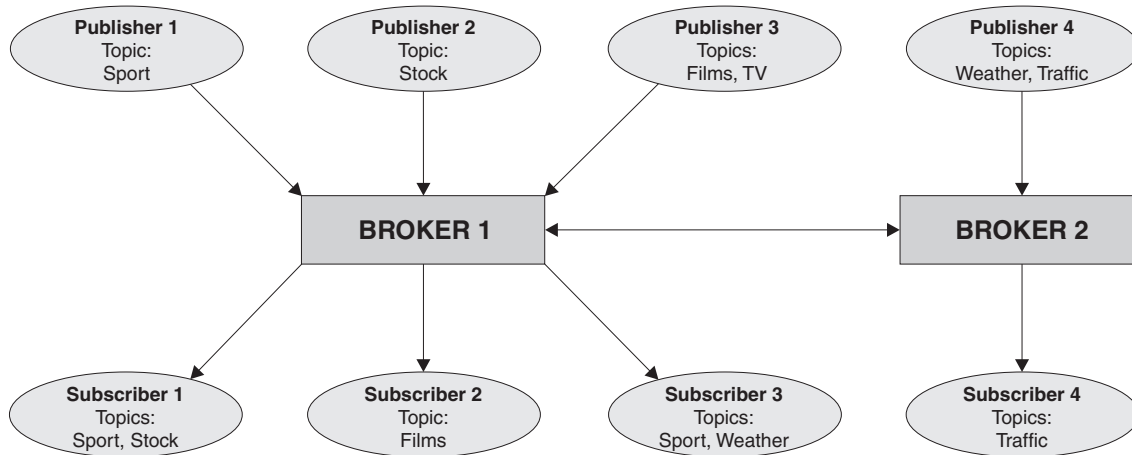


Figure 2. Publish/subscribe example with two brokers

How does it work?

Publishers, subscribers, and brokers communicate with each other using *command messages*. These messages are used to do the following things:

Publisher and broker

The following communications take place between publishers and brokers:

1. A publisher can register its intention to publish information about certain topics (this is optional: registration can take place with the first publication, or not at all, as described in “Registering with the broker” on page 35).
2. A publisher sends publication messages to the broker, containing the publication data (or referring to it). The messages can be forwarded directly to the subscribers, or, in the case of retained publications, be held at the broker until requested by a subscriber.
3. A publisher can send a message to the broker requesting that a retained publication held at the broker be deleted.
4. A publisher can deregister with the broker when it has finished sending messages about a certain topic.

These interactions are all described in Chapter 4, “Writing publisher applications,” on page 35.

Subscriber and broker

The following communications take place between subscribers and brokers:

1. A subscriber registers with a broker, specifying the topics that it is interested in.

How does it work?

2. The broker sends to the subscriber subsequent publications that match the topics specified. Alternatively, the subscriber can request retained publications held at the broker.
3. The subscriber can deregister with the broker for certain topics when it is no longer interested in them.

These interactions are all described in Chapter 5, “Writing subscriber applications,” on page 41.

Broker and broker

The following communications take place between brokers:

1. Brokers can exchange subscription registrations and deregistrations.
2. Brokers can exchange publications, and requests to delete publications.
3. Brokers can exchange information about themselves.

These interactions are illustrated in Figure 3.

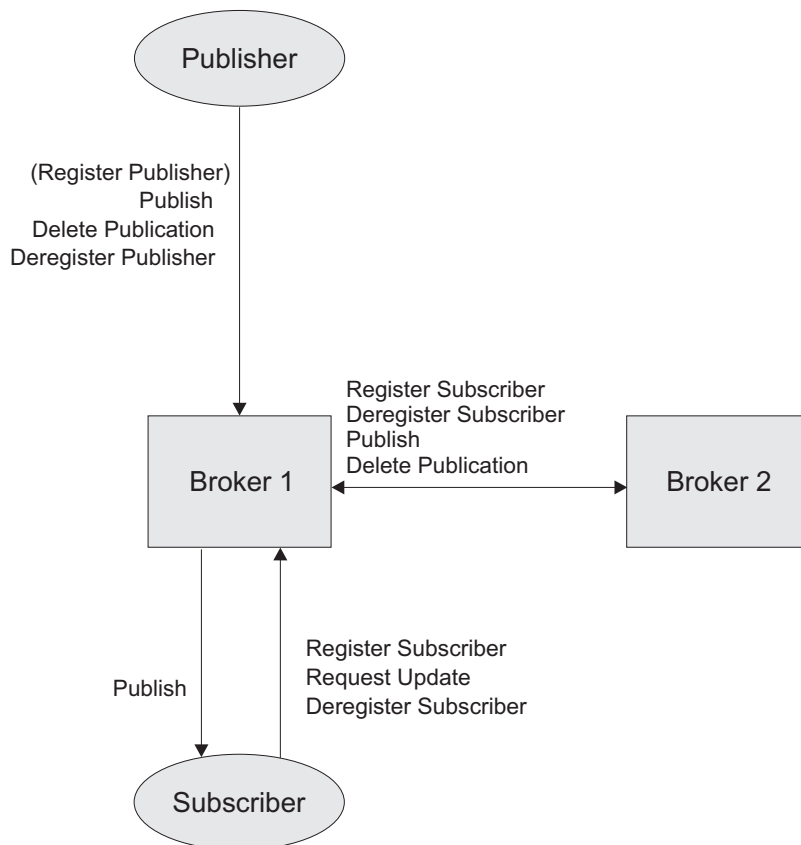


Figure 3. Communication between publishers, subscribers, and brokers

How WebSphere MQ Publish/Subscribe relates to WebSphere MQ

WebSphere MQ Publish/Subscribe is a function of WebSphere MQ. The broker runs on WebSphere MQ for AIX, HP-UX, iSeries™, Linux, Windows, and Solaris. It uses standard WebSphere MQ facilities (but note that it does not support message groups or segmented messages).

You can have one broker on each WebSphere MQ queue manager. The broker uses the same name as the queue manager.

Applications can be written with standard WebSphere MQ programming techniques, using the Message Queue Interface (MQI) or the Application Messaging Interface (AMI).

Publishers and subscribers do not have to be on the same machine as a broker. They can reside anywhere in the network, provided that there is a route from their queue manager to the broker. So, for example, you could have a publisher on z/OS® and a subscriber on Solaris.

How WebSphere MQ Publish/Subscribe relates to WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker

WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker work with WebSphere MQ messaging, extending its basic connectivity and transport capabilities to provide a powerful *message broker* solution driven by business *rules*. Messages are formed, routed, and transformed according to the rules defined by an easy-to-use graphical user interface.

Diverse applications can exchange information in unlike forms, with brokers handling the processing required for the information to arrive in the right place in the correct format, according to the rules you have defined. The applications have no need to know anything other than their own conventions and requirements.

Applications also have much greater flexibility in selecting which messages they want to receive, because they can specify a *topic* filter, or a *content-based* filter, or both, to control the messages made available to them.

WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker provide a framework that supports supplied, basic, functions along with *plug-in* enhancements, to enable rapid construction and modification of business processing rules that are applied to messages in the system.

WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker address the needs of business and application integration through management of information flow. It provides services based on message brokers to allow you to do the following:

- 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.
- Create structured topic names, topic-based access control functions, content-based subscriptions, and subscription points.

How WebSphere MQ Publish/Subscribe relates to WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker

- Exploit a plug-in interface to develop message processing node types that can be incorporated into the broker framework to complement or replace the supplied nodes, or to incorporate node types developed by Independent Software Vendors (ISVs).
- Enable instrumentation by products such as those developed by Tivoli®, using system management hooks.

The benefits of WebSphere Business Integration Message Broker and WebSphere Business Integration Event Broker can be realized both within and beyond your enterprise:

- Your processes and applications can be integrated by providing message and data transformations in a single place, the broker. This helps reduce costs of application upgrades and modifications.
- You can extend your systems to reach your suppliers and customers, by meeting their interface requirements within your brokers. This can help you improve the quality of your interactions and allow you to respond more quickly to changing or additional requirements.

Chapter 2. System design

This chapter discusses the things that you need to consider when you design your WebSphere MQ Publish/Subscribe system. It contains the following topics:

- “Topics”
- “Streams” on page 10
- “Broker networks” on page 11
- “Different types of publication” on page 14

The “Sample application” on page 15 illustrates how these features can be used in practice.

Topics

A topic identifies what a publication is about. It consists of a character string.

You can use any characters within the single-byte character set for which the machine is configured in a topic string. However, a topic string might need to be translated to a different character representation, so you are recommended to use only those characters that are available in the configured character set of all relevant machines. Topic strings are case sensitive, and a blank character has no special meaning. A null character terminates the string and is not considered to be part of it.

Subscribers can specify a topic or range of topics, using wildcards, for the information that they want.

Matching topic strings

The wildcard characters recognized by WebSphere MQ Publish/Subscribe are:

- * Zero or more characters
- ? One character

In the example shown in Figure 1 on page 4, the high-level topic of ‘Sport’ might be divided into separate topics covering different sports, such as:

Sport/Soccer
Sport/Golf
Sport/Tennis

These might be divided further, to separate different types of information about each sport, such as:

Sport/Soccer/Fixtures
Sport/Soccer/Results
Sport/Soccer/Reports

Note: WebSphere MQ Publish/Subscribe does not recognize that the ‘/’ character is being used in a special way. However, it is recommended that the ‘/’ character is used as a separator to ensure compatibility with other WebSphere business integration functions.

The following topic strings could be used in subscriptions to retrieve particular sets of information:

Topics

***** All information on Sport, Stock, Films and TV.

Sport/*

All information on Soccer, Golf and Tennis.

Sport/Soccer/*

All information on Soccer (Fixtures, Results and Reports).

Sport/*/Results

All Results for Soccer, Golf and Tennis.

Note that wildcards do not span streams (see “Streams”).

The percent character ‘%’ is used as an escape character, to allow these characters to be used in a topic string. For example, the string ‘ABC%*D’ represents the actual topic ABC*D. If the string ABC%*D is specified in a **Publish** message (where wildcard characters are not allowed), the string could be matched by a subscriber specifying the string ABC?D.

To use a % character in a topic string, specify two percent characters ‘%%’. A percent character in a string must always be followed by a ‘*’, a ‘?’, or another ‘%’ character.

If wildcard characters are not allowed in a message, a ‘*’ or ‘?’ character can be present only if it is immediately preceded by a ‘%’ character so that the ‘*’ or ‘?’ character loses its wildcard semantics. Therefore, ABC%*D is a valid topic string in a **Publish** message but ABC*D is not.

Streams

Streams provide a way of separating the flow of information for different topics. A stream is implemented as a set of queues, one at each broker that supports the stream. Each queue has the same name (the name of the stream). The default stream set up between all the brokers in a network is called SYSTEM.BROKER.DEFAULT.STREAM.

Streams can be created by an application or by the administrator. Stream names are case sensitive, and stream queues must be local queues (not alias queues). Stream names beginning with the characters ‘SYSTEM.BROKER.’ are reserved for WebSphere MQ use. For more information see “Broker queues” on page 99.

A broker has a separate thread for each stream that it supports. If multiple streams are used, the broker can process publications arriving at different stream queues in parallel. Other advantages of using streams are as follows:

- To provide a high level grouping of topics.

Streams act as high-level qualifiers for topics. For instance, in the example shown in Figure 1 on page 4, a separate stream might be set up for Sport. In this case, to get the soccer results you need to subscribe to the Soccer/Results topic specifying the ‘Sport’ stream. The other topics (Stock, Films, TV) remain on the default stream, unless other streams are set up for them.

Note that wildcard characters are not used for stream names, and that wildcards do not span streams. For example, a subscriber to topic ‘*’ on the ‘Sport’ stream does not receive publications published on other streams.

- To restrict the range of publications and subscriptions that a broker has to deal with.

A given stream can be restricted to a subtree of a hierarchy or the stream can be split into separate hierarchies that are not connected (see “Broker networks”). For example, if broker 1 in Figure 4 does not support a stream supported by its children, brokers 2 and 3 each form the root of a separate hierarchy for that stream, and no subscriptions or publications flow between the two hierarchies.

- To provide access control.

A broker has a stream queue for each stream that it supports. Normal WebSphere MQ access control techniques can be used to control whether a particular application is authorized to put messages onto this queue (publish to this stream), or to browse messages from the queue (subscribe to it). Although a subscribing application does not get messages from the broker’s queue directly, the broker checks the subscriber’s authorization to subscribe to the broker’s queue when it registers the subscription. This authorization check takes place at the broker to which the application publishes or subscribes, not at other brokers to which the publication or subscription might be propagated.

The administrator can change publishers’ and subscribers’ stream queue authorizations dynamically (using normal WebSphere MQ queue management facilities), although the changes might not take effect until the broker is restarted.

- To define a certain quality of service for broker-to-broker communication of publications.

You can send information associated with one stream along different channels from those used for another stream. For example, a non-urgent stream might have its associated channels active only during the night.

- To allow different queue attributes (such as maximum message length) to be assigned for publications on different streams.

Broker networks

You can link brokers together to form a network of brokers. A broker network must be arranged as a hierarchy. The broker at the top of the hierarchy is called the *root broker*. The root broker can have one or more *child brokers*, and is known as the *parent broker* to these brokers. The child brokers can also have child brokers, and so on, as illustrated in Figure 4.

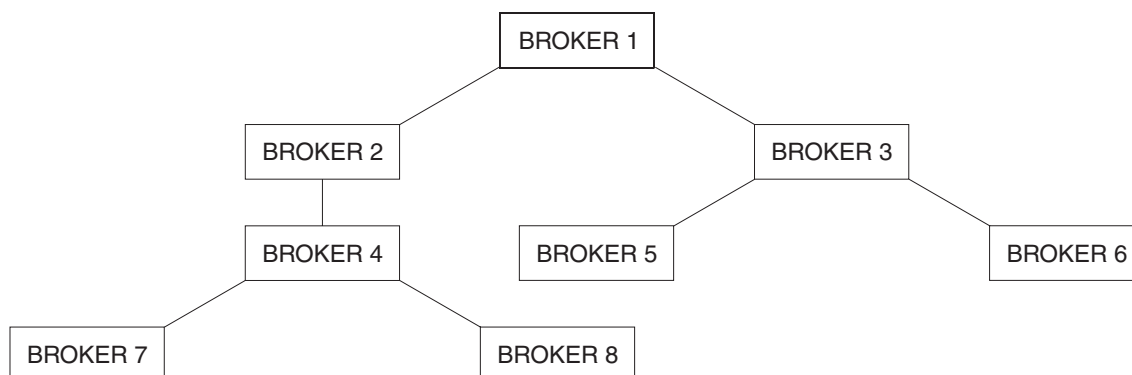


Figure 4. Simple broker hierarchy. Broker 1 is the root broker and brokers 2 and 3 are its children. Broker 4 is the child of broker 2 and the parent of brokers 7 and 8.

Using a hierarchy reduces the number of channels that need to be defined because each broker does not need to be connected to every other broker. Both publication and subscription traffic take a hierarchic route to their destinations.

Broker networks

Each broker maintains administrative information about its parent broker. When a broker first starts, it communicates with its parent. In this way, each broker knows the identities of its immediate children as well as its parent. These are known as the broker's *neighbors*.

Define the hierarchy from the root down and, if it is necessary to delete brokers, delete them from the bottom up. This usually means that to change the root broker you have to delete the whole network and start again (in exceptional cases you can use the `clrmqbrk` command described on 114).

Passing subscription information between brokers

Subscriptions flow to all nodes in the network that support the stream in question. This is shown in Figure 5.

A broker consolidates all the subscriptions that are registered with it, whether from applications directly or from other brokers. In turn, it registers subscriptions for these topics with its neighbors, unless a subscription already exists. This is shown in Figure 6 on page 13.

When an application publishes information, the receiving broker forwards it (possibly through one or more other brokers) to any applications that have valid subscriptions for it, including applications registered at other brokers supporting this stream (for global publications). This is shown in Figure 7 on page 13.

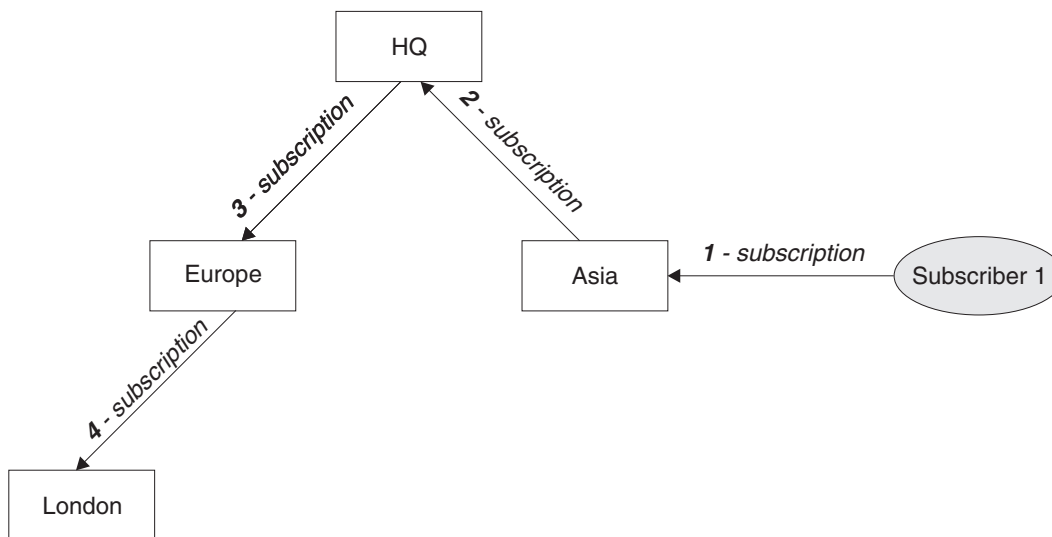


Figure 5. Propagation of subscriptions through a broker network. Subscriber 1 registers a subscription for a particular topic and stream on the Asia broker (1). The subscription for this topic is forwarded to all other brokers in the network that support the stream (2,3,4).

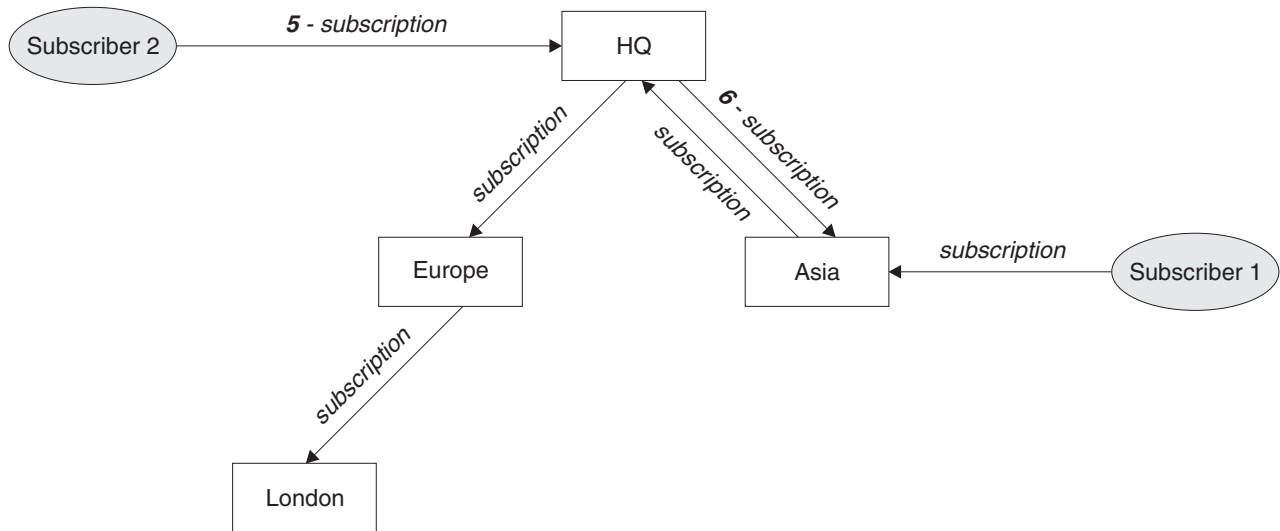


Figure 6. Multiple subscriptions. Subscriber 2 registers a subscription, with the same topic and stream as in Figure 5 on page 12, on the HQ broker (5). The subscription for this topic is forwarded to the Asia broker, so that it is aware that subscriptions exist elsewhere on the network (6). The subscription does not have to be forwarded to the Europe broker, because a subscription for this topic has already been registered (step 3 in Figure 5 on page 12).

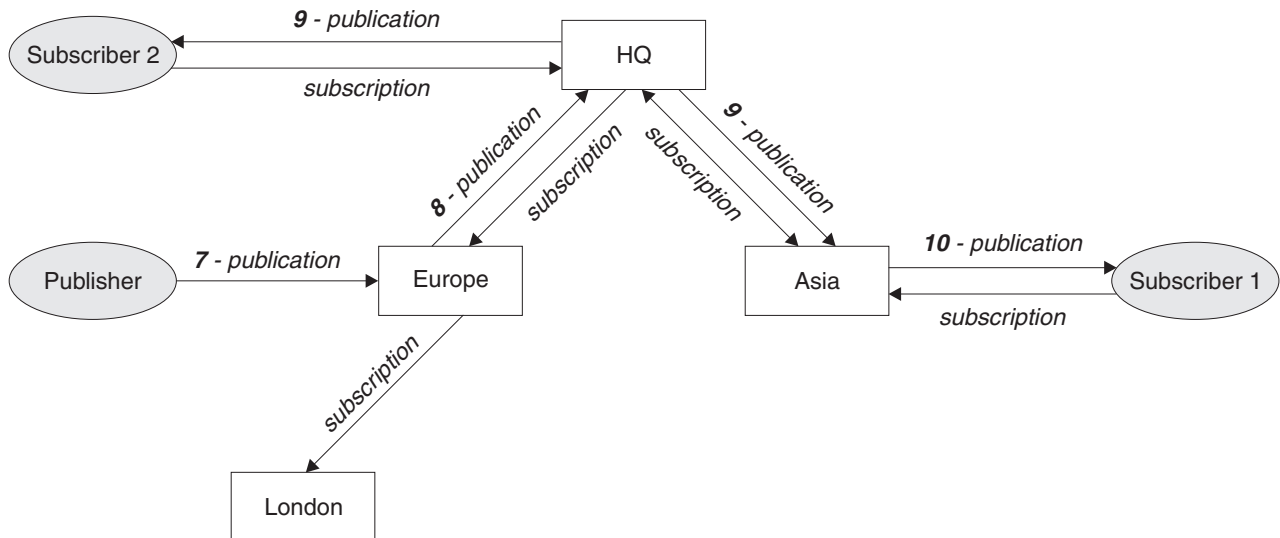


Figure 7. Propagation of publications through a broker network. A publisher sends a publication, on the same topic and stream as in Figure 6, to the Europe broker (7). A subscription for this topic exists from HQ to Europe, so the publication is forwarded to the HQ broker (8). However, no subscription exists from London to Europe (only from Europe to London), so the publication is not forwarded to the London broker. The HQ broker sends the publication directly to subscriber 2 and to the Asia broker (9), from where it is forwarded to subscriber 1 (10).

When a broker sends any publish or subscribe message to another broker, it sets its own user ID in the message, and uses its own authority to put the message. This means that the broker must have the authority to put messages onto other brokers' queues (unless the channel is set up to put incoming messages with the message channel agent's authority). This also means that all authorization checks are performed at the publisher's or subscriber's local broker.

For more information about brokers, see Part 3, “Managing the broker,” on page 97.

Different types of publication

The broker can handle publications it receives in different ways, depending on the type of information contained in the publication.

Local and global publications

A publication that is made available through all the brokers on a network is called a *global publication*. If required, access to publications can be restricted to subscribers that use the same broker as the publisher. This is called a *local publication*, and it can be specified when the publisher registers with the broker, or each time it sends publications to the broker. Local publications are not forwarded to other brokers.

Subscribers can specify whether they want to receive local publications or global publications (but not both) when they register with the broker. Subscribers subscribing to global publications do not receive local publications, even if they are published at the same broker that their subscription was registered at.

State and event information

Publications can be categorized as follows:

State publications

State publications contain information about the current *state* of something, such as the price of stock or the current score in a soccer match. When something happens (for example, the stock price changes or the soccer score changes), the previous state information is no longer required because it is superseded by the new information.

A subscriber will want to receive the current version of the state information when it starts up, and be sent new information whenever the state changes.

Event publications

Event publications contain information about individual *events* that occur, such as a trade in some stock or the scoring of a particular goal. Each event is independent of other events.

A subscriber will want to receive information about events as they happen.

Retained publications

By default, a broker deletes a publication when it has sent that publication to all the interested subscribers. This type of processing is suitable for event information, but is not always suitable for state information. A publisher can specify that it wants the broker to keep a copy of a publication, which is then called a *retained publication*. The copy can be sent to subsequent subscribers who register an interest in the topic. This means that new subscribers don't have to wait for information to be published again before they receive it. For example, a subscriber registering a subscription to a stock price would receive the current price straightaway, without waiting for the stock price to change (and hence be re-published).

The broker retains only one publication for each topic, so the old publication is deleted when a new one arrives. It is recommended that you do not have more than one publisher sending retained publications on the same topic.

Subscribers can specify that they do *not* want to receive retained publications, and existing subscribers can ask for duplicate copies of retained publications to be sent to them.

When deciding whether to use retained publications, you need to consider several factors.

- Will your publications contain state information or event information?

Event publications do not usually have to be retained. For state information, if all the subscriptions to a topic are in place *before* any publications are made on that topic (and no new ones are expected), there is no need to retain publications because they will be delivered to all subscribers when they are published.

Another reason why publications might not need to be retained is if they are very frequent (for example, every second), because a new subscriber (or a subscriber recovering from a failure) receives the current state almost immediately after it subscribes.

- How will the subscriber application recover from a failure?

If the publisher does not use retained publications, the subscriber application might need to store its current state locally. If the publisher does use retained publications, the subscriber application can use the **Request Update** message to refresh its state information after a restart.

Note that the broker will continue to send publications to a registered subscriber even if that subscriber is not running. This could lead to a buildup of messages on the subscriber queue, which can be avoided if the subscriber registers with the 'Publish on Request Only' option. The subscriber must then refresh its state periodically using the **Request Update** command message. Note that in this case the subscriber does *not* receive any non-retained publications.

- What are the performance implications of retaining publications?

The broker needs to write retained publications to disk during the **Publish** request, which reduces throughput. If the publications are very large, a considerable amount of queue space (and hence disk space) is needed to store the retained publication of each topic. In a multi-broker environment, retained publications are also stored by all other brokers in the network that have a matching subscription.

Sample application

The sample application (see Chapter 9, "Sample programs," on page 91) simulates 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 8 on page 16.

Sample application

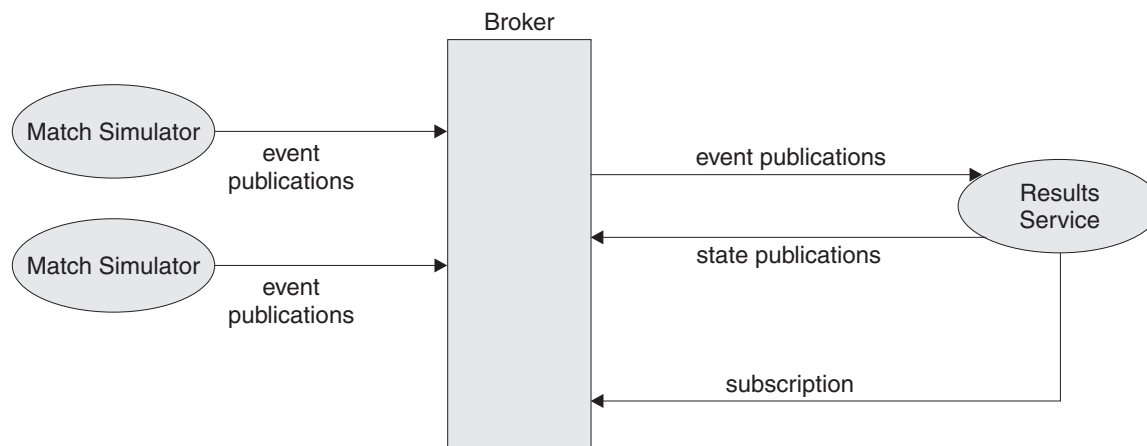


Figure 8. The results service application. The match simulators publish events when a match starts or finishes, or a goal is scored. The results service subscribes to these events, and publishes the latest scores as state publications.

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 `SAMPLE.BROKER.RESULTS.STREAM` stream. (The sample program sets up its own stream to avoid any possible conflict with customer applications on the default stream).

- When a match starts, the names of the teams are published on the `Sport/Soccer/Event/MatchStarted` topic.
- 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, because 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 topic `Sport/Soccer/State/LatestScore/Team1 Team2`, where Team1 and Team2 are the names of the teams in the match.

A subscriber wanting to receive all the latest scores could register a wildcard subscription to topic `Sport/Soccer/State/LatestScore/*`. If it was interested in one particular team only, it could register a different wildcard subscription to topic `Sport/Soccer/State/LatestScore/*MyTeam*`.

Note that the results service must be started before the match simulators, otherwise it might miss some events and so cannot 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 the results service 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 that contain latest scores. (This is done using a different *CorrelId* as explained in "Publisher and subscriber identity" on page 27.)

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 is still registered (no **Deregister Subscriber** message has been sent) it includes any event publications that arrived while the results service was inactive.

This sample program illustrates the following aspects of a Publish/Subscribe application:

- The use of streams other than the default stream.
- Event publications (not retained).
- State publications (retained).
- Wildcard matching of topic strings.
- Multiple publishers on the same topics (event publications only).
- The need to subscribe to a topic *before* it is published on (event 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.

Further details of the messages sent between the publisher, subscriber and broker, and the results service sample program, are given in Part 2, “Writing applications,” on page 19.

Part 2. Writing applications

Chapter 3. Introduction to writing applications	21	Broker restart.	43
Message flows	22	Changing an application's registration	43
Simplified message flow	23	Requesting information	43
Message ordering	26	Requesting information from the broker	43
Ensuring that messages are retrieved in the		Requesting information from a publisher	44
correct order	26	Deregistering as a subscriber	44
Publisher and subscriber identity	27		
Subscription name and identity.	28	Chapter 6. Format of command messages	47
The message descriptor	29	MQRFH – Rules and formatting header	47
Messages sent to the broker	29	Fields	48
Publications forwarded by the broker.	30	Structure definition in C	50
Persistence and units of work	31	Publish/Subscribe name/value strings	51
Limitations	31	Options using string constants	52
Group messages.	31	Options using integer constants	52
Segmented messages	32	Sending a command message with the RFH	
Cluster queues	32	structure	52
Data conversion of MQRFH structure.	32	Publication data	53
Using the Application Messaging Interface	32	Double-byte character sets	53
AMI publish/subscribe functions	32		
Publish command	32	Chapter 7. Publish/Subscribe command	
Register Subscriber command	32	messages	57
Deregister Subscriber command	33	Delete Publication	58
Receive a publication	33	Required parameters	58
		Optional parameters	58
Chapter 4. Writing publisher applications	35	Example	58
Registering with the broker	35	Error codes	58
Choosing not to register	36	Deregister Publisher	60
Options you can specify when registering as a		Required parameters	60
publisher	36	Optional parameters	60
Queue name	36	Example	61
Selecting a stream	36	Error codes	61
Publisher identity	36	Deregister Subscriber	62
Registration scope	36	Required parameters	62
Registration expiry	36	Optional parameters	62
Broker restart.	37	Example	64
Changing an application's registration	37	Error codes	64
Publishing information	37	Publish	65
Publication data	37	Required parameters	65
Including data in the message	37	Optional parameters	65
Referring to data in the message	37	Example	69
Retained publications	38	Error codes	69
Expiry of retained publications	38	Register Publisher	70
Publishing locally and globally	38	Required parameters	70
Deleting information	38	Optional parameters	70
Deregistering with the broker	39	Example	71
		Error codes	71
Chapter 5. Writing subscriber applications	41	Register Subscriber	72
Registering as a subscriber	41	Required parameters	72
Subscriber queues	42	Optional parameters	72
Options you can specify when registering as a		Example	78
subscriber	42	Error codes	79
Queue name	42	Request Update	80
Selecting a stream	42	Required parameters	80
Subscriber identity	42	Optional parameters	80
Subscription scope	43	Example	81
Subscription expiry	43	Error codes	81

Chapter 8. Error handling and response	
messages	83
Error handling by the broker	83
Response messages	84
Message descriptor for response messages	84
Types of error response	85
OK response	85
Warning response	85
Error response	85
Broker responses	86
Standard parameters	86
Optional parameters	86
Examples	88
Error codes applicable to all commands	88
Problem determination	88
 Chapter 9. Sample programs	 91
Sample application	92
Running the application	93
Possible extensions	95
Application Messaging Interface samples	96

Chapter 3. Introduction to writing applications

Applications use command messages to communicate with the broker when they want to publish or subscribe to information. These messages use the WebSphere MQ Rules and Formatting Header (RF Header), which is described in Chapter 6, “Format of command messages,” on page 47. The content of each command message starts with an MQRFH structure. This structure contains a name/value string, which defines the type of command the message represents and any parameters associated with the command. In the case of a **Publish** command message, the name/value string is usually followed by the data to be published, in any format specified by the user. Broker responses to command messages also use the MQRFH structure.

The normal Message Queue Interface (MQI) calls (such as MQPUT and MQGET) can be used to put RF Header command messages to the broker queue, and to retrieve response messages and publications from their respective queues. The MQI is described in the *WebSphere MQ Application Programming Guide*. Most command messages are sent to the broker’s control queue (SYSTEM.BROKER.CONTROL.QUEUE), but **Publish** and **Delete Publication** command messages are sent to the appropriate stream queue at the broker (for example, SYSTEM.BROKER.DEFAULT.STREAM).

Alternatively, you can use the WebSphere MQ Application Messaging Interface (AMI) to send messages to and receive them from the broker. The AMI constructs and interprets the fields in the RF Header, so you don’t need to understand its structure. In addition, the application programmer is not concerned with details of how WebSphere MQ sends the message. These details (for instance, the queue name and fields in the message descriptor) are contained in AMI services and policies set up by a system administrator. The AMI is available as a SupportPac™.

This chapter describes the things that you need to know before you start writing a publisher or subscriber application. It discusses the following topics:

- “Message flows” on page 22
- “Message ordering” on page 26
- “Publisher and subscriber identity” on page 27
- “The message descriptor” on page 29
- “Persistence and units of work” on page 31
- “Limitations” on page 31
- “Using the Application Messaging Interface” on page 32

You can find more information about writing applications in the following chapters:

- Chapter 4, “Writing publisher applications,” on page 35
- Chapter 5, “Writing subscriber applications,” on page 41
- Chapter 6, “Format of command messages,” on page 47
- Chapter 7, “Publish/Subscribe command messages,” on page 57
- Chapter 8, “Error handling and response messages,” on page 83
- Chapter 15, “Finding out about other publishers and subscribers,” on page 151

Sample programs to illustrate the techniques used are described in Chapter 9, “Sample programs,” on page 91.

Message flows

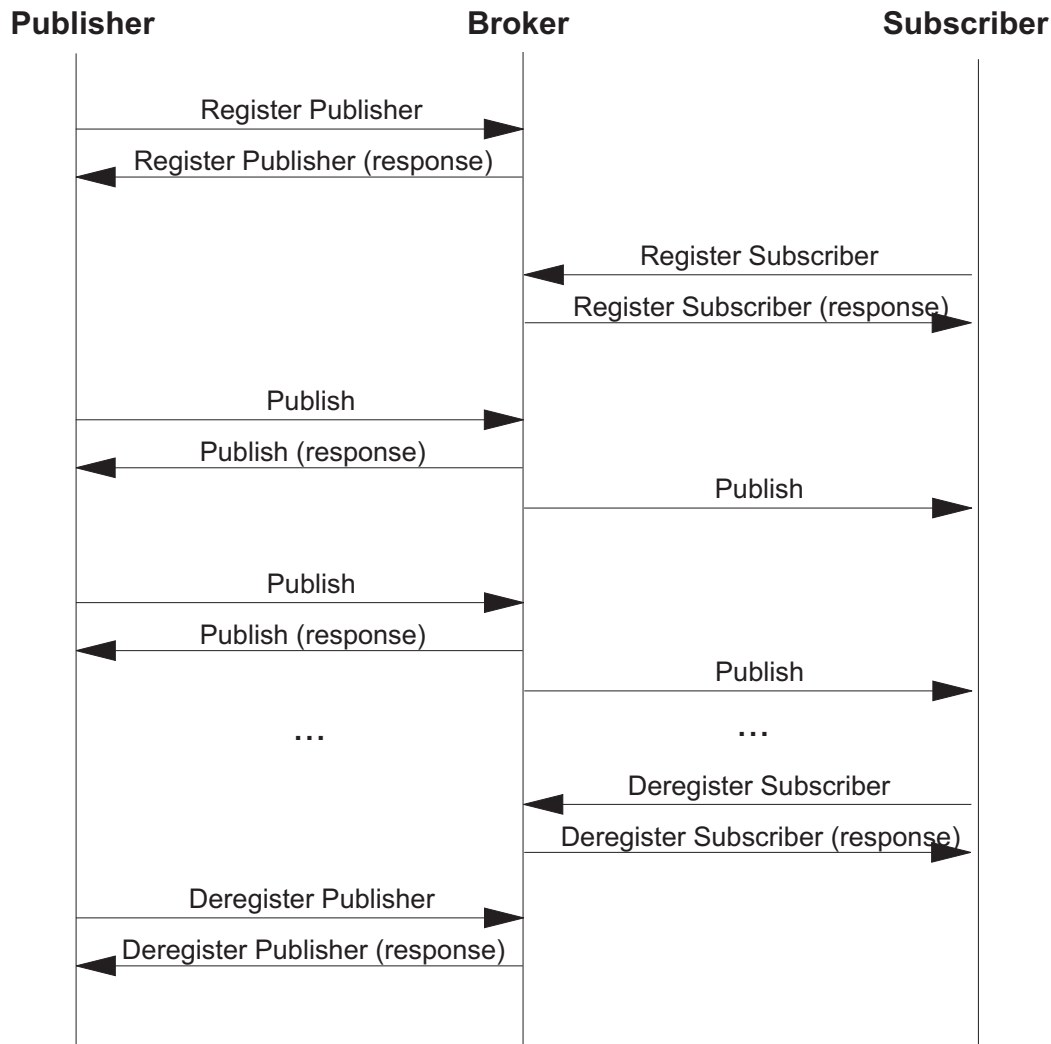


Figure 9. Basic flow of messages

Figure 9 shows the basic flow of messages using the **Register Publisher**, **Deregister Publisher**, **Register Subscriber**, **Deregister Subscriber** and **Publish** command message and responses. This flow applies to all event publications, and to state information where the subscriber wants to get the latest published state of a topic.

The responses are optional, and the **Register Publisher** and **Deregister Publisher** command messages can be omitted (publishers can choose not to register, or to register on their first publish command). So the flow diagram can be simplified as shown in Figure 10 on page 23.

Simplified message flow

Figure 10 is a simplified version of Figure 9 on page 22 with the optional messages

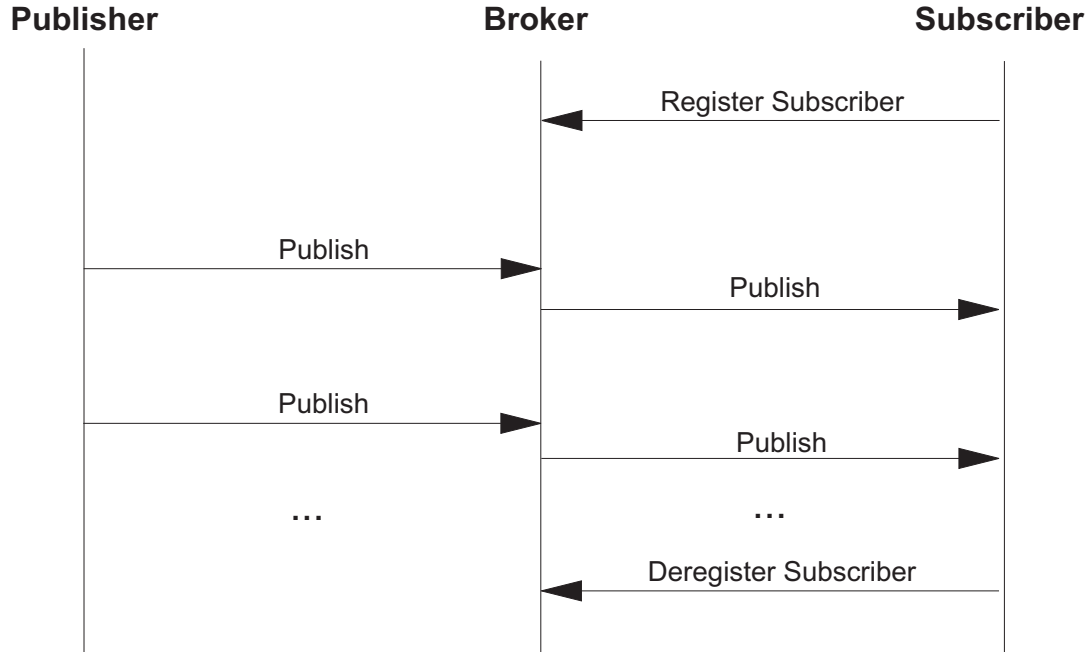


Figure 10. Simplified flow of messages

and responses omitted.

Figure 11 on page 24 shows how publish and subscribe messages flow between the publisher, the subscriber, and the broker queues. In Figure 12 on page 24 this is extended to a two-broker system.

The flow of messages when retained publications are used is shown in Figure 13 on page 25. In this case, the subscriber receives the current retained publication as soon as it registers a subscription. In Figure 14 on page 25, the subscriber registers with the 'Publish on Request Only' option, so it doesn't receive the publication until it sends a **Request Update** command message. (Note that the first publication is not delivered to the subscriber, because it is updated by the second publication before the update request is received).

Message flows

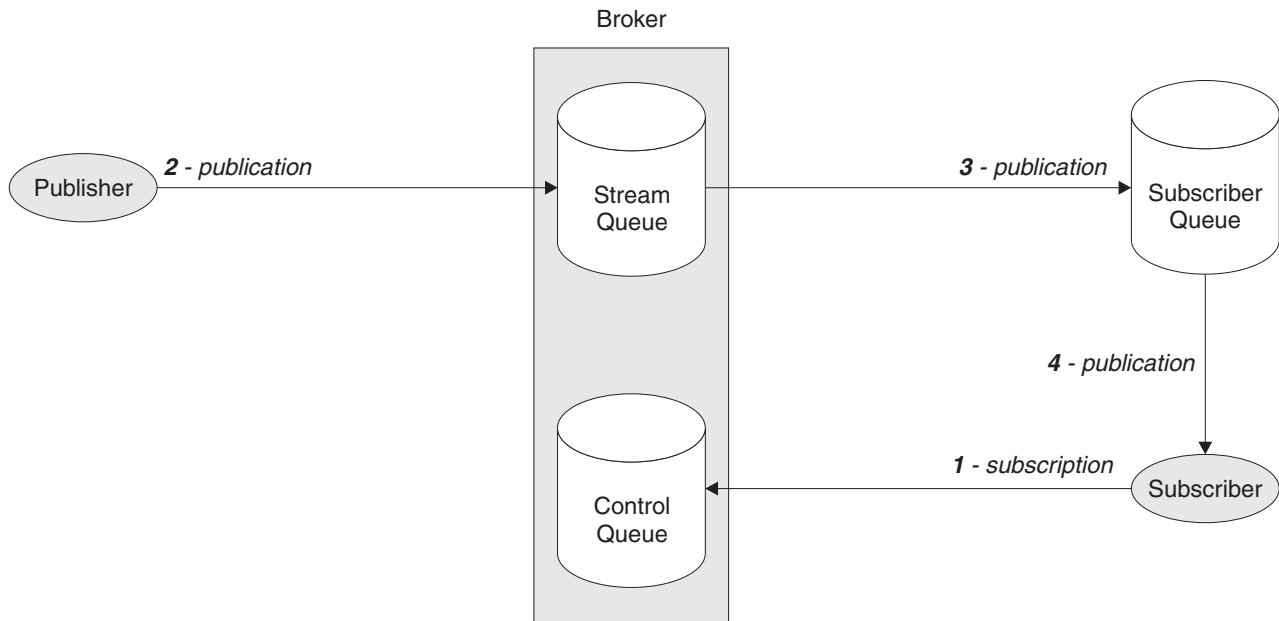


Figure 11. Flow of messages in a single-broker system. The subscriber registers a subscription by putting a message on the broker's control queue (1). Subsequently, a publisher puts a publication message, for the same topic, on the corresponding stream queue in the broker (2). The broker forwards the publication by putting the same message on the subscriber queue (3), from where the subscriber application can get it (4).

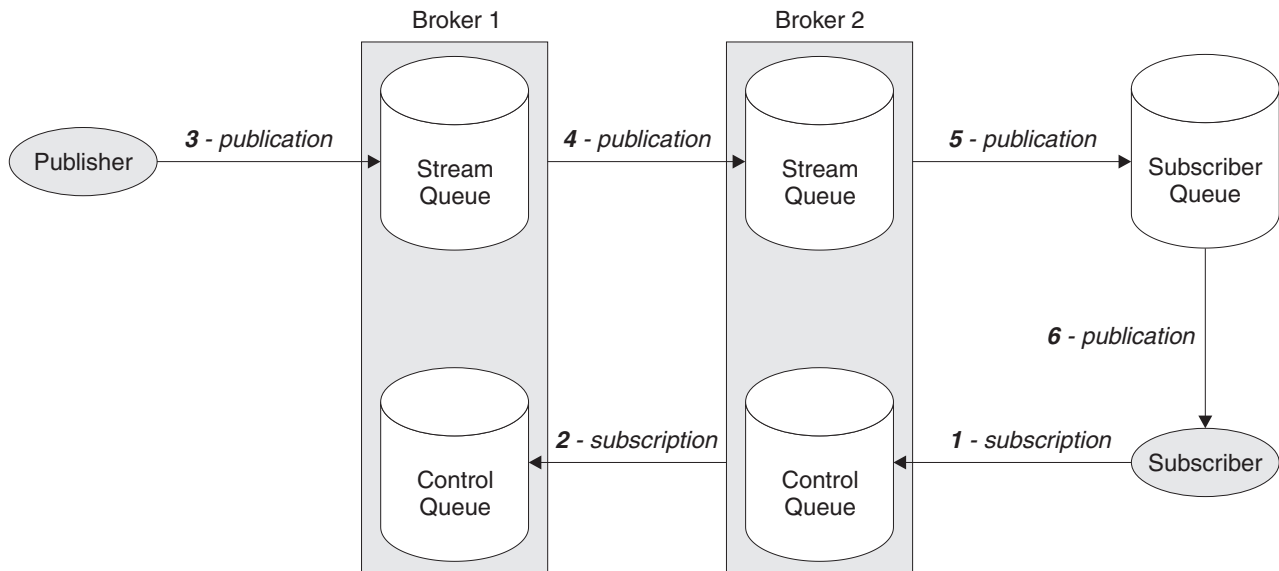


Figure 12. Flow of messages in a multi-broker system. The subscriber registers a subscription as in Figure 11(1). Broker 2 forwards the subscription by putting a message on the control queue of Broker 1 (2). Subsequently, a publisher puts a publication message, for the same topic, on the corresponding stream queue in Broker 1 (3). The publication is forwarded to Broker 2 (4), and then to the subscriber queue (5), from where the subscriber application can get it (6).

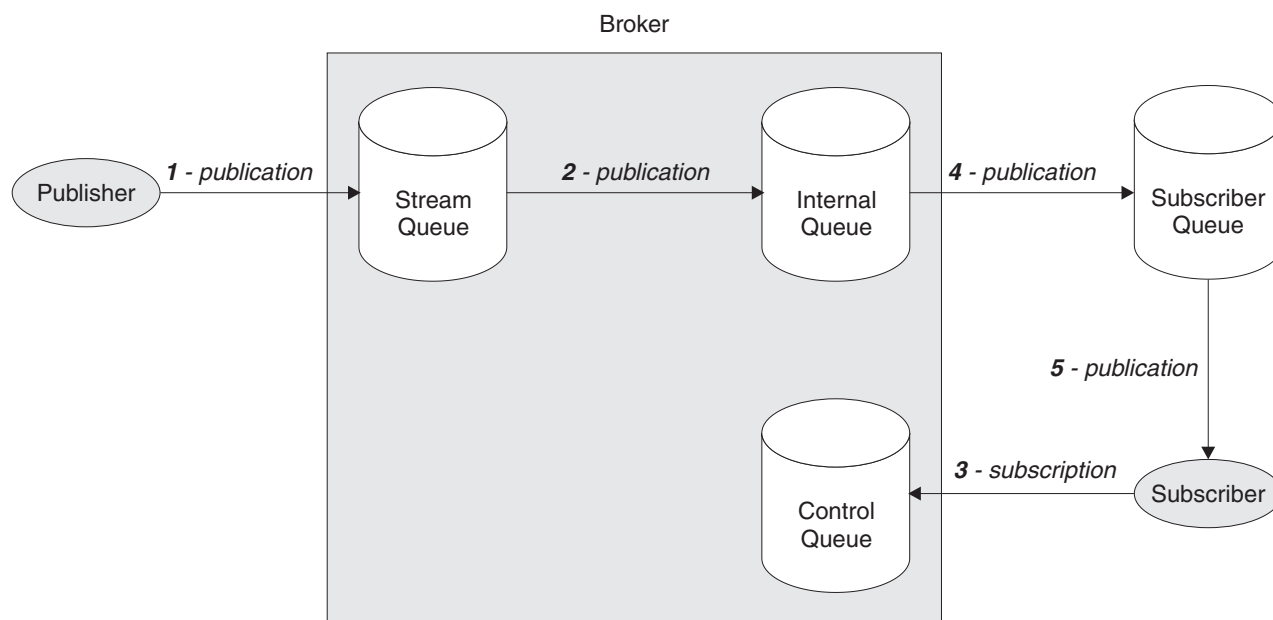


Figure 13. Flow of messages using retained publications. A publisher sends a retained publication by putting a message on the appropriate stream queue in the broker (1). The broker stores the publication on an internal queue (2). Subsequently, a subscriber registers a subscription, to the same topic and stream, by putting a message on the broker's control queue (3). The broker sends the current retained publication for this topic by putting a message on the subscriber queue (4), from where the subscriber application can get it (5).

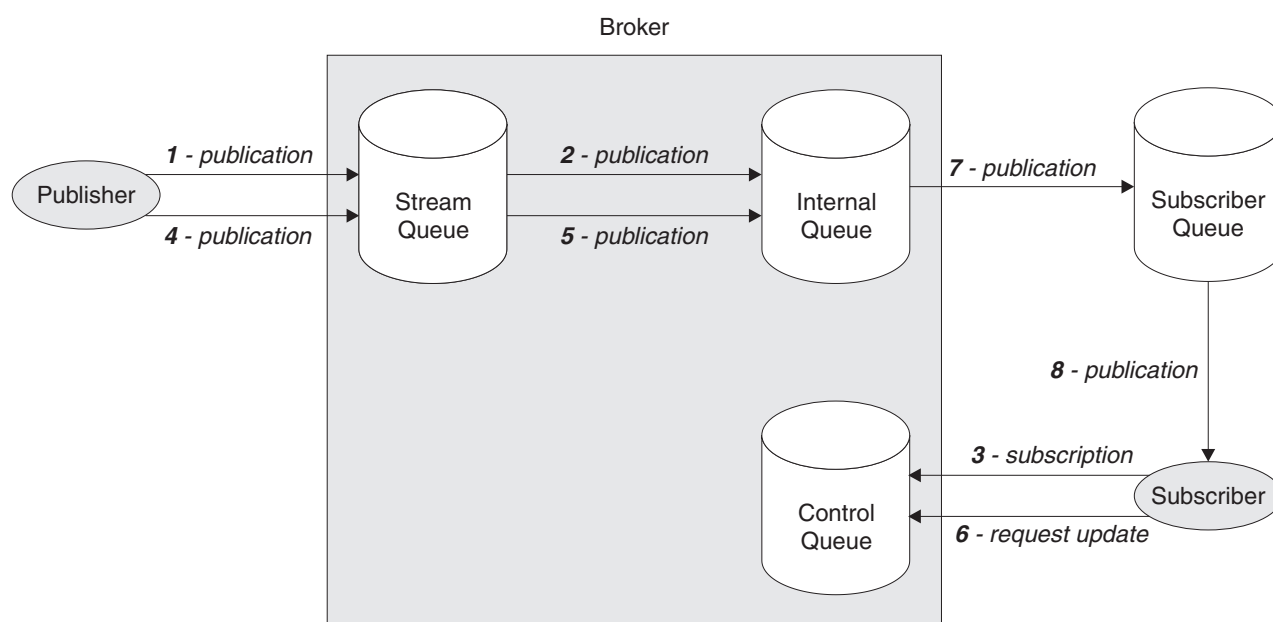


Figure 14. Flow of messages using publish on request only. A publisher sends a retained publication to a stream queue in the broker (1). The broker stores it on an internal queue (2). A subscriber registers a subscription, to the same topic and stream, by putting a message on the broker's control queue (3), but it uses the 'Publish on Request Only' option so the broker takes no action. Subsequently, the publisher sends a second retained publication to the broker (4), which replaces the first one on the internal queue (5). The subscriber then sends a request update message to the broker's control queue (6). This causes the broker to send the current retained publication to the subscriber queue (7), from where the subscriber application can get it (8).

Message ordering

For a given stream, messages are published by brokers in the same order as they are received from publishers (subject to reordering based on message priority). This normally means that each subscriber receives messages from a particular broker, on a particular topic and stream, from a particular publisher in the order that they are published by that publisher.

However, as with all WebSphere MQ messages, it is possible for messages, occasionally, to be delivered out of order. This can happen in the following situations:

- 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, so that a message is put to a dead-letter queue and therefore delayed, while subsequent messages pass straight through.
- If the administrator deletes a broker or uses the **clrmqbrk** command (**CLRMQMBRK** on iSeries) when publishers and subscribers are still operating, causing queued messages to be put to the dead-letter queue and subscriptions to be interrupted.

If these circumstances cannot occur, publications are always delivered in order.

Ensuring that messages are retrieved in the correct order

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 *SequenceNumber* parameter is supported on the **Publish** message. A publisher can include this with each message, increasing the value by one for each successive message that it publishes for the same stream and topic. The broker does not check or set this parameter; the responsibility for it lies with the publisher. The number can be checked by the subscriber, which needs to remember the last sequence number it received for each stream and topic combination.

If a subscriber receives a publication message that is out of order, it can react in various ways:

- If it needs only the latest information (for example, a stock price) and the sequence number is greater than it should be (that is, one or more previous publications have not yet been received), this publication message is accepted. If the sequence number is less than it should be (that is, this is a previous publication), the publication message is ignored.
- If it needs to keep track of all information, it must record this information and its sequence number.
- A *PublishTimestamp* parameter, in Universal time, is provided on the **Publish** message. A publisher can include this with each message (with or without the *SequenceNumber* parameter). This is particularly useful if subscribers are interested only in the latest information; they can check whether the timestamp is greater than that of the last **Publish** message that they processed.

In both of the above solutions, the publisher and subscriber need to remember information about the last message they processed for a particular stream and topic. In the first solution this is the *SequenceNumber* for the **Publish** message, and in the second solution it is the *PublishTimestamp*. This information might need to be remembered 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.

Publisher and subscriber identity

A publisher's or subscriber's identity consists of the following:

- Their queue name.
- Their queue manager name (this can be blank to indicate the local queue manager).
- Correlation identifier (this is optional).

Alternatively, the subscriber's identity can consist of a subscription name. See "Subscription name and identity" on page 28.

The correlation identifier can be used to distinguish between different publishers or subscribers using the same queue. If different subscribers are using the same queue, all publications sent by the broker to a subscriber specify the correlation identifier in the *CorrelId* field of the message descriptor (MQMD).

Note: For responses, MQRO_XX_CORREL_ID report options determine the correlation identifier used. Applications using a correlation identifier for identification typically specify the *CorrelId* and the MQRO_PASS_CORREL_ID option.

The recipient can then use **MQGET** with the *CorrelId* to retrieve the messages.

This allows several applications to share a queue (this might be desirable if there are many clients). It also allows one application to distinguish between publications arising from different subscriptions. An example of this is in the sample program described on page 15. When the results service restarts, it subscribes to the topic Sport/Soccer/State/LatestScore/*, with the 'Publish on Request Only' option. It uses a different *CorrelId* from that used to subscribe to the Sport/Soccer/Event/* publications. This allows it to retrieve from the same queue all the retained 'LatestScore' publications before it starts processing the event publications again.

An identity that includes the correlation identifier in the message descriptor is established by including MQPS_CORREL_ID_AS_IDENTITY in the *RegistrationOptions* parameter of the **Register Publisher** or **Register Subscriber** message (or of the **Publish** message for implicit registration). The correlation identifier to be used as part of the identity must not be zero.

If MQPS_CORREL_ID_AS_IDENTITY is not set, the identity does not include the correlation identifier and the broker uses a correlation identifier of its own choosing when sending messages to that publisher or subscriber. When a broker selects the correlation identifier itself, this does not conflict with other message identifiers or correlation identifiers generated by queue managers.

A single publisher or subscriber queue can therefore support multiple identities, each with a specific correlation identifier value, plus one further identity for which the correlation identifier is not specified (MQPS_CORREL_ID_AS_IDENTITY was not set for registration). Each identity is treated by the broker as being independent

Publisher and subscriber identity

of the others. (Usually, however, a queue has either a number of identities each with its own specific correlation identifier, or only one identity with no specific correlation identifier).

MQPS_CORREL_ID_AS_IDENTITY should be set by a publisher whose identity includes a correlation identifier when sending a **Publish** message to the broker, so that the broker can identify the publisher using the *CorrelId* field in the MQMD. If such a message is received by the broker when there is no registration in effect for the publisher's queue and the correlation identifier specified, an implicit registration is performed (unless MQPS_NO_REGISTRATION is specified).

When a **Publish message** is sent by a broker to a subscriber whose identity includes a correlation identifier, the *CorrelId* field in the MQMD is set to the required correlation identifier. The correlation identifier sent to the subscriber depends only upon what the subscriber set when it registered. The correlation identifier used by the publisher is independent of the correlation identifier sent to the subscriber.

MQPS_CORREL_ID_AS_IDENTITY is valid for the **Deregister Publisher** and **Deregister Subscriber** message, to delete a registration for an identity that includes a correlation identifier.

The value used for a correlation identifier that is part of a publisher's or subscriber's identity needs to be unique only between the other users of the same queue. The MQPMO_NEW_CORREL_ID option can be used to cause the queue manager to generate a unique value.

Subscription name and identity

Publish/Subscribe broker subscribers can be identified by their queue name, queue manager name, and optional correlation identifier. This, in conjunction with a topic, identifies an individual subscription, referred to here as the *traditional identity* of the subscription. An additional attribute to subscriptions, known as the subscription name, can be used instead of the traditional identity to reference a subscription.

The subscription name must be unique within the stream for which the subscription applies. On first registration, the traditional identity must be specified. The subscription name can be specified on the first registration or added to the subscription subsequently (at which time the traditional identity must also be specified to tie the two together). When the subscription name has been defined for the subscription, subsequent commands need specify only the subscription name to access (modify or deregister) the subscription. The underlying traditional identity for the subscription can now be changed by specifying the same subscription name with new traditional identity information on a **Register Subscriber** command.

A subscription name can be associated only with a single traditional identity at any one time within any stream of a broker (and in particular, with a single topic at a time), although it is possible to reuse a subscription name for a different subscription after the original has been deregistered, and it is possible to use the same subscription name in different streams on the same broker or on any stream of a different broker for different subscriptions. Subscription names are arbitrary character strings with no length limit. Subscription names that start with "MQ" are reserved for internal use.

If multiple applications require access to the same subscription, the broker can manage their access by using subscriber identities. A subscribing application can specify a subscriber identity (an application-generated unique string) on a **Register Subscriber** or **Deregister Subscriber** command to add or remove itself from the broker-managed list of interested applications. The concepts of shared and exclusive access to a subscription are supported by the broker in much the same way as shared and exclusive access to WebSphere MQ objects is supported by a queue manager. The use of subscription identities on a subscription does not effect the publication of matching publications to that subscription; a single copy of each publication is still sent to the defined subscription queue no matter how many subscriber identities are currently registered. Deregistering with a subscription identity from a subscription does not delete the subscription unless the subscription identity list becomes empty as a result of removing the identity from that list. Identity names that start with "MQ" are reserved for internal use.

The message descriptor

This section gives information about the values you must set in the message descriptor (MQMD) for messages that you send to the broker. It also explains the values that the broker sets in the message descriptor for publication messages it forwards to subscribers.

Messages sent to the broker

This section shows the values set for fields in the MQMD for messages sent to the broker.

Report

See *MsgType* (below), and "Error handling by the broker" on page 83.

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 succeeds partially, 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 is ever sent.

Format

Set to MQFMT_RF_HEADER.

MsgId

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

CorrelId

Specifies the *CorrelId* that can optionally be included as part of the

The message descriptor

subscriber's identity. When used with the MQRO_PASS_CORREL_ID option in the *Report* field, it is also 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 publisher or subscriber queue that has the advantage that the *QName* parameter can be omitted from the message text. If, however, responses are to be sent to a different queue, the *QName* parameter is needed.

ReplyToQMgr

Queue manager for responses.

Note that a putting application can leave this field blank (the default value), in which case the local queue manager puts its own name in this field.

Expiry

Expiry of the subscription or publication.

Publications forwarded by the broker

This section shows the values set for fields in the MQMD for publications sent by the broker to subscribers.

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

Report

Set to MQRO_NONE.

MsgType

Set to MQMT_DATAGRAM.

Expiry

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

Set to MQFMT_RF_HEADER.

MsgId

Set to MQMI_NONE, so that the queue manager generates 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

Set by the publisher or as a resolved value if the publisher specified MQPRI_PRIORITY_AS_Q_DEF.

Persistence

Set by the publisher or as a resolved value if the publisher specified MQPER_PERSISTENCE_AS_Q_DEF.

ReplyToQ

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 registered).

ApplIdentityData

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

PutApplType

Set to MQAT_BROKER.

PutApplName

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

Set to blanks.

Persistence and units of work

Subscriber and publisher registration messages should normally be sent as persistent messages (registrations themselves are always persistent, regardless of the persistence of the messages that caused them). Publication messages can be either persistent or non-persistent. Brokers maintain the persistence and priority of publications as set by the publisher.

When reading messages from stream 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 options set in the queue manager configuration file, qm.ini (or equivalent). This is described in "Broker configuration stanza" on page 102.

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 **Request Update** is used).

Publishers and subscribers can choose whether or not 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 re-startable.

Limitations

This section describes some limitations of WebSphere MQ Publish/Subscribe.

Group messages

Group messages are not supported by WebSphere MQ Publish/Subscribe. If a group message is sent to the broker, it does not cause an error, but the group message flags in the message descriptor are not forwarded by the broker.

Segmented messages

Segmented messages are not supported by WebSphere MQ Publish/Subscribe. If a segmented message is sent to the broker, it is rejected as not valid.

If you want to distribute a segmented message to subscribers, you can publish a short notification that the message is available, offering to accept 'direct requests' for the full message (see "Publish" on page 65).

Cluster queues

Stream queues must not be cluster queues.

Data conversion of MQRFH structure

You might have a client application (publisher or subscriber) running on a version of WebSphere MQ that does not support data conversion of the MQRFH structure. The application can pass publish/subscribe messages to other queue managers provided that CONVERT(NO) is specified on the sending channel.

Using the Application Messaging Interface

The WebSphere MQ Application Messaging Interface (AMI) provides a simple interface that application programmers can use without needing to understand all the options available in the WebSphere MQ Message Queue Interface (MQI). The options that are needed in a particular installation are defined by a system administrator, using services and policies.

The AMI has functions to generate the most commonly used publish/subscribe command messages, and to receive a publication from the broker. It is available for the C, C++, and Java™ programming languages. The name of the function (or method) depends on the programming language being used. In the case of C, there are two sets of functions: the high-level interface and the object interface.

AMI publish/subscribe functions

The AMI publish/subscribe functions are:

- Publish command
- Register Subscriber command
- Deregister Subscriber command
- Receive a publication

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 parameters in the command message, such as the *topic*. Other parameters in the command message are specified by the AMI *service* that you use to send the message (the service is set up by the system administrator). You can modify these parameters by changing the appropriate name/value elements before sending the command message; 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 7, “Publish/Subscribe command messages,” on page 57.

There are no AMI functions to generate **Delete Publication**, **Deregister Publisher**, **Register Publisher**, 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 *WebSphere MQ Application Messaging Interface* book for details of how to use the functions mentioned above (including the name/value element helper functions).

Chapter 4. Writing publisher applications

Publisher applications communicate with the broker using command messages in the RF Header format (or the equivalent functions in the Application Messaging Interface). Publishers can register with the broker before they start publishing information, they can register implicitly with their first publication, or they can choose not to register. When they have finished publishing information, they can deregister with the broker. They can also delete retained publications. This chapter discusses the following topics:

- “Registering with the broker”
- “Publishing information” on page 37
- “Deleting information” on page 38
- “Deregistering with the broker” on page 39

You can see an example of a publisher application in Chapter 9, “Sample programs,” on page 91.

The only configuration the administrator has to perform before you can define an application as a potential publisher is to set up the necessary security authorization to enable the application to put messages to the required stream queues, and, if explicit registration is required, to send messages to the broker’s control queue (see “Broker queues” on page 99).

Registering with the broker

Publisher applications can register their intention to publish with a broker.

There are two ways for a publisher to register with a broker:

- The publisher can send a **Register Publisher** command message to the broker’s control queue (SYSTEM.BROKER.CONTROL.QUEUE) to indicate that a publisher will be, or is capable of, publishing data on one or more specified topics. This message can also be sent by another application on a publisher’s behalf. This command is described in “Register Publisher” on page 70.
- The publisher can register with the broker implicitly when it sends its first **Publish** command message to a stream queue at the broker (such as SYSTEM.BROKER.DEFAULT.STREAM or SAMPLE.BROKER.RESULTS.STREAM). This command is described in “Publish” on page 65. However, if the broker is not currently aware of the stream specified, a **Register Publisher** command message is necessary for the broker to recognize the stream queue.

A publishing application might not know if a stream is supported by a particular broker. In this case it is recommended that the publisher issues the **Register Publisher** command message and waits for a response that indicates that the stream is known to the broker, before sending the first **Publish** command message.

An application can register with the same broker more than once, and can also register with many different brokers. An application that is already registered as a subscriber can also register as a publisher. This is the case in the sample application (see “Sample application” on page 15). The results service registers as a *subscriber* to the events published by the match simulators, and as a *publisher* of the latest scores.

Choosing not to register

Publishers do not have to register with a broker. This saves the programming overhead of performing the registration, and of deregistering when the publisher has finished. However, other applications cannot find out about unregistered publishers because they do not appear in metatopics (see “Metatopics” on page 151 for information about these). Unregistered publishers can send **Publish** command messages to the broker, specifying that they do not want the broker to perform an implicit registration, provided that both of the following statements are true:

- The publisher does not need to be listed in the metatopics
- The publisher’s stream is already known to the broker

Options you can specify when registering as a publisher

When a publisher registers with a broker, it must specify the topics that it is going to publish information about. It can specify the name of more than one topic, but it cannot use wildcards to specify a range of topics.

Queue name

A publisher is required to specify a queue when it registers and also when it issues **Publish** command messages (unless it specifies the `MQPS_NO_REGISTRATION` option). This is the queue to which any **Request Update** command messages sent directly by a subscriber to this publisher are normally sent. The publisher specifies the queue to which any responses from the broker are to be sent using the *ReplyToQ* and *ReplyToQMgr* parameters; this queue can also be the publisher’s queue.

Selecting a stream

You can specify the name of the stream to which the specified topics apply. If you do not specify this, the `SYSTEM.BROKER.DEFAULT.STREAM` is assumed.

Publisher identity

The identity of the publisher consists of the name of the queue and queue manager that it uses, as described in “Publisher and subscriber identity” on page 27. You can specify these names when you register as a publisher. If you do not specify these names, the names of the reply-to queue and reply-to queue manager specified in the message descriptor (MQMD) of the command message are used for this instead.

You can also specify that you want to use the correlation identifier in the message descriptor as part of the publisher’s identity.

A publisher can register anonymously. In this case its identity is not divulged by the broker, except to subscribers to metatopics that have additional authority (see “Authorized metatopics” on page 153).

Registration scope

If the broker is part of a network, the publisher can specify whether it wants its publications (a) sent to subscribers who have registered local subscriptions on that broker only (a local publication), or (b) distributed to other brokers in the network and sent to all subscribers, including those on that broker, who have registered global subscriptions (a global publication).

Registration expiry

Publisher registrations do not expire, even if you specify a value for *Expiry* field of the message descriptor. The value you set for *Expiry* might however cause the command message to expire before it is processed by the broker.

Broker restart

Publisher registrations and retained publications are maintained across broker restarts.

Changing an application's registration

If a publisher has registered, it can use the **Register Publisher** command message again to increase the range of topics it wants to publish for, or to change the options for topics that it has already registered for. This command should be sent to the broker's control queue.

Publishing information

When an application wants to publish some information, it sends a **Publish** command message to the stream queue at the broker. This command is described in "Publish" on page 65.

The publisher must specify the topic to which the publication applies. If a publication matches several subscriptions for which a subscriber is registered, only one copy of the publication is sent to the subscriber for all matching subscriptions. The publisher can also specify the name of a stream; however, this is not necessary if the message is put to the correct stream queue at the broker.

If the publisher is not registered with the broker for those topics, the broker automatically registers the publisher when it receives this message, unless you tell it not to (see "Choosing not to register" on page 36).

If an application is registered as both a publisher and a subscriber for a topic, it can use an option when publishing to say that it does not want to receive a copy of this publication.

Publication data

Publishers can include the publication data in the message, or they can refer to it.

Including data in the message

Publication data is usually appended to the **Publish** command message, following the *NameValueString* of the MQRFH header, as shown in "Publication data" on page 53. The characteristics of the data are defined in the *Encoding*, *CodedCharSetId* and *Format* fields of the MQRFH header. Alternatively, string data can be contained within the *NameValueString*.

Referring to data in the message

Publishers can make information available to subscribers directly, without going through the broker. The publisher needs to advertise the fact that it is publishing information about a topic, and that it is willing to receive direct requests for this information from subscribers.

There are two ways that a subscriber can find out about this information:

- From a publication received in a normal way.
The publisher can use a normal publication to advertise the fact that it has more information about a topic (for example, a large file in several different formats). The publisher should also specify the topic name to be used (which could be the same, or different) and where the subscriber can find the information.
- From a subscription to the metatopics.

Publishing information

The publisher can register with the broker specifying that it accepts direct requests for information about a topic. Subscribers that request information about publishers (metatopics) will discover the names of publishers who publish on this topic. See “Metatopics” on page 151 for information about metatopics.

Retained publications

When a publication specifies that it is to be retained, any previously retained publication for this stream and topic combination is replaced, so that the information is always at the latest level. See “Retained publications” on page 14 for information about retained publications.

Mixing retained and non-retained publications on the same topic in a stream is not recommended. If an application does this and publishes a non-retained publication, any previously retained publication is still retained.

It is not recommended for two or more applications to publish retained publications to the same topic and stream. If two applications do publish a retained publication about the same topic on the same stream simultaneously, it is difficult to determine which publication is retained. If these publishers use two different brokers, it is possible that different retained publications could be active at different brokers for the same topic and stream.

Expiry of retained publications

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

Publishing locally and globally

Publishers can specify that they want a publication to be published locally. If they do not specify this, the publication is made available globally through all the brokers in the network. Local publications can be received only by subscribers who register local subscriptions at the same broker as the publisher. Local retained publications are retained only at this broker.

Applications can publish and subscribe locally to the same topic and stream at different brokers. Each broker deals with the publications and subscriptions in isolation from the other brokers.

Mixing local and global publications and subscriptions to the same topic and stream is not recommended. A local publication is not delivered to a subscriber registered globally, even if they are at the same broker.

Deleting information

Publishers can request that the broker delete retained publications for specified topics. To do this, send the **Delete Publication** command message to the stream queue at the broker to tell it to delete its copy of any data for the specified topics. This command is described in “Delete Publication” on page 58.

The application needs the same authority to delete publications as it needs to publish messages for the specified stream. You do not have to be a registered publisher to be able to delete publications.

If you want to delete some of the information that was originally published in a message that covered more than one topic, the broker deletes the publication only for the topics you specify, and retains the rest.

If different publishers publish data on the same stream and topics, the data that is deleted might have originated from a different publisher.

You can also specify if you want to delete retained publications published locally at the broker, or those published globally.

Deregistering with the broker

When a publisher that is registered with a broker no longer wants to publish information on a topic, it can use the **Deregister Publisher** command message to deregister with the broker. This message should be sent to the `SYSTEM.BROKER.CONTROL.QUEUE`. This command is described in “Deregister Publisher” on page 60.

This command can be used if the publisher registered with the broker explicitly using **Register Publisher**, or implicitly using **Publish**. A publisher cannot deregister if it chose not to register in the first place.

The application must specify one of the following:

- Deregister for all topics for which it was registered.
- Deregister for a subset of the topics for which it is registered if it wants to continue publishing on other topics. It must specify one or more topics, and it can use wildcards.

You must specify the stream name for these topics, unless it is the default (`SYSTEM.BROKER.DEFAULT.STREAM`).

You must also specify the name of the publisher’s queue and queue manager.

The publisher registration must be deregistered by the same user that registered it originally, unless the deregistering application is allowed to put the message as the appropriate user (for example using alternate user authority to open the `SYSTEM.BROKER.CONTROL.QUEUE` for that user).

Chapter 5. Writing subscriber applications

Subscriber applications communicate with the broker using command messages in the RF Header format (or the equivalent functions in the Application Messaging Interface). Subscribers need to register with a broker before they can start receiving publications. They can also request certain types of publication from the broker or directly from the publisher.

This chapter discusses the following topics:

- “Registering as a subscriber”
- “Requesting information” on page 43
- “Deregistering as a subscriber” on page 44

You can see an example of a subscriber application in Chapter 9, “Sample programs,” on page 91.

Registering as a subscriber

Subscriber applications need to register their interest in receiving publications with a broker. Before you can define an application as a potential subscriber, you must set up the necessary security authorization to enable the application to do the following:

- Put a message to the broker’s control queue.
- Browse the required stream queues.
- Put a message to the subscriber queue that will be used to receive publications.

Send the **Register Subscriber** command message to the `SYSTEM.BROKER.CONTROL.QUEUE` to register as a subscriber. This command is described in “Register Subscriber” on page 72.

Your application should send this message to a broker’s control queue (see “Broker queues” on page 99). to indicate that it wants to subscribe to the topics specified in the message. Alternatively, an application can send this message to register on behalf of another application that wants to subscribe. If an application subscribes on behalf of another application, the user ID of the subscribing application is used. The application needs alternate user authority if a different user ID is used. An application that has already registered as a publisher can also register as a subscriber.

An application can register with the same broker more than once, and can also register with many different brokers.

When a subscriber has registered with a broker, the subscription is persistent and survives broker and queue manager restarts, regardless of the persistence of the **Register Subscriber** command message.

When a subscriber registers with the broker, it must specify the topics that it is interested in. It can specify the name of more than one topic, and it can also use wildcards to specify a range of topics (described in “Topics” on page 9). If a subscriber has many (different) registrations that match the topic of a publication, only one copy of the publication is sent to it.

Subscriber queues

A subscriber queue is the queue where publications for that subscriber are sent. The subscriber specifies the name of the queue when it registers a subscription. If the subscriber is at the same queue manager as the broker, the subscriber's queue name must not be the same as that of the stream. Such a subscription is rejected. Even if the subscriber's and broker's queue managers are different, it is strongly recommended that you use different names for the queues.

If a subscribing application registers multiple subscriptions (for the same or different streams), it can choose whether all **Publish** command messages are sent to the same queue, or whether **Publish** command messages for different subscriptions go to different queues.

The queue name, queue manager name and correlation identifier (if one is specified) of a subscriber's queue or a subscription name are used by the broker to identify the subscriber (as described in "Publisher and subscriber identity" on page 27). When the broker publishes information about subscribers, if a subscriber has registered several subscriptions for the same stream that are all to be sent to the same queue, and the subscriptions are not distinguished with different correlation identifiers, the subscriber appears as a single application.

If publications for different subscriptions are sent to different queues, or use a different *CorrelId*, the broker regards these as being from multiple subscribers (even though the subscriber might be a single application).

Options you can specify when registering as a subscriber

The options that a subscriber specifies when registering determine which publications (if any) are sent to it by the broker. Any previously retained publications for the topics specified are sent immediately after registration (unless the subscriber specifies new publications only, which are those published *after* the subscriber registered with the broker).

Alternatively, the subscriber can request that it is not sent any publications about a topic unless it asks for them using the **Request Update** command message. This method is applicable where publications have been retained, and an application might want to know the latest information about a topic.

Queue name

The queue where messages for a subscriber should be sent is called the subscriber queue. This queue must not be a temporary dynamic queue. The subscriber specifies the name of the queue when it registers a subscription.

Selecting a stream

You can specify the name of the stream to which the specified topics apply. If you do not specify this, the `SYSTEM.BROKER.DEFAULT.STREAM` is used.

You can also request that publication messages that are sent to the subscriber include the name of the stream to which the publication applies, even if the publisher did not include the name in the publication.

Subscriber identity

The identity of the subscriber consists of a subscription name or the name of the queue and queue manager that it uses, as described in "Publisher and subscriber identity" on page 27. You can specify these names when you register as a subscriber. If you do not specify these names, the following, specified in the

message descriptor (MQMD) of the command message, are used instead: the names of the reply-to queue and reply-to queue manager, and, optionally, the correlation identifier.

You can also use the correlation identifier in the message descriptor as part of the subscriber's identity. You might need to do this if, for example, the broker publishes information about subscribers, and a subscriber has registered several subscriptions for the same stream that are all to be sent to the same queue. If the subscriptions are not distinguished with different correlation identifiers, the subscriber appears as a single application.

If the different subscriptions are to be sent to different queues, the broker believes that these are from multiple subscribers even though the subscriber might be a single application.

If required, you can tell the broker that the identity of the subscriber should not be divulged by the broker when the broker publishes information about subscribers (unless the request comes from a subscriber with additional authority).

Subscription scope

If the broker is part of a network, the subscriber can specify whether it wants to subscribe to local publications sent to the local broker only, or whether it wants its subscription distributed to other brokers in the network.

Subscription expiry

The values you set for the *Expiry* attribute in the message descriptor (MQMD) of the **Register Subscriber** command message determines when the subscription expires. This is measured from the time the subscription request is put. This means that the message could expire before the subscriber is registered with the broker. If this is set to MQEI_UNLIMITED, the subscription does not expire, and the subscriber continues to receive publications until it explicitly deregisters.

Broker restart

Subscriber registrations are maintained across broker restarts. Any subsequent publications for the specified topics are forwarded to the subscriber, including any that arrived while the broker was inactive.

Changing an application's registration

When a subscriber has registered, it can use the **Register Subscriber** command message again to increase the range of topics that it wants to receive information for, or to change the options for topics that it has already registered for.

When a subscription is reregistered, the values you set for the *Expiry* attribute in the message descriptor (MQMD) of the **Register Subscriber** command message determines when the subscription expires. This is measured from the time the subscription request is put. Thus the **Register Subscriber** command message can be used to refresh a subscription before it expires.

Requesting information

A subscriber can request information from the broker, or directly from a publisher.

Requesting information from the broker

A subscriber can request a retained publication on a specified topic from the broker. To do this, it uses the **Request Update** command message, which is

Requesting information

described in “Request Update” on page 80. Applications usually do this if, when they registered with the broker, they asked to be sent publications on request only. If the broker has a retained publication for the topic specified, it is sent to the subscriber.

This command message can also be sent by a subscriber that did not register in this way, to request that the latest copy of a publication be sent to it. This might be necessary if a subscriber has already seen a publication, but has failed without saving it, and on restart wants to see it again.

This command message can be satisfied only by a retained publication at the broker (see “State and event information” on page 14). If the broker to which this message is sent has no retained publication for the topic specified, the request fails.

Requesting information from a publisher

Under some circumstances, subscribers can request information directly from a publisher without involving the broker.

A publisher can specify that it is willing to receive direct requests for information from other applications. In this case, the publisher must make its queue and queue manager names (and possibly correlation identifier) known to subscribers by including them in a publication that advertises the availability of other publications on direct request.

Alternatively, subscribers can subscribe to information about publishers (called metatopics). They can discover the names of publishers who are willing to accept direct requests for publications on this topic. (See “Metatopics” on page 151 for information about metatopics.)

The subscriber can use this information to send a normal WebSphere MQ message (using the MQI) directly to the publisher. The publisher can then use the MQI to send the publication directly to the subscriber.

Deregistering as a subscriber

When a subscriber no longer wants to receive publications on a topic, send the **Deregister Subscriber** command message to the broker’s control queue. This command is described in “Deregister Subscriber” on page 62.

This tells the broker to stop sending publications, about the topics specified, to the subscriber.

An application must specify one of the following:

- Deregister for all topics for which it was registered.
- Deregister for a subset of the topics for which it is registered if it still wants to receive publications on other topics. It must specify one or more topics. If the original subscription used wildcards, it must be deregistered using the same wildcard topic.

You must specify the stream name for these topics, unless it was the default (SYSTEM.BROKER.DEFAULT.STREAM).

You must also specify the name of the subscriber’s queue and queue manager, unless they are the same as the reply-to queue and reply-to queue manager in the message descriptor of the command message. The subscription must be

deregistered by the same user that registered it originally, unless the deregistering application is allowed to put the **Deregister Subscriber** message as the appropriate user (for example, using alternate user authority to open the `SYSTEM.BROKER.CONTROL.QUEUE` for that user and *CorrelId*).

Chapter 6. Format of command messages

Applications use command messages to communicate with the broker when they want to publish or subscribe to information. These messages use the WebSphere MQ Rules and Formatting Header (RF Header). Each message or response starts with an MQRFH structure, which includes a *NameValueString*. This consists of a succession of tag names and values (name/value pairs), which define the type of command the message represents and any options that apply to it. In the case of a **Publish** command message, the MQRFH header is usually followed by the data being published, in a format defined in the MQRFH structure. Alternatively, string publication data can be included within the *NameValueString*, using appropriate tag names and values defined by the publisher.

This chapter discusses the following topics:

- “MQRFH – Rules and formatting header”
- “Publish/Subscribe name/value strings” on page 51
- “Publication data” on page 53

The name/value pairs that define the parameters needed for the command messages are detailed in Chapter 7, “Publish/Subscribe command messages,” on page 57.

If you are using the WebSphere MQ Application Messaging Interface (AMI) to communicate with the broker, you don’t need to understand all the information in this chapter. The AMI constructs and interprets the RF Header and its name/value pairs (see “Using the Application Messaging Interface” on page 32). However, you might find it useful to read this chapter, in particular the section on publication data.

MQRFH – Rules and formatting header

The following table summarizes the fields in the structure.

Table 2. Fields in MQRFH

Field	Description	Page
<i>StrucId</i>	Structure identifier	48
<i>Version</i>	Structure version number	48
<i>StrucLength</i>	Total length of MQRFH including string containing name/value pairs	48
<i>Encoding</i>	Numeric encoding	48
<i>CodedCharSetId</i>	Coded character set identifier	49
<i>Format</i>	Format name	49
<i>Flags</i>	Flags	49
<i>NameValueString</i>	String containing name/value pairs	49

The MQRFH structure defines the format of the rules and formatting header. This header can be used to send string data in the form of name/value pairs.

Rules and formatting header

The format name of an MQRFH structure is MQFMT_RF_HEADER. The fields in the MQRFH structure and the name/value pairs are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes the MQRFH, or by those fields in the MQMD structure if the MQRFH is at the start of the application message data.

Character data in the MQRFH (including the *NameValueString* field) must belong to a single-byte character set (SBCS). The user data that follows *NameValueString* can belong to any supported character set (SBCS or DBCS).

This structure is supported in the following environments: AIX, DOS client, HP-UX, Linux, OS/2®, z/OS, Solaris, Windows client, Windows, and Windows 2000.

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_1

Version-1 rules and formatting header structure.

The initial value of this field is MQRFH_VERSION_1.

StrucLength (MQLONG)

Total length of MQRFH including string containing name/value pairs.

This is the length in bytes of the MQRFH structure, including the *NameValueString* field at the end of the structure. The length does *not* include any user data that follows the *NameValueString* field.

To avoid problems with data conversion of the user data in some environments, make sure that *StrucLength* is a multiple of four.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *NameValueString* field:

MQRFH_STRUC_LENGTH_FIXED

Length of fixed part of MQRFH structure.

The initial value of this field is MQRFH_STRUC_LENGTH_FIXED.

Encoding (MQLONG)

Numeric encoding.

This specifies the representation used for numeric values in the user data (if any) that follows the string containing the name/value pairs. 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)

Coded character set identifier.

This specifies the coded character set identifier of character strings in the user data (if any) that follows the string containing the name/value pairs.

Note: When a message is put, this field must be set to the nonzero value that specifies the character set of the user data. If this is not done, it is not possible to convert the message using the MQGMO_CONVERT option when the message is retrieved.

The initial value of this field is 0.

Format (MQCHAR8)

Format name.

This specifies the format name of the user data (if any) that follows the string containing the name/value pairs.

Pad the name 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 MQRFH structure. Do not specify a name with leading or embedded blanks.

The initial value of this field is MQFMT_NONE.

Flags (MQLONG)

Flags.

The following can be specified:

MQRFH_NONE

No flags.

The initial value of this field is MQRFH_NONE.

NameValueString (MQCHARn)

String containing name/value pairs.

This is a variable-length character string containing name/value pairs in the form:

```
name1 value1 name2 value2 name3 value3 ...
```

Each name or value must be separated from the adjacent name or value by one or more blank characters; these blanks are not significant. A name or value can contain significant blanks by prefixing and suffixing the name or value with the double-quote character; all characters between the open double-quote and the matching close double-quote are treated as significant. In the following example, the name is FAMOUS_WORDS, and the value is Hello World:

```
FAMOUS_WORDS "Hello World"
```

A name or value can contain any characters other than the null character (which acts as a delimiter for *NameValueString*). However, to assist interoperability, an application might prefer to restrict names to the following characters:

Rules and formatting header

- First character: upper case or lower case alphabetic (A through Z, or a through z), or underscore.
- Second character: upper case or lower case alphabetic, decimal digit (0 through 9), underscore, hyphen, or dot.

If a name or value contains one or more double-quote characters, the name or value must be enclosed in double quotes, and each double quote within the string must be doubled, for example:

```
Famous_Words "The program displayed ""Hello World"""
```

Names and values are case sensitive, that is, lowercase letters are not considered to be the same as uppercase letters. For example, FAMOUS_WORDS and Famous_Words are two different names.

The length in bytes of *NameValueString* is equal to *StrucLength* minus MQRFH_STRUC_LENGTH_FIXED. To avoid problems with data conversion of the user data in some environments, make sure that this length is a multiple of four. *NameValueString* must be padded with blanks to this length, or terminated earlier by placing a null character following the last value in the string. The null and bytes following it, up to the specified length of *NameValueString*, are ignored.

Note: Because the contents and length of the *NameValueString* field are not fixed, no initial value is given for this field, and it is omitted from the “Structure definition in C.”

Table 3. Initial values of fields in MQRFH

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQRFH_STRUC_ID	'RFHb' (See note 1)
<i>Version</i>	MQRFH_VERSION_1	1
<i>StrucLength</i>	MQRFH_STRUC_LENGTH_FIXED	32
<i>Encoding</i>	MQENC_NATIVE	See note 2
<i>CodedCharSetId</i>	None	0
<i>Format</i>	MQFMT_NONE	'bbbbbbbb'
<i>Flags</i>	MQRFH_NONE	0

Notes:

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 MQRFH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQRFH MyRFH = {MQRFH_DEFAULT};
```

Structure definition in C

```
typedef struct tagMQRFH {  
    MQCHAR4  StrucId;          /* Structure identifier */  
    MQLONG   Version;          /* Structure version number */  
    MQLONG   StrucLength;      /* Total length of MQRFH including string  
                               containing name/value pairs */  
    MQLONG   Encoding;         /* Numeric encoding */  
};
```

```

MQLONG  CodedCharSetId; /* Coded character set identifier */
MQCHAR8  Format;         /* Format name */
MQLONG  Flags;          /* Flags */
} MQRFH;

```

Publish/Subscribe name/value strings

The MQRFH format is used to encode command messages that are sent to the WebSphere MQ Publish/Subscribe broker. The *NameValueString* field within the RF header contains name/value pairs that describe the command to be carried out by the broker. If the command being issued is a **Publish** command, publication data (in a format defined by the publisher) can follow the *NameValueString* field.

The *NameValueString* can contain any number of name/value pairs, but only those in which the tag-name begins with the characters 'MQPS' are recognized by the broker. Other name/value pairs (which can be defined by the publisher to encode publication data, for instance) are ignored by the broker.

The first occurrence of an 'MQPS' tag-name must be MQPSCommand, followed by a tag-value that identifies the command to be carried out. Subsequent 'MQPS' tag-names and their values identify any options for that command (if they occur *before* the MQPSCommand tag-name, the command fails).

Each name or value must be separated from the adjacent name or value by one or more blank characters. The C header file **cmqpsc.h** defines tag-names and values that can be used by publisher and subscriber applications when building command messages to be sent to the broker. Blank enclosed versions of the constants are provided to simplify construction of a *NameValueString*. For example, topics are specified using a tag-name of MQPSTopic, and the following three constants are provided in the **cmqpsc.h** header file:

```

#define MQPS_TOPIC      "MQPSTopic"
#define MQPS_TOPIC_B    " MQPSTopic "
#define MQPS_TOPIC_A    ' ','M','Q','P','S','T','o','p','i','c',' '

```

The MQPS_TOPIC constant is not enclosed by blanks. If it is used to build a *NameValueString*, the application must add blanks between tag-names and values. The version of the constant with the '_B' suffix includes the necessary blanks. The version with the '_A' suffix also includes the blanks, but is in character array form. These constants are most suited for initialization of a C structure that is being used to define a fixed layout of a *NameValueString*.

For example, the **Delete Publication** command can be issued to delete retained publications throughout the broker network. A topic of '*' matches all topics within the stream that the command is sent to, so using this deletes all retained publications. A *NameValueString* to perform such a command can be constructed as follows.

If the constants without blanks are used, the blanks must be inserted, for example:

```

MQCHAR DeleteCmd[] =
    MQPS_COMMAND " " MQPS_DELETE_PUBLICATION " " MQPS_TOPIC " *";

```

This can be simplified by using the constants with blanks, for example:

```

MQCHAR DeleteCmd[] =
    MQPS_COMMAND_B MQPS_DELETE_PUBLICATION_B MQPS_TOPIC_B "*";

```

A subscribing application might need to analyze a *NameValueString*, for instance to determine the topic associated with each publication it receives. One approach is to

Name/value strings

break down the entire *NameValueString* into its constituent parts. An illustration of this approach is given in the results service sample application (see Chapter 9, “Sample programs,” on page 91). A simpler approach is to use the `sscanf` in the C runtime library to determine the position of the `MQPSTopic` tag-name in the string. Since `sscanf` automatically strips away white space, the `MQPS_TOPIC` constant (without the blanks) is needed here.

Options using string constants

Some commands have options associated with them, which are also specified to the broker by name/value pairs. They are defined in the C header file **cmqpsc.h**. Multiple registration options, publication options and delete options are allowed, so the `MQPSRegOpts`, `MQPSPubOpts` and `MQPSDelOpts` tag-names can be repeated with different values. The effect is cumulative.

For example, to register an anonymous local publisher on topic ‘News’, the following *NameValueString* is needed:

```
MQPSCommand  RegPub
MQPSRegOpts  Anon
MQPSRegOpts  Local
MQPSTopic    News
```

Options using integer constants

Alternatively, an application can specify all its options using a single name/value pair. This might be useful when the presence or absence of an option is conditional upon program logic. In this case, the combined set of options can be specified as a single decimal numeric value. The C header file **cmqcfc.h** provides corresponding integer constants for all the options. In the previous example, the constants `MQREGO_ANONYMOUS` and `MQREGO_LOCAL` are relevant. The anonymous option has a decimal value of 2, and the local option has a decimal value of 4, so the following *NameValueString* is equivalent:

```
MQPSCommand  RegPub
MQPSRegOpts  6
MQPSTopic    News
```

Sending a command message with the RFH structure

Figure 15 on page 53 shows how the RFH structure (including the *NameValueString*) is appended to the Message Descriptor to send a message to a broker. In this case, the message is to register a subscriber to the topic “IBM® Stock Price”. Part of the message descriptor is shown, together with the message data that consists of the RFH structure. Pad the *NameValueString* to a multiple of four bytes.

Details of the name/value pairs for all the command messages are given in Chapter 7, “Publish/Subscribe command messages,” on page 57.

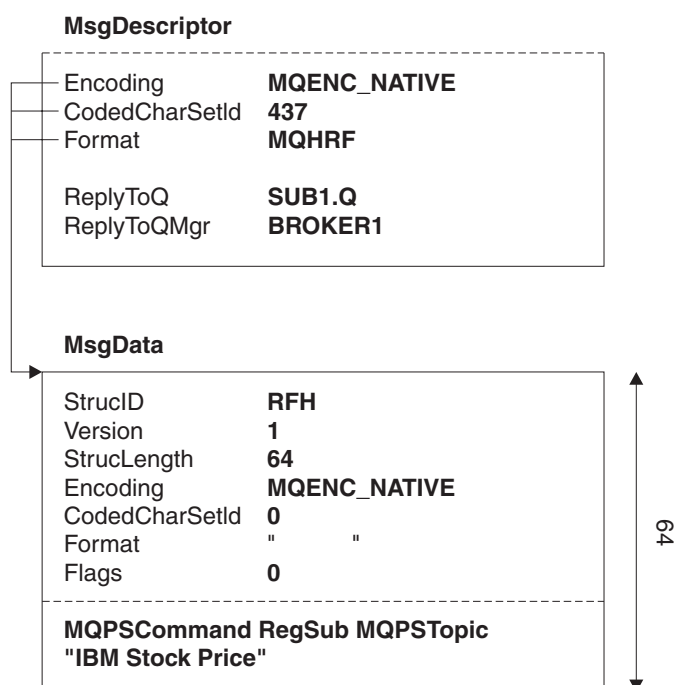


Figure 15. Message descriptor and RFH structure. The message descriptor indicates that the subscriber has nominated its subscriber queue to be the same as its reply queue. It also defines the encoding and CCSID of the RFH structure, which follows as the message data. The encoding and CCSID fields in the RFH structure are not set, because there is no data following the RFH structure (compare with Figure 16 on page 54). Note that the length of the RFH structure includes the NameValueString (which contains the name/value pairs defining the Register Subscriber command). The topic string is quoted because it contains significant blanks.

Publication data

Publication data, or *UserData*, can be appended to a **Publish** command message after the *NameValueString*. The format of the data is defined in the *Encoding*, *CodedCharSetId* and *Format* fields of the MQRFH header. Alternatively, publication data can be included within the *NameValueString*, by means of user defined name/value pairs (which must not begin with the characters 'MQ'), or the system provided *StringData* and *IntegerData* tags. More details are given in “Publish” on page 65.

Figure 16 on page 54 shows how publication data can be appended to the RFH structure. Note how the encoding, CCSID and format of the publication data are defined in the RFH structure. In Figure 17 on page 54 the publication data is included within the *NameValueString*, and in Figure 18 on page 55, the format of the publication data is defined by the user.

Double-byte character sets

Publication data can use a single-byte character set (SBCS) or a double-byte character set (DBCS) code page. However, if a publishing application publishes information in SBCS, a subscribing application receiving that information must not request the data to be converted to DBCS (because the MQRFH header would be converted as well, and the header must be SBCS).

Publication data

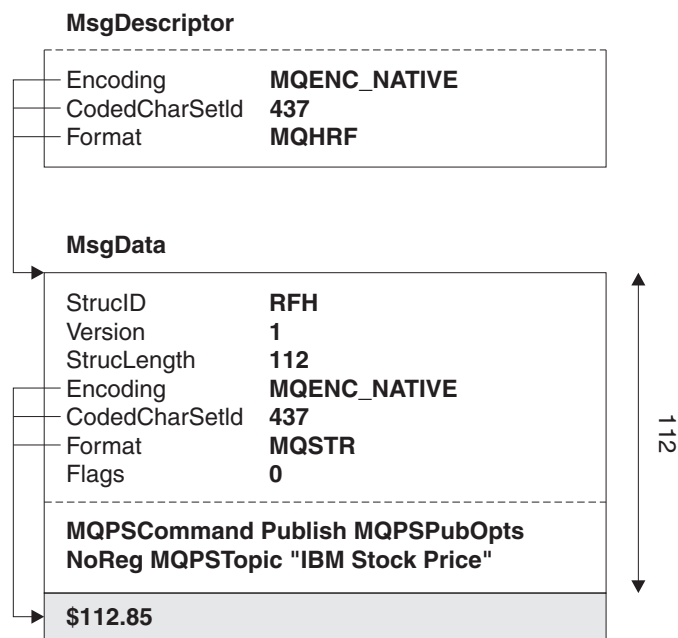


Figure 16. Publication data after the RFH structure. In this example, the publication data (\$112.85) that is being published as string data in MQSTR format, is appended to the message after the NameValueString. Note that the RFH StrucLength includes the NameValueString, but not the publication data. The message descriptor defines the encoding, CCSID and format of the RFH structure, which in turn defines the encoding, CCSID and format of the publication data.

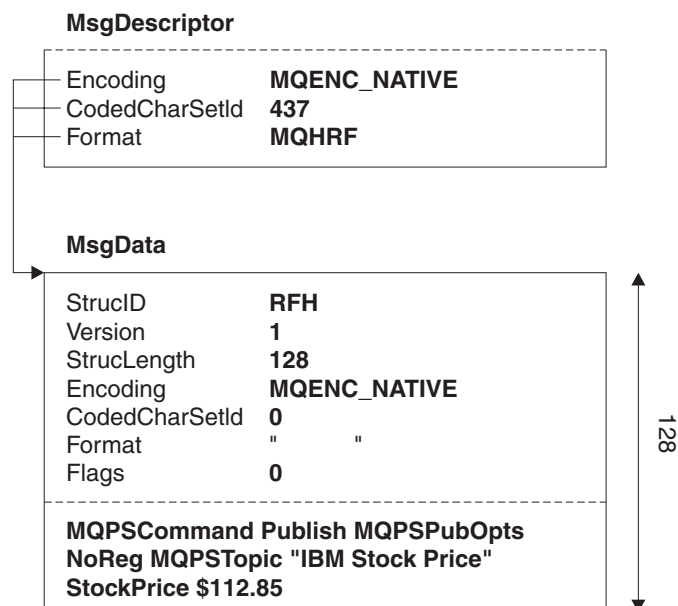


Figure 17. Publishing data within the NameValueString. Publication data can be included within the NameValueString, by means of one or more user-defined name/value pairs, as shown in this example. The encoding and CCSID fields in the RFH structure are not set, because there is no following data. The receiving application must parse the RFH structure to extract the publication data.

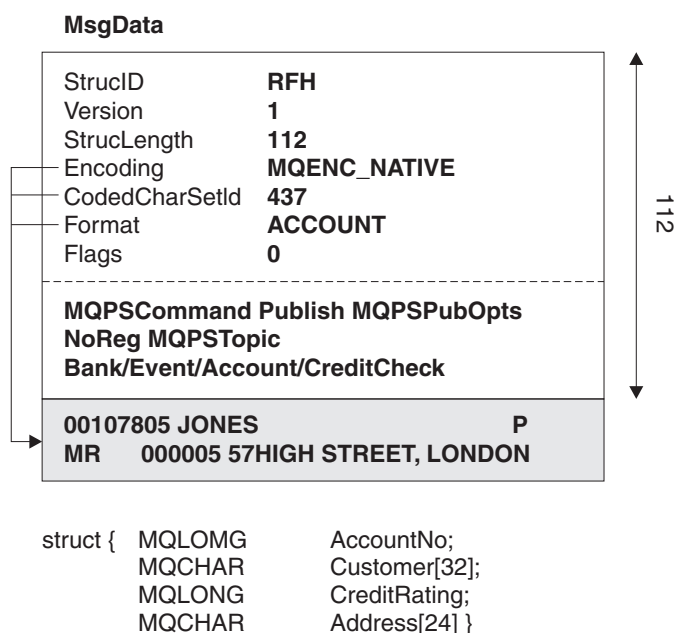


Figure 18. User-defined publication data. In this example, the format of the publication data is set to a user-defined format, ACCOUNT, which contains character and numeric data. When the broker processes Publish messages, it converts the RFH header (but not the publication data) to its own CCSID and encoding. The user must write a data conversion routine if the publication is sent to subscribing applications that use a different CCSID or encoding.

In the previous examples, it is assumed that the subscribing or publishing application is running in an explicit code page of 437. However, for reasons of portability, applications can use the special CCSID value MQCCSI_Q_MGR in the message descriptor if they are using the same code page as the queue manager they are communicating with. In addition, the special value MQCCSI_INHERIT can be set in the CCSID field of the RF header to indicate that the publication data is in the same CCSID as the character data in the header.

Figure 19 on page 56 shows how the CCSID for the RF header and the publication data can be inherited from the message descriptor.

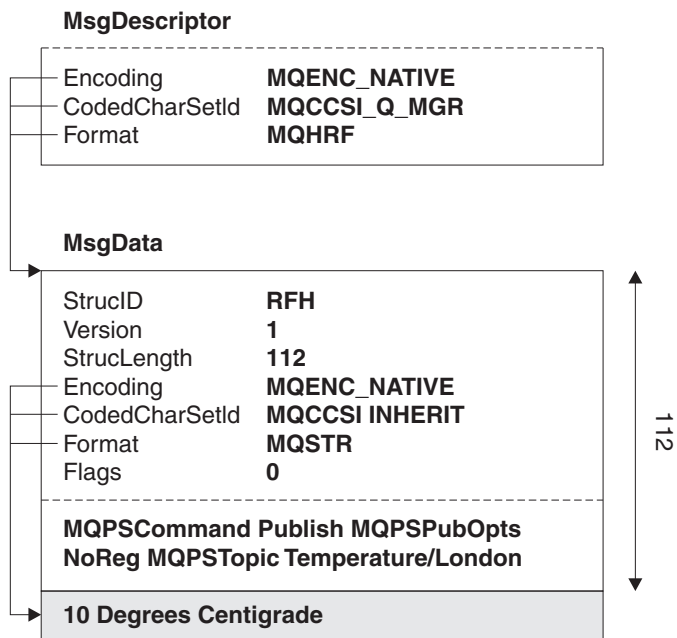


Figure 19. Inheriting the CCSID. The message descriptor uses the special value MQCCSI_Q_MGR to indicate that data within the RFH structure is in the same CCSID as the queue manager. The value of MQCCSI_INHERIT in the RFH structure indicates that the same CCSID is used for the publication data.

Chapter 7. Publish/Subscribe command messages

This chapter describes the name/value pairs that define the parameters needed for the following command messages:

- “Delete Publication” on page 58
- “Deregister Publisher” on page 60
- “Deregister Subscriber” on page 62
- “Publish” on page 65
- “Register Publisher” on page 70
- “Register Subscriber” on page 72
- “Request Update” on page 80

Chapter 6, “Format of command messages,” on page 47 describes how to send these command messages using the Rules and Formatting header.

If you are using the WebSphere MQ Application Messaging Interface (AMI) to communicate with the broker, you don’t need to understand all the information in this chapter. The AMI constructs and interprets the RF Header and its name/value pairs (see “Using the Application Messaging Interface” on page 32). However, you might find it useful to read this chapter to see what options are available in each command message. Some of the options are directly accessible through parameters in an AMI function such as **amPublish**. Others can be accessed using an AMI name/value element helper function such as **amMsgGetElement**, or a macro such as **AmMsgGetStreamName**.

Delete Publication

The **Delete Publication** command message is sent from a publisher (or another broker) to a broker's stream queue to tell it to delete its copy of any retained publications for the specified topics within that stream.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "DeletePub" (string constant: MQPS_DELETE_PUBLICATION)

Command must be the first parameter in the *NameValueString*.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: The topic for which published information is to be deleted. Wild cards can be used to delete several topics.

Topic can be repeated for as many topics as required.

Optional parameters

DeleteOptions

name: "MQPSDelOpts" (string constant: MQPS_DELETE_OPTIONS)
value: The following delete options can be specified:
"Local"
 (string constant: MQPS_LOCAL, integer constant: MQDELO_LOCAL).

Retained publications published locally at this broker (that is, with RetainPub and Local specified) are deleted. Those published globally (that is, with RetainPub but not Local specified) are not deleted, even if they were published at this broker.

The default if *DeleteOptions* is omitted is that global retained publications are deleted at all brokers in the network, but local retained publications are not deleted. Mixing local and global publications to the same topic and stream is not recommended. See "Publish" on page 65 for more information about retained local publications.

StreamName

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)
value: The name of the publication stream for the specified *Topic(s)*.

The default value is the name of the stream queue to which the message is sent.

Example

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

```
MQPSCommand    DeletePub
MQPSStreamName SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic      "Sport/Soccer/State/LatestScore/Team1 Team2"
```

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3075	MQRCCF_INCORRECT_STREAM	Stream name does not match stream queue.
3087	MQRCCF_DEL_OPTIONS_ERROR	Invalid delete options supplied.

Deregister Publisher

The **Deregister Publisher** command message is sent from a publisher, or another application on a publisher's behalf, to a broker's control queue to indicate that a publisher is no longer publishing data on the topics contained in the message.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "DeregPub" (string constant: MQPS_DEREGISTER_PUBLISHER)

Command must be the first parameter in the *NameValueString*.

Optional parameters

QueueManagerName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)
value: The publisher's queue manager name.

For a message sent by a publisher, if *QueueManagerName* is not present, it defaults to the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it matches a publisher that registered with a blank queue manager name.

For a message sent by a broker, *QueueManagerName* is omitted.

QueueName

name: "MQPSQName" (string constant: MQPS_Q_NAME)
value: The publisher's queue name.

For a message sent by a publisher, if *QueueName* is not present, it defaults to the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

For a message sent by a broker, the *QueueName* parameter is omitted.

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)
value: The following registration options can be specified:

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY).

The *CorrelId* in the MQMD (which must not be zero) is part of the publisher's identity.

"DeregAll"

(string constant: MQPS_DEREGISTER_ALL, integer constant: MQREGO_DEREGISTER_ALL)

All topics registered for this publisher are to be deregistered. If this option is set, the *Topic* parameter must be omitted.

The default if *RegistrationOptions* is omitted is that no options are set. In this case, the *Topic* parameter is required.

StreamName

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)
value: The name of the publication stream for the specified *Topic(s)*.

The default value is SYSTEM.BROKER.DEFAULT.STREAM.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)

value: The topic being deregistered. Wildcards are allowed.

If DeregAll is specified in *RegistrationOptions*, the *Topic* parameter must be omitted. Otherwise, it is required, and can optionally be repeated for as many topics as needed.

Example

Here is an example of a *NameValueString* for a **Deregister Publisher** command message. This deregisters a publisher for all topics it has registered that match *Stock/**. The publisher's identity, including the *CorrelId*, is taken from the defaults in the MQMD.

```
MQPSCommand  DeregPub
MQPSRegOpts  CorrelAsId
MQPSTopic    Stock/*
```

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3073	MQRCCF_NOT_REGISTERED	Publisher or subscriber not registered.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3082	MQRCCF_UNKNOWN_STREAM	Stream not defined to broker and cannot be created.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.

Deregister Subscriber

The **Deregister Subscriber** command message is sent from a subscriber, another application on a subscriber's behalf, or another broker, to a broker's control queue to indicate that it no longer wants to subscribe to the topics specified.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "DeregSub" (string constant: MQPS_DEREGISTER_SUBSCRIBER)

Command must be the first parameter in the *NameValueString*.

Optional parameters

QueueManagerName

name: "MQPSQMGrName" (string constant: MQPS_Q_MGR_NAME)
value: The subscriber's queue manager name.

If *QueueManagerName* is not present, it defaults to the *ReplyToQMGr* name in the message descriptor (MQMD). If the resulting name is blank, it matches a subscriber that registered with a blank queue manager name.

QueueName

name: "MQPSQName" (string constant: MQPS_Q_NAME)
value: The subscriber's queue name.

If *QueueName* is not present, it defaults to the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)
value: The following registration options can be specified:

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY).

The *CorrelId* in the MQMD (which must not be zero) is part of the subscriber's identity.

"DeregAll"

(string constant: MQPS_DEREGISTER_ALL, integer constant: MQREGO_DEREGISTER_ALL).

All topics registered for this subscriber are to be deregistered. If this option is set, the *Topic* parameter must be omitted.

"FullResp"

(string constant: MQPS_FULL_RESPONSE, integer constant: MQREGO_FULL_RESPONSE).

When *FullResp* is specified, all the attributes of the subscription are returned in the response message if the command does not fail. See details under **Register Subscriber**. When *FullResp* is specified, *DeregAll* is not permitted in the **Deregister Subscriber** command or multiple topics.

"LeaveOnly"

(string constant: MQPS_LEAVE_ONLY, integer constant: MQREGO_LEAVE_ONLY).

When `LeaveOnly` is specified with a `SubIdentity` that is in the identity set for the subscription, the `SubIdentity` is removed from the identity set for the subscription, but the subscription is not removed from the broker, even if the resulting identity set is empty.

If the `SubIdentity` value is not in the identity set the command fails.

`LeaveOnly` must be specified with a `SubIdentity`.

If neither `LeaveOnly` nor `SubIdentity` are specified, the subscription is removed regardless of the contents of the identity set for the subscription.

"VariableUserId"

(string constant: `MQPS_VARIABLE_USER_ID`, integer constant: `MQREGO_VARIABLE_USER_ID`).

If the subscription to be deregistered has `VariableUserId` set this must be set when the **Deregister Subscriber** command is sent to indicate which subscription is being deregistered. Otherwise, the `userid` of the **Deregister Subscriber** command will be used to identify the subscription. This is overridden (along with the other subscriber identifiers) if a subscription name is supplied.

The default if this tag is omitted is that no options are set. In this case, the *Topic* parameter is required.

StreamName

name: `"MQPSSStreamName"` (string constant: `MQPS_STREAM_NAME`)

value: The name of the publication stream for the specified *Topic(s)*.

The default value is `SYSTEM.BROKER.DEFAULT.STREAM`.

SubIdentity

name: `"MQPSSubIdentity"` (string constant: `MQPS_SUBSCRIPTION_IDENTITY`)

value: Subscription identity.

See **Register Subscriber** for more details. If the `SubIdentity` is in the identity set for the subscription, it is removed from the set.

If the identity set becomes empty as a result of this, the subscription is removed from the broker (unless `LeaveOnly` is specified).

If the identity set still contains other identities, the subscription is not removed from the broker and publication flow is not interrupted.

SubName

name: `"MQPSSubName"` (string constant: `MQPS_SUBSCRIPTION_NAME`)

value: Subscription name.

The `SubName` value takes precedence over all other identifier fields except the `userid` unless `VariableUserId` is set on the subscription itself.

If `VariableUserId` is not set, the **Deregister Subscriber** command succeeds only if the `userid` of the command message matches that of the subscription.

If a subscription exists that matches the traditional identity of this command but has no `SubName`, the **Deregister Subscriber** command fails.

If an attempt is made to deregister a subscription that has a `SubName` using a command message that matches the traditional identity but with no `SubName` specified, the command succeeds.

Deregister Subscriber

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)

value: The topic being deregistered. Wild cards are allowed, but a specified topic string must match exactly the corresponding string that was originally specified in the **Register Subscriber** command.

If DeregAll is specified in *RegistrationOptions*, the *Topic* parameter must be omitted. Otherwise, it is required, and can optionally be repeated for as many topics as needed. Topics specified can be a subset of those for which the subscriber is registered if it wants to retain subscriptions to the other topics.

Example

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

```
MQPSCommand      DeregSub
MQPSRegOpts       CorrelAsId
MQPSSStreamName   SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic         Sport/Soccer/State/LatestScore/*
```

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3073	MQRCCF_NOT_REGISTERED	Publisher or subscriber not registered.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3082	MQRCCF_UNKNOWN_STREAM	Stream not defined to broker and cannot be created.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.
3153	MQRCCF_SUB_NAME_ERROR	Subscription exists but has no SubName.
3154	MQRCCF_SUB_IDENTITY_ERROR	SubIdentity is not in the identity set for the subscription.

Publish

The **Publish** command message is used to publish information on specific topics. It is sent from either:

- From a publisher (or another broker) to a broker's stream queue
- From a broker to a subscriber's stream queue

Publication data can be appended to the message, after the *NameValueString*, in a format defined by the *Encoding*, *CodedCharSetId* and *Format* fields in the MQRFH header.

Alternatively, publication data can be included within the *NameValueString*, using name/value pairs such as the *StringData* and *IntegerData* parameters defined below, or any other name/value pairs defined by the publisher (provided the tag-name does not begin with the characters 'MQ').

Required parameters

Command

name: "MQPSCCommand" (string constant: MQPS_COMMAND)
value: "Publish" (string constant: MQPS_PUBLISH)

Command must be the first parameter in the *NameValueString*.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: The topic that categorizes this publication. No wild cards are allowed.

Topic can be repeated for as many topics as required. For example, an application might publish information under topic 'Topic 1', which is then enhanced to publish extra information. The new publications might use topics 'Topic 1' and 'Topic 1 enhanced', so that subscribers to 'Topic 1 enhanced' would be sure to get the additional information, while existing subscribers to 'Topic 1' could still access the basic information in the same publication.

Optional parameters

IntegerData

name: "MQPSIntData" (string constant: MQPS_INTEGER_DATA)
value: Optional publication data as an integer.

The meaning is as defined by the publisher. *IntegerData* can be repeated, interspersed with *StringData* tags if required, to send publication data in any manner defined by the publisher.

PublicationOptions

name: "MQPSPubOpts" (string constant: MQPS_PUBLICATION_OPTIONS)
value: The following publication options can be specified:

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQPUBO_CORREL_ID_AS_IDENTITY).

The *CorrelId* in the MQMD (which must not be zero) is part of the publisher's identity (for messages sent by a publisher to a broker). For messages sent from a broker to a subscriber, this option is not changed by the broker.

"IsRetainedPub"

(string constant: MQPS_IS_RETAINED_PUBLICATION, integer constant: MQPUBO_IS_RETAINED_PUBLICATION).

Can be set only 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. A subscriber can receive such a publication immediately after registering (or later if a publication has been retained at another broker that is temporarily inaccessible). It can also be received in response to a **Request Update** command.

The broker sets this option only if the subscriber registered with the InformIfRet option.

"NoReg"

(string constant: MQPS_NO_REGISTRATION, integer constant: MQPUBO_NO_REGISTRATION).

Valid only if the recipient is a broker.

If the publisher is not already registered with the broker as a publisher for this stream and topic, this option stops the broker from performing an implicit registration. If the publisher is already registered, the registration is unchanged, and has no effect on this publication.

"OtherSubsOnly"

(string constant: MQPS_OTHER_SUBSCRIBERS_ONLY, integer constant: MQPUBO_OTHER_SUBSCRIBERS_ONLY).

Valid only if the recipient is a broker.

Allows simpler processing of conference-type applications. It tells the broker not to send the publication to the publisher even if he has subscribed. For example, a group of applications can all subscribe to the same topic (for example, "Conference"). Using this option, each application can publish information into the conference without themselves receiving the information.

"RetainPub"

(string constant: MQPS_RETAIN_PUBLICATION, integer constant: MQPUBO_RETAIN_PUBLICATION).

Valid only if the recipient is a broker.

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 its current subscribers.

The default is that no publication options are set.

PublishTimestamp

name: "MQSPubTime" (string constant: MQPS_PUBLISH_TIMESTAMP)
value: Optional publication timestamp set by the publisher.

This is of length 16 characters in the format:

YYYYMMDDHHMMSSTH

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

QMgrName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)

value: The publisher's queue manager name.

For a message sent by a publisher, the default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it represents a publisher that can be reached by resolving *QName* at the broker.

For a message sent by a broker, *QMgrName* is present only if it was explicitly included by the publisher. (Note that it is not removed by the broker if the publisher has registered with Anon)

QName

name: "MQPSQName" (string constant: MQPS_Q_NAME)

value: The publisher's queue name.

For a message sent by a publisher, the default is the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case (unless *PublicationOptions* specifies NoReg and not OtherSubsOnly).

For a message sent by a broker, *QName* is present only if it was explicitly included by the publisher. (Note that it is not removed by the broker if the publisher has registered with Anon)

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)

value: The registration options listed below can be specified, subject to the following conditions:

If NoReg is not specified in *PublicationOptions*:

- If the publisher is already registered, the registration options are changed to the values specified, if this tag is present. If it is not present, the registration options are unchanged.
- If the publisher is not already registered, an implicit registration is performed. The registration options are those specified by the *RegistrationOptions* parameter, if it is present. If it is not present, no options are set.

If NoReg is specified in *PublicationOptions*, any current registration has no effect and it is not changed. *RegistrationOptions* can be specified.

If Local is specified in *RegistrationOptions*, the publication is restricted to local subscribers and any other valid options are not acted on by the broker.

The following registration options can be set:

"Anon"

(string constant: MQPS_ANONYMOUS, integer constant: MQREGO_ANONYMOUS).

Valid only if the recipient is a broker.

Tells the broker that the identity of the publisher is not to be divulged, except to subscribers with additional authority.

This option (or the lack of it) overrides the option setting for any previous publication on the same topics (or publisher registration).

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY).

Publish

The *CorrelId* in the MQMD (which must not be zero) is part of the publisher's identity. This option is assumed if *CorrelAsId* is set in the *PublicationOptions*.

"DirectReq"

(string constant: MQPS_DIRECT_REQUESTS, integer constant: MQREGO_DIRECT_REQUESTS).

Tells the recipient that the publisher is willing to receive direct requests for publication information from other applications (not just from the broker).

The publisher's queue and queue manager names can be included in a **Publish** message sent by a publisher, so that the names are visible to the subscriber.

This option (or the lack of it) overrides the option setting for any previous publication on the same topics (or registration in the case of a publisher to a broker, or the value returned in the response to a subscriber registration).

This option must not be set if *Anon* is also set.

"Local"

(string constant: MQPS_LOCAL, integer constant: MQREGO_LOCAL).

Valid only if the recipient is a broker.

Tells the broker that publications published by this publisher should be sent only to subscribers that registered at this broker specifying *Local*.

SequenceNumber

name: "MQPSSeqNum" (string constant: MQPS_SEQUENCE_NUMBER)

value: Optional sequence number set by the publisher.

This 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 stream and topic are published to different interconnected brokers, it is the responsibility of the publisher to ensure that sequence numbers, if used, are meaningful.

StreamName

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)

value: The name of the publication stream for the specified *Topic(s)*.

This defaults to the name of the stream queue to which the message is sent if sent to a broker, or an unspecified stream name if the message is sent to a subscriber. A subscriber can request that the broker always include *StreamName* in **Publish** messages by specifying "InclStreamName" when it registers.

StringData

name: "MQPSStringData" (string constant: MQPS_STRING_DATA)

value: Optional publication data as a character string.

The meaning and format are as defined by the publisher. *StringData* can be repeated, interspersed with *IntegerData* tags if required, to send publication data in any manner defined by the publisher.

Example

Here are some examples of a *NameValueString* for a **Publish** command message. The first example is for an Event Publication sent by the match simulator in the sample application to indicate that a match has started, with 'No Registration' specified for the publisher:

```
MQPSCCommand    Publish
MQPSPubOpts     NoReg
MQPSSStreamName SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic       Sport/Soccer/Event/MatchStarted
```

The second example is for a State Publication, so 'Retain Publication' is specified as well. In this case the results service is publishing the latest score in the match between Team1 and Team2.

```
MQPSCCommand    Publish
MQPSPubOpts     RetainPub
MQPSPubOpts     NoReg
MQPSSStreamName SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic       "Sport/Soccer/State/LatestScore/Team1 Team2"
```

In both examples the publication data (the names of the teams, or the latest score) follows the *NameValueString*, as string data in MQSTR format.

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3075	MQRCCF_INCORRECT_STREAM	Stream not defined to broker and cannot be created.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.
3084	MQRCCF_PUB_OPTIONS_ERROR	Invalid publication options supplied.

Register Publisher

The **Register Publisher** command message is sent from a publisher (or another application on a publisher's behalf) to a broker's control queue to indicate that a publisher will be, or is capable of, publishing data on one or more specified topics.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "RegPub" (string constant: MQPS_REGISTER_PUBLISHER)

Command must be the first parameter in the *NameValueString*.

Topic

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: The topic for which the publisher will be providing publications. Wild cards are not allowed.

Topic can be repeated for as many topics as required.

Optional parameters

QMgrName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)
value: The publisher's queue manager name.

For a message sent by a publisher, the default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it represents a publisher that can be reached by resolving *QName* at the broker.

For a message sent by a broker, *QMgrName* is present only if *DirectReq* is set in the *RegistrationOptions* tag.

QName

name: "MQPSQName" (string constant: MQPS_Q_NAME)
value: The publisher's queue name.

For a message sent by a publisher, the default is the *ReplyToQ* name in the message descriptor (MQMD), which must not be blank in this case.

For a message sent by a broker, *QName* is present only if *DirectReq* is set in the *RegistrationOptions* tag.

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)
value: The following registration options can be specified:

"Anon"

(string constant: MQPS_ANONYMOUS, integer constant: MQREGO_ANONYMOUS)

Tells the broker that the identity of the publisher is not to be divulged, except to subscribers with additional authority.

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor, MQMD, (which must not be zero) is part of the publisher's identity.

"DirectReq"

(string constant: MQPS_DIRECT_REQUESTS, integer constant: MQREGO_DIRECT_REQUEST)

Tells the recipient that the publisher is willing to receive direct requests for publication information from other applications (that is, not just from the broker).

This option must not be set if Anon is also set.

"Local"

(string constant: MQPS_LOCAL, integer constant: MQREGO_LOCAL)

Tells the broker that publications published by this publisher should be sent only to subscribers that registered on this broker specifying Local.

If the *RegistrationOptions* parameter is omitted and the publisher is already registered, its registration options are unchanged. If the publisher is not already registered, the default is that no registration options are set.

StreamName

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)

value: The name of the publication stream for the specified *Topic(s)*.

The default value is SYSTEM.BROKER.DEFAULT.STREAM.

Example

Here is an example of a *NameValueString* for a **Register Publisher** command message. The publisher is registering with the 'Direct Requests' option, for the Stock/IBM topic on the default stream. The queue name and queue manager name are specified so that subscribers can respond directly to the publisher.

```
MQPSCmd    RegPub
MQPSRegOps DirectReq
MQPSQMGrName Broker1
MQPSQName  STOCK.IBM.PUBLISHER.QUEUE
MQPSTopic  Stock/IBM
```

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3082	MQRCCF_UNKNOWN_STREAM	Stream not defined to broker and cannot be created.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.

Register Subscriber

The **Register Subscriber** command message is sent from a subscriber (or another application on its behalf), or a broker, to a broker's control queue to indicate that it wants to subscribe to the topics specified.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "RegSub" (string constant: MQPS_REGISTER_SUBSCRIBER)

Command must be the first parameter in the *NameValueString*.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: The topic for which the subscriber wants to receive publications. Wild cards are allowed.

Topic can be repeated for as many topics as required.

Optional parameters

QMgrName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)
value: The subscriber's queue manager name.

The default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it represents a publisher that can be reached by resolving *QName* at the broker.

QName

name: "MQPSQName" (string constant: MQPS_Q_NAME)
value: The subscriber's queue name.

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

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)
value: The following registration options can be specified:

"AddName"

(string constant: MQPS_ADD_NAME, integer constant: MQREGO_ADD_NAME)

If AddName is specified, the SubName field is mandatory.

If AddName is specified for an existing subscription that matches the traditional identity of this **Register Subscriber** command, but with no current SubName value, the SubName specified in this command is added to the subscription.

If a subscription already exists by this SubName, or if a matching subscription (as identified by the traditional identity) with a different SubName exists on this stream, the command fails.

"Anon"

(string constant: MQPS_ANONYMOUS, integer constant: MQREGO_ANONYMOUS)

Tells the broker that the identity of the publisher is not to be divulged, except to subscribers with additional authority.

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor, MQMD, (which must not be zero) is part of the subscriber's identity.

"DupsOK"

(string constant: MQPS_DUPLICATES_OK, integer constant: MQREGO_DUPLICATES_OK)

Setting this option results in the occasional delivery of duplicate publications to the subscriber. The subscriber should be tolerant of such duplicate publications.

The advantage this option provides is reduced overhead in the broker that can enhance performance.

"FullResp"

(string constant: MQPS_FULL_RESPONSE, integer constant: MQREGO_FULL_RESPONSE)

If a response message is requested and this option is specified, all attributes of a subscription are returned in the response message of any command that does not fail. When using MQRFH messages, the NameValueString of the response is in the following format. First the standard response fields (space delimited):

```
MQPSCompCode    <value>
MQPSReason      <value>
MQPSReasonText  <value>
```

followed by all subscription fields, if defined, (again space delimited) as they appear after any registration changes were made as a result of a **Register Subscriber** command, or before any changes were made as a result of a **Deregister Subscriber** command.

```
MQPSCommand      <value>
MQPSSubName      <value>    (Might not be present)
MQPSTopic        <value>
MQPSQMgrName     <value>
MQPSSQName       <value>
MQPSCorrelId     <value>    (Might not be present,
                             48-byte character representation of hex chars)
MQPSUserId       <value>
MQPSRegOpts      <value>    (Can be repeated)
MQPSSubIdentity  <value>    (Might not be present, can be repeated)
MQPSSubUserData  <value>    (Might not be present)
```

FullResp is valid only when the command message (**Register Subscriber** or **Deregister Subscriber**) refers to only a single subscription. Therefore, only a single topic is permitted in the command, otherwise the command fails.

When PCF structures are used, the above data is returned in an equivalent PCF structured message.

If no response is returned (for example, MQMT_DATAGRAM), this option is ignored.

"IncludeStreamName"

(string constant: MQPS_INCLUDE_STREAM_NAME, integer constant: MQREGO_INCLUDE_STREAM_NAME)

Each **Publish** message that is sent must include the *StreamName* parameter. The broker does this by adding the appropriate name/value pair to the *NameValueString* of the message. The *NameValueString* is extended if necessary.

If this option is not set, *StreamName* is included only if it was specified explicitly by the publisher.

"InformIfRet"

(string constant: MQPS_INFORM_IF_RETAINED, integer constant: MQREGO_INFORM_IF_RETAINED)

The broker informs the subscriber if a publication is retained when a **Publish** message is sent. It does this by adding the name/value pair "MQSPubOpts IsRetainedPub" to the *NameValueString* of the message (after the *StreamName* if that has been added in accordance with the *IncludeStreamName* option).

Use this option if a subscriber needs to distinguish between new publications and old publications that were retained by the broker before the subscription was made. If this option is specified, the broker always adds the name/value pair to a publication sent in response to a **Request Update** command.

"JoinExcl"

(string constant: MQPS_JOIN_EXCLUSIVE, integer constant: MQREGO_JOIN_EXCLUSIVE)

Indicates that the specified SubIdentity should be added as the exclusive member of the identity set for the subscription, and that no other identities can be added to the set.

If the subscription is currently exclusively locked, the command fails if the identity with the exclusive lock is not the one in this command message; if it is the same identity, the command succeeds, but returns a warning of MQRCCF_ALREADY_JOINED.

If the identity has already joined 'shared' and is the sole entry in the set, the set is changed to an exclusive lock held by this identity. Otherwise, if the subscription currently has other identities in the identity set (with shared access) the command fails.

If an application attempts to register with a SubIdentity and the userid differs from that currently registered with the subscription, it fails if VariableUserId is not set on the original subscription or, if it is set, the userid of the command message is checked for authority to browse the stream queue and put to the subscriber's queue; if it does not have sufficient authority, the command fails.

This option is valid only when SubIdentity is specified.

"JoinShared"

(string constant: MQPS_JOIN_SHARED, integer constant: MQREGO_JOIN_SHARED)

Indicates that the specified SubIdentity should be added to the identity set for the subscription.

If the subscription currently has zero or more members in the identity set and none match this identity, and it is not exclusively locked (see "JoinExcl"), the command succeeds and adds this identity to the set.

If the identity already has a shared entry for this subscription, the command succeeds but returns a warning of `MQRCCF_ALREADY_JOINED`.

If the subscription is currently locked exclusively, `MQRCCF_SUBSCRIPTION_LOCKED` is returned, unless the identity that has the subscription locked is the same identity as the one in this command message, in which case the lock is atomically modified to a shared lock.

If an application attempts to register with a `SubIdentity`, and the `userid` differs from the one currently registered with the subscription, it fails if `VariableUserId` is not set on the original subscription. If it is set, the `userid` of the command message is checked for authority to browse the stream queue and put to the subscriber's queue; if it does not have sufficient authority, the command fails.

This option is valid only when `SubIdentity` is specified.

"Local"

(string constant: `MQPS_LOCAL`, integer constant: `MQREGO_LOCAL`)

Tells the broker that the subscription is local and should not be distributed to other brokers in the network. Only publications published at this node by a publisher specifying `Local` are sent to this subscriber.

"Locked"

(string constant: `MQPS_LOCKED`, integer constant: `MQREGO_LOCKED`)

Can be set only by the broker.

This subscription is currently locked (someone has exclusive access to the subscription). This option is automatically set and unset against the subscription as identities `JoinExcl` and `leave`. Anyone inquiring on the subscription (either by `metatopics` or the `FullResp` option) can see this option set and the current identity set, thus identifying the owner of the lock.

"NewPubsOnly"

(string constant: `MQPS_NEW_PUBLICATIONS_ONLY`, integer constant: `MQREGO_NEW_PUBLICATIONS_ONLY`)

Tells the broker that no currently retained publications are to be sent, only new publications. If a subscriber re-registers and changes this option so that it is not set, it is possible that a publication that has already been sent to it is sent to it again.

"NoAlter"

(string constant: `MQPS_NO_ALTERATION`, integer constant: `MQREGO_NO_ALTERATION`)

When `NoAlter` is specified, the **Register Subscriber** command does not modify an existing matching subscription's attributes. This option has no effect when a subscription is created. This is the converse of the default behavior for a subsequent subscription that matches the identity of an existing subscription overwriting any modifiable attributes of the original subscription.

Register Subscriber

If a SubIdentity is supplied along with a Join option, the identity is added to the identity set (if possible) irrespective of the NoAlter option, because this applies to a subscription's attributes not its current state.

"NonPers"

(string constant: MQPS_NON_PERSISTENT, integer constant: MQREGO_NON_PERSISTENT)

Any publication sent from a broker to a subscriber that specified this option is sent as a non-persistent message, irrespective of the persistence of the publication message received by the broker.

If you set this option, you cannot set "Pers", "PersAsPub", or "PersAsQueue".

"Pers"

(string constant: MQPS_PERSISTENT, integer constant: MQREGO_PERSISTENT)

Any publication sent from a broker to a subscriber that specified this option is sent as a persistent message, irrespective of the persistence of the publication message received by the broker.

If you set this option, you cannot set "NonPers", "PersAsPub", or "PersAsQueue".

"PersAsPub"

(string constant: MQPS_PERSISTENT_AS_PUBLISH, integer constant: MQREGO_PERSISTENT_AS_PUBLISH)

Any publication sent from a broker to a subscriber that specified this option is sent with the persistence of the original publication. This is the default option.

If you set this option, you cannot set "NonPers", "Pers", or "PersAsQueue".

"PersAsQueue"

(string constant: MQPS_PERSISTENT_AS_Q, integer constant: MQREGO_PERSISTENT_AS_Q)

Any publication sent from a broker to a subscriber that specified this option is sent with the persistence specified on the subscriber queue. The persistence is derived from the DEFPSIST setting of the subscriber queue definition local to the broker: for example, the transmission queue to the subscriber's queue manager if the subscriber's queue manager is remote from the broker's queue manager.

If you set this option, you cannot set "NonPers", "Pers", or "PersAsPub".

"PubOnReqOnly"

(string constant: MQPS_PUBLISH_ON_REQUEST_ONLY, integer constant: MQREGO_PUBLISH_ON_REQUEST_ONLY)

Indicates that the subscriber polls only for information with **Request Update**. The broker is not to send unsolicited messages to the subscriber.

This option is not propagated if the broker sends this subscription to other brokers in the network. Publications are sent to it in the normal

way, and these publications must specify `RetainPub` to be eligible for return in response to a **Request Update** message.

"VariableUserId"

(string constant: `MQPS_VARIABLE_USER_ID`, integer constant: `MQREGO_VARIABLE_USER_ID`)

When `VariableUserId` is specified, the identity of the subscriber (queue name, queue manager name, and correlation identifier) is not restricted to a single userid. This allows any user to modify or deregister the subscription when they have suitable authority. To add this option to an existing subscription the command must come from the same userid as the original subscription itself.

If a **Register Subscriber** command message specifying this option refers to an existing subscription with this option set, and the userid of this message differs from the original subscription, the command succeeds only if the userid of the new command message has authority to browse the stream queue, and put authority to the subscriber queue of the modified subscription (that is, existing Publish/Subscribe authority check for a subscriber). On successful completion, future publications to this subscriber are put to the subscriber's queue with the new userid.

If a **Register Subscriber** command message without this option set refers to an existing subscription with this option set, the option is removed from this subscription, and the userid of the subscription is now fixed. If at this time a subscriber already exists that has the same identity (queue name, queue manager name, and correlation identifier), but with a different userid associated to it, the command fails.

If the Registration Options parameter is omitted and the subscriber is already registered, its registration options are unchanged. If the subscriber is not already registered, the default is that no registration options are set.

StreamName

name: `"MQPSStreamName"` (string constant: `MQPS_STREAM_NAME`)
value: The name of the publication stream for the specified *Topic(s)*.

The default value is `SYSTEM.BROKER.DEFAULT.STREAM`.

SubIdentity

name: `"MQPSSubIdentity"` (string constant: `MQPS_SUBSCRIPTION_IDENTITY`)
value: The subscription identity.

Used to represent an application with an interest in a subscription. The broker maintains a set of subscriber identities for each subscription; each subscription can allow its identity set to hold only a single identity, or unlimited identities (see the `"JoinShared"` and `"JoinExcl"` options).

A **Register Subscriber** command that specifies the `JoinShared` or `JoinExcl` option adds the `SubIdentity` to the subscription's identity set, if it is not already there. Any alteration of the subscription's attributes as the result of a **Register Subscriber** command where a `SubIdentity` is specified succeeds only if it would be the only member of the set of identities for this subscription. Otherwise the command fails.

If no `SubIdentity` is specified the alteration succeeds irrespective of a possible set of identities.

Register Subscriber

The maximum length of a SubIdentity is defined by MQ_SUB_IDENTITY_LENGTH.

SubName

name: "MQPSSubName" (string constant: MQPS_SUBSCRIPTION_NAME)
value: The subscription name.

If SubName is specified, the subscription name is the single field used to identify a subscription, overriding the traditional identity.

If a subscription already exists that matches the traditional identity of this command, but has no SubName, the **Register Subscriber** command fails unless the AddName option is specified.

If an existing named subscription is to be altered by another **Register Subscriber** command specifying the same SubName, and the values of Topic, QMgrName, QName and CorrelId in the new command match a different existing subscription (with or without a SubName defined), the command fails: two subscription names cannot refer to a single subscription.

Altering or deregistering a subscription that has a SubName is also allowed by a command message that matches the traditional identity but with no SubName specified.

When a SubName value is specified, only one topic attribute is permitted.

If the underlying topic of the subscription is changed, existing retained publications are sent to the subscriber, whether or not they received them as a result of a previous topic for this subscription.

SubUserData

name: "MQPSSubUserData" (string constant: MQPS_SUBSCRIPTION_USER_DATA)
value: The subscription user data.

Variable length text string. The value is stored by the broker with the subscription but has no influence on publication delivery to the subscriber. The value can be altered by re-registering to the same subscription with a new value. This attribute is for the use of the application.

Example

Here is an example of a *NameValueString* 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.

MQPSCommand	RegSub
MQPSRegOpts	PubOnReqOnly
MQPSRegOpts	CorrelAsId
MQPSStreamName	SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic	Sport/Soccer/State/LatestScore/*

Here is the same message using the equivalent decimal registration options:

MQPSCommand	RegSub
MQPSRegOpts	33
MQPSStreamName	SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic	Sport/Soccer/State/LatestScore/*

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3081	MQRCCF_NOT_AUTHORIZED	Publisher or subscriber not registered.
3082	MQRCCF_UNKNOWN_STREAM	Stream not defined to broker and cannot be created.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.
3152	MQRCCF_DUPLICATE_SUBSCRIPTION	A subscription by this SubName exists or a matching subscription (identified by the traditional identity) with a different SubName exists on this stream.
3153	MQRCCF_SUB_NAME_ERROR	The SubName is invalid: it is of zero length or contains invalid escape sequences.
3154	MQRCCF_SUB_IDENTITY_ERROR	The SubIdentity is not in the identity set for the subscription and neither JoinShared nor JoinExcl was specified.
3155	MQRCCF_SUBSCRIPTION_IN_USE	The subscription has other identities in the identity set, with shared access.
3156	MQRCCF_SUBSCRIPTION_LOCKED	The subscription is locked exclusively by another identity.
3157	MQRCCF_ALREADY_JOINED	The identity already has a shared entry for this subscription.

Request Update

The **Request Update** command message is sent from a subscriber to a broker to request an update publication for the topic specified. This is normally used if the subscriber specified the option "PubOnReqOnly" (publish on request only) when it registered. If the broker has a retained publication for the topic, this is sent to the subscriber. If not, the request fails.

Required parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)
value: "ReqUpdate" (string constant: MQPS_REQUEST_UPDATE)

Command must be the first parameter in the *NameValueString*.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: The topic the subscriber is requesting. Wild cards are allowed, in which case the subscriber might receive multiple retained publications.

Only one occurrence of Topic is allowed in this message.

Optional parameters

QMgrName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)
value: The subscriber's queue manager name.

The default is the *ReplyToQMgr* name in the message descriptor (MQMD). If the resulting name is blank, it matches a publisher with a blank queue manager name (that is, local to the broker).

QName

name: "MQPSQName" (string constant: MQPS_Q_NAME)
value: The subscriber's queue name.

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

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)
value: The following registration options can be specified:

"CorrelAsId"

(string constant: MQPS_CORREL_ID_AS_IDENTITY, integer constant: MQREGO_CORREL_ID_AS_IDENTITY)

The *CorrelId* in the message descriptor (MQMD), which must not be zero, is part of the subscriber's identity.

"VariableUserId"

(string constant: MQPS_VARIABLE_USER_ID, integer constant: MQREGO_VARIABLE_USER_ID)

If the subscription of the request update command has *VariableUserId* set, this must be set when the **Request Update** is sent to indicate which subscription is referred to. Otherwise, the *userid* of the **Request Update** command is used to identify the subscription. This is overridden (along with the other subscriber identifiers) if a subscription name is supplied.

If `VariableUserId` is set and the `userid` differs from that of the subscription, the command succeeds only if the `userid` of the new command message has authority to browse the stream queue, and put authority to the subscriber queue of the subscription (that is, existing Publish/Subscribe authority check for a subscriber), otherwise it fails.

StreamName

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)

value: The name of the publication stream for the specified *Topic(s)*.

The default value is SYSTEM.BROKER.DEFAULT.STREAM.

SubName

name: "MQPSSubName" (string constant: MQPS_SUBSCRIPTION_NAME)

value: The subscription name.

The `SubName` value takes precedence over all other identifier fields except the `userid` unless `VariableUserId` is set on the subscription itself.

If a subscription exists that matches the traditional identity of this command, but has no `SubName`, the **Request Update** command fails.

If an attempt is made to request an update for a subscription that has a `SubName` using a command message that matches the traditional identity, but with no `SubName` specified, the command succeeds.

Example

Here is an example of a *NameValueString* 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.

```
MQPSCommand      ReqUpdate
MQPSRegOpts      CorrelAsId
MQPSStreamName    SAMPLE.BROKER.RESULTS.STREAM
MQPSTopic         Sport/Soccer/State/LatestScore/*
```

Error codes

The following reason codes might be returned in the *NameValueString* of the broker response message to this command, in addition to those shown on page 88.

Reason	Reason text	Explanation
3071	MQRCCF_STREAM_ERROR	Stream name too long or contains invalid characters.
3072	MQRCCF_TOPIC_ERROR	Topic name has an invalid length or contains invalid characters.
3073	MQRCCF_NOT_REGISTERED	Publisher or subscriber not registered.
3074	MQRCCF_Q_MGR_NAME_ERROR	Queue manager name invalid.
3076	MQRCCF_Q_NAME_ERROR	Queue name invalid.
3077	MQRCCF_NO_RETAINED_MSG	No retained message exists for this topic.
3078	MQRCCF_DUPLICATE_IDENTITY	Publisher or subscriber identity already assigned to another user ID.
3080	MQRCCF_CORREL_ID_ERROR	Correlation identifier used as part of identity but is all binary zero.
3081	MQRCCF_NOT_AUTHORIZED	Subscriber not authorized to browse broker's stream queue or subscriber queue.
3082	MQRCCF_UNKNOWN_STREAM	Stream not defined to broker and cannot be created.
3083	MQRCCF_REG_OPTIONS_ERROR	Invalid registration options supplied.

Request Update

Reason	Reason text	Explanation
3153	MQRCCF_SUB_NAME_ERROR	A subscription with no SubName matches the traditional identity of the command.

Chapter 8. Error handling and response messages

Messages sent to and by a broker are subject to exception processing, report generation and dead-letter queue processing in the same way as other WebSphere MQ messages. A message can indicate that a response is not required, is required only if there is an error, only if the command succeeds, or always required.

Response messages can be generated by the broker to each command message issued by a publisher or subscriber. Response messages indicate the success or failure of a request and also the reason for the failure. Responses are given only by the broker to which the messages are initially sent.

The following topics are discussed in this chapter:

- “Error handling by the broker”
- “Response messages” on page 84
- “Broker responses” on page 86
- “Problem determination” on page 88

Error handling by the broker

Any message received by a broker that is not of *Format* MQFMT_RF_HEADER (or MQFMT_PCF in the case of the system management messages described in Part 4, “System programming,” on page 139) is treated as an error. It is written to the dead-letter queue (or discarded, depending on the report options), and an exception report generated, if requested. If a message is of the correct format but has some other error (for example, a syntax error), or if the broker cannot process it correctly (for example, it cannot retain a message), the following happens:

- If a response has been requested, one is generated.
 - If the response cannot be enqueued at the broker, the response is put to the dead-letter queue (responses are always generated with MQRO_NONE).
 - If the response cannot be put to the dead-letter queue, the response is discarded if this is allowed (this depends on the type of response message), depending on the broker configuration parameters.
 - If the response could not be discarded or put to the reply-to queue or the dead-letter queue, the command is backed out, and the input message is put to the dead-letter queue with a *Reason* of MQRCCF_BROKER_COMMAND_FAILED, or discarded, as indicated by the report options. An exception report message is generated, if requested.
 - If the input message or response cannot be put to the dead-letter queue or discarded, the command is backed out and the input message is restored to the input queue if the message is within syncpoint. The input message is retried periodically, and (less frequently) a message is written to the queue manager log to alert the administrator.
- If a response has not been requested, one is not sent, and no further action is appropriate for this message.

If an input message is put to the dead-letter queue, no response and publication messages are sent. It might be appropriate for the input message to be restored and reprocessed when the error has been resolved.

Error handling by the broker

If the message is a **Publish** command message, and there is a problem sending an outgoing message on to a subscriber, the processing is as follows:

- The outgoing message is put to the dead-letter queue, if this is permitted by the broker and queue manager configuration. If the outgoing message cannot be put to the dead-letter queue because of a failure or because it is not permitted by the broker and queue manager configuration, it is discarded if this is permitted by the broker and queue manager configuration.
- If the outgoing message cannot be put to the dead-letter queue or discarded, the input message is restored. The input message is retried after suitable time interval, and (less frequently) a message is written to the log to alert the administrator.

Note: If the broker cannot put a publication message onto a destination queue or the dead-letter queue, and cannot discard the message, the broker continues trying to put the publication message onto the destination queue (at suitable intervals) and does not continue processing subsequent messages.

The dead-letter queue and discard options for nonpersistent messages are specified in queue manager configuration file (qm.ini or equivalent). These options are described in Chapter 10, “Setting up a broker,” on page 99.

Response messages

Each command message that the broker processes can generate a response message. A response message has a similar format to a command message; the *NameValueString* in the MQRFH header contains the response to the command. Response messages are sent to the queue identified by the *ReplyToQ* and *ReplyToQMgr* fields in the message descriptor of the original message.

The *MsgType* and *Report* options specified in the message descriptor of the command message, together with the success or failure of the command, determine whether response messages are sent or not. If no responses are requested, and the command message contains an error, it is discarded.

Notes:

1. If there are multiple errors in a command message, a single response message is generated.
2. Brokers do not request publishers or subscribers to generate responses.

Message descriptor for response messages

When the broker sends a response message, all the fields of the message descriptor are set to their default values, except the following:

CorrelId

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.

Expiry

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

Format

Set to MQFMT_RF_HEADER.

MsgId

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

MsgType

Set to MQMT_REPLY.

Persistence

The same value as in the original command message.

Priority

The same value as in the original command message.

PutApplName

Set to the first 28 characters of the queue manager name.

PutApplType

Set to MQAT_QMGR.

Report

Set to zeroes.

Other context fields are set as if generated with MQPMO_PASS_IDENTITY_CONTEXT.

Types of error response

The broker generates three types of response:

OK response

This indicates that the command completed successfully. The response consists of a message that contains an MQRFH format header with the *CompCode* tag name in the *NameValueString* set to the value of MQCC_OK.

An OK response is sent by the broker if the command message was sent with a *MsgType* of MQMT_REQUEST, or if it was sent with a *MsgType* of MQMT_DATAGRAM and the MQRO_PAN *Report* option was set.

Warning response

This indicates that the command was only partially successful. The response consists of a message that contains an MQRFH format header with the *CompCode* tag name in the *NameValueString* set to the value of MQCC_WARNING. The *Reason* and the *ReasonText* tag names and values identify the nature of the warning.

A warning response is sent by the broker if the command message was sent with a *MsgType* of MQMT_REQUEST, or if it was sent with a *MsgType* of MQMT_DATAGRAM and either the MQRO_PAN or MQRO_NAN *Report* options were set.

Error response

This indicates that the command has failed. The response consists of a message that contains an MQRFH format header with the *CompCode* name in the *NameValueString* set to the value of MQCC_FAILED. The *Reason* and the *ReasonText* names and values identify the nature of the failure, and additional names and values can be used to give more information.

Response messages

Error responses are sent by the broker if the command message was sent with a *MsgType* of MQMT_REQUEST, or if it was sent with a *MsgType* of MQMT_DATAGRAM and the MQRO_NAN *Report* option was set.

Broker responses

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.

The standard parameters listed below are always returned in the order shown. In the case where an error is being reported, they can be followed by an optional parameter (depending on the command message that failed) that gives more information about the error.

With multiple errors, the group of standard and optional parameters are repeated as necessary.

The *NameValueString* of the command message that caused an error is usually appended to the broker response message following the MQRFH structure, to assist in diagnosis of the error. However, in the case of an MQRC_RFH_ERROR or MQRCCF_MSG_LENGTH_ERROR, the *NameValueString* of the command message that caused the error is not appended to the broker response message.

Standard parameters

CompCode

name: "MQPSCompCode" (string constant: MQPS_COMPCODE)
value: The completion code is returned in decimal form, and takes one of three values:
MQCC_OK
Command completed successfully
MQCC_WARNING
Command completed with warning
MQCC_FAILED
Command failed

Reason

name: "MQPSReason" (string constant: MQPS_REASON)
value: A decimal value corresponding to the error code. It is set to the value of MQRC_NONE if *CompCode* is set to MQCC_OK.

Error codes are listed on page 88, and in the sections describing individual command messages.

ReasonText

name: "MQPSReasonText" (string constant: MQPS_REASON_TEXT)
value: A string corresponding to the error code. It is set to MQRC_NONE if *CompCode* is set to MQCC_OK.

Error codes are listed on page 88, and in the sections describing individual command messages.

Optional parameters

Command

name: "MQPSCommand" (string constant: MQPS_COMMAND)

value: The incorrect command that was specified when a command fails with MQRC_RFH_COMMAND_ERROR.

DeleteOptions

name: "MQPSDelOpts" (string constant: MQPS_DELETE_OPTIONS)

value: The incorrect delete options that were specified when a command fails with MQRCCF_DEL_OPTIONS_ERROR.

ErrorId

name: "MQPSErrorId" (string constant: MQPS_ERROR_ID)

value: An additional reason code (decimal value) when a command fails with MQRCCF_Q_MGR_NAME_ERROR, MQRCCF_Q_NAME_ERROR or MQRCCF_NOT_AUTHORIZED. For example, the value might be MQRC_UNKNOWN_ENTITY indicating that the subscriber is not authorized because it is unknown to the broker.

ErrorPos

name: "MQPSErrorPos" (string constant: MQPS_ERROR_POS)

value: A decimal value indicating the position in the *NameValueString* of the command message sent to the broker at which an error was found. An error at the first character is reported with an error position of zero.

If the first 'MQPS' tag isn't MQPSCommand, the command fails with an MQRC_RFH_COMMAND_ERROR, and the MQPSErrorPos tag indicates the position of the offending tag.

If no 'MQPS' tags were encountered, the command fails with an MQRC_RFH_COMMAND_ERROR, and the MQPSErrorPos tag is set to the last character in the string.

If an 'MQPS' tag doesn't have a matching value, or a quoted name or value doesn't have a matching end quote, the command fails with an MQRC_RFH_STRING_ERROR, and the MQPSErrorPos tag indicates the position in the string where the error was detected.

ParameterId

name: "MQPSParmId" (string constant: MQPS_PARAMETER_ID)

value: The incorrect parameter that was specified, or the parameter that was missing, when a command fails with MQRC_RFH_PARM_ERROR, MQRC_RFH_DUPLICATE_PARM or MQRC_RFH_PARM_MISSING.

PublicationOptions

name: "MQSPubOpts" (string constant: MQPS_PUBLICATION_OPTIONS)

value: The incorrect publication options that were specified when a command fails with MQRCCF_PUB_OPTIONS_ERROR.

QMgrName

name: "MQPSQMgrName" (string constant: MQPS_Q_MGR_NAME)

value: The invalid queue manager name that was specified when a command fails with MQRCCF_Q_MGR_NAME_ERROR.

QName

name: "MQPSQName" (string constant: MQPS_Q_NAME)

value: The invalid queue name that was specified when a command fails with MQRCCF_Q_NAME_ERROR.

RegistrationOptions

name: "MQPSRegOpts" (string constant: MQPS_REGISTRATION_OPTIONS)

value: The incorrect registration options that were specified when a command fails with MQRCCF_REG_OPTIONS_ERROR.

StreamName

Broker responses

name: "MQPSStreamName" (string constant: MQPS_STREAM_NAME)
value: The unknown or incorrect stream name that was specified when a command fails with MQRCCF_UNKNOWN_STREAM or MQRCCF_STREAM_ERROR.

Topic

name: "MQPSTopic" (string constant: MQPS_TOPIC)
value: Up to 256 characters of the incorrect topic name that was specified when a command fails with MQRCCF_TOPIC_ERROR.

UserId

name: "MQPSUserId" (string constant: MQPS_USER_ID)
value: The user ID to which the publisher or subscriber is currently assigned when a command fails with MQRCCF_DUPLICATE_IDENTITY.

Examples

Here are some examples of the *NameValueString* in a **Broker Response** message. A successful response is as follows:

```
MQPSCompCode    0
MQPSReason      0
MQPSReasonText  MQRC_NONE
```

Examples of failure responses are:

```
MQPSCompCode    2
MQPSReason      2102
MQPSReasonText  MQRC_RESOURCE_PROBLEM
```

```
MQPSCompCode    2
MQPSReason      3082
MQPSReasonText  MQRCCF_REG_OPTIONS_ERROR
MQPSRegOpts     DeregAll
```

Error codes applicable to all commands

The following reason codes might be returned in the *NameValueString* of the response message for any of the commands, in addition to the codes listed for each command message. See *WebSphere MQ Messages* for detailed descriptions of these codes.

Reason	Reason text	Explanation
2334	MQRC_RFH_ERROR	MQRFH structure not valid.
2335	MQRC_RFH_STRING_ERROR	"NameValueString" field not valid.
2336	MQRC_RFH_COMMAND_ERROR	Command not valid.
2337	MQRC_RFH_PARM_ERROR	Parameter not valid.
2338	MQRC_RFH_DUPLICATE_PARM	Duplicate parameter.
2339	MQRC_RFH_PARM_MISSING	Parameter missing.
3016	MQRCCF_MSG_LENGTH_ERROR	Message length not valid.
3023	MQRCCF_MD_FORMAT_ERROR	Format not valid.
3050	MQRCCF_ENCODING_ERROR	Encoding error.
3079	MQRCCF_INCORRECT_Q	Command sent to wrong broker queue.

Problem determination

Check that you are not using WebSphere MQ facilities that are not supported by WebSphere MQ Publish/Subscribe (see "Limitations" on page 31).

Problems with brokers are reported as AMQ58xx messages, which are described in *WebSphere MQ Messages*.

Problems with the command messages sent to brokers by publisher and subscriber applications are reported in broker response messages (described in “Broker responses” on page 86). Set the *MsgType* and *Report* options in the message descriptor of the command message so that the broker sends a response message (see “The message descriptor” on page 29).

Even if there are no problems with the brokers and command messages, you might find that subscribers do not receive the publications they expect. Here is a list of possible causes:

- One or more of the brokers in the network isn’t running.
- The subscription has expired, or failed to be made in the first place.
 Use the *amqspds* sample to check that the broker has knowledge of the subscribing application’s subscription.
- If the publishing application is running at a different broker, a channel might be down.
 Check that all channels between the publishing and subscribing brokers have been started. If not, the subscriber’s publication might be sitting on a transmission queue.
- If the publishing application is running at a different broker, the subscription might not have been propagated to that broker yet.
 Even though a subscribing application has received a positive reply to its **Register Subscriber** command message, the subscription might not have propagated to the publishing broker. Check all channels between the subscribing and publishing brokers. Also check the `SYSTEM.BROKER.CONTROL.QUEUE` at each broker, because an intermediate broker might not have processed the propagated subscription yet.
 Note that brokers process publish messages in batches. This is controlled by the *PublishBatchSize* parameter (see “Broker configuration parameters” on page 102). The effect of this is that, in general, publish messages are processed more rapidly than subscriptions. If you are loading your system with a large number of new subscriptions, there might be a delay before they are propagated to all brokers in the network.
- The publishing application might not have published successfully.
 Don’t always assume that the problem is with the subscribing application. Make sure that the publishing application received a positive response message from its broker. If it is publishing using `MQMT_DATAGRAM` messages and doesn’t specify either the `MQRO_NAN` or `MQRO_PAN` report options, the broker won’t send it a reply message, even if the **Publish** command messages fails. If such a publishing application doesn’t use the `NoReg` publication option, it must set up a valid *ReplyToQ* in the message descriptor.
- The broker might be putting the subscriber’s publications to the dead-letter queue.
 There might be a problem with the subscriber’s queue. For example, it might be put-inhibited or the publications might be too large for the queue. In this case the broker, by default, puts these messages to the dead-letter queue (DLQ). Check the DLQ at the subscriber’s broker. The broker also issues message AMQ5882 if it has had to put a message to the DLQ.
- The stream might not be supported by all necessary brokers.

Problem determination

If the publication is not being published on the default stream, all brokers in the network between the publishing and subscribing brokers must support the stream you are using. Use the `amqspcd` sample to check that the stream is supported by all necessary brokers.

Chapter 9. Sample programs

Table 4 shows the techniques demonstrated by the sample programs supplied with WebSphere MQ Publish/Subscribe on AIX, HP-UX, Linux, Solaris, and Windows.

Table 4. Sample programs for AIX, HP-UX, Linux, Solaris, and Windows

Technique	C source	Executable	MQSC script
Results service	amqsresa.c	amqsres	-
Match simulator	amqsgama.c	amqsgam	-
Administration application	amqspstda.c	amqspstd	-
Routing exit	amqspstda.c	-	-
Create definitions for sample application	-	-	amqsresa.tst
Create stream on another broker	-	-	amqsgama.tst
Create administration app. reply queue	-	-	amqspstda.tst
Create SYSTEM.BROKER.MODEL.STREAM	-	-	amqsfmda.tst

Table 5 shows the techniques demonstrated by the sample programs supplied with WebSphere MQ Publish/Subscribe on iSeries.

Table 5. Sample programs for iSeries

Technique	C source	Executable	MQSC script
Results service	AMQSRESA	AMQSRESA	-
Match simulator	AMQSGAMA	AMQSGAMA	-
Administration application	AMQSPSDA	AMQSPSDA	-
Routing exit	AMQSPSRA	-	-
Create definitions for sample application	-	-	AMQSRESA
Create stream on another broker	-	-	AMQSGAMA
Create administration app. reply queue	-	-	AMQSPSDA
Create SYSTEM.BROKER.MODEL.STREAM	-	-	AMQSFMDA

You can find the samples in the following directories (libraries for iSeries).

AIX

source files and MQSC scripts

/usr/mqm/samp/pubsub

amqspstda.*

/usr/mqm/samp/pubsub/admin

executables

/usr/mqm/samp/bin

HP-UX, Linux, and Solaris

source files and MQSC scripts

/opt/mqm/samp/pubsub

amqspstda.*

/opt/mqm/samp/pubsub/admin

Sample programs

executables

/opt/mqm/samp/bin

iSeries

source files

QMMSAMP\QCSRC

MQSC scripts

QMMSAMP\QMMS

executables

QMMS

Windows

source files and MQSC scripts

<drive:directory>\WebSphere MQ\TOOLS\C\SAMPLES\SUBSUB

amqspda.*

<drive:directory>\WebSphere MQ\TOOLS\C\SAMPLES\SUBSUB\admin

executables

<drive:directory>\WebSphere MQ\TOOLS\C\SAMPLES\BIN

The sample programs are described in the following sections:

- amqsr and amqsgm in “Sample application”
- amqspda in “Sample program for administration information” on page 157
- amqspsr in “Sample routing exit” on page 137

You must start the queue manager before running the MQSC scripts. In addition, before running the executables, you must start the broker (see Chapter 11, “Controlling the broker,” on page 107).

Instructions for compiling the samples can be found in the *WebSphere MQ Application Programming Guide*.

Sample application

The following aspects of the results service application are described in “Sample application” on page 15:

- The use of streams other than the default stream.
- Event publications (not retained).
- State publications (retained).
- Wildcard matching of topic strings.
- Multiple publishers on the same topics (event publications only).
- The need to subscribe to a topic *before* it is published on (event 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.

The application’s use of multiple subscription identities on the same subscriber queue is covered in “Publisher and subscriber identity” on page 27, and the following aspects are described in Chapter 6, “Format of command messages,” on page 47:

- MQRFH format messages.
- MQRFH *NameValueString* parsing.
- MQRFH broker response message checking.

- **Publish, Register Subscriber, Request Update, Delete Publication and Deregister Subscriber** command messages.
- Separate user data in **Publish** messages.

Running the application

To run the application on a single queue manager, first start the queue manager and then enter the following command:

```
runmqsc QMgrName < amqsresa.tst
```

where QMgrName is the queue manager that the results service will use (if QMgrName is omitted, the default queue manager will be assumed). This creates the appropriate queues on the queue manager. Then start the broker (see Chapter 11, “Controlling the broker,” on page 107).

The results service program is started by entering the following:

```
amqsres QMgrName
```

QMgrName is optional, and defaults to the default queue manager. The results service produces the following output:

```
Results Service is ready for match input,
instances of amqsgam can now be started.
```

You can now start one or more match simulators by entering the following command:

```
amqsgam Team1 Team2 QMgrName
```

QMgrName is optional, as before.

Typical output from a match simulator is:

```
Match between Team1 and Team2
GOAL! Team2 scores after 20 minutes
GOAL! Team1 scores after 25 minutes
GOAL! Team1 scores after 38 minutes
GOAL! Team2 scores after 73 minutes
Full time
```

This would produce corresponding output from the results service, for example:

```
LATEST: Team1 0, Team2 0
LATEST: Team1 0, Team2 1
LATEST: Team1 1, Team2 1
LATEST: Team1 2, Team2 1
LATEST: Team1 2, Team2 2
FULLTIME: Team1 2, Team2 2
```

A match simulator can be run on a different queue manager in the broker hierarchy if required. In this case, you need to enter the following command to create the appropriate stream queue on that queue manager:

```
runmqsc QMgrName < amqsgama.tst
```

You must do this *before* starting the results service and the match simulator.

The team names must be 31 characters or less in length, and contain no blanks. The simulator runs for 30 seconds and scores goals at random for each side.

The simulator publishes event publications on the following topics:

Sample application

```
Sport/Soccer/Event/MatchStarted  
Sport/Soccer/Event/ScoreUpdate  
Sport/Soccer/Event/MatchEnded
```

The *UserData* is contained in a formatted string following the *NameValueString* of the MQRFH header. In the case of 'MatchStarted' or 'MatchEnded' it consists of both team names in the following structure:

```
{  
    MQCHAR32  Team1;  
    MQCHAR32  Team2;  
}
```

For a 'ScoreUpdate' the *UserData* consists of the name of the team that scored, for example:

```
MQCHAR32  TeamThatScored;
```

The team names are NULL padded to 32 characters.

The results service program subscribes to these three topics to monitor the state of play in the matches that are active. It publishes the latest score in the match between Team1 and Team2 on the following topic:

```
Sport/Soccer/State/LatestScore/Team1 Team2
```

In this case the *UserData* is a variable string containing the data in the format:
"Team1Score Team2Score"

For example "0 0" or "2 1".

Figure 20 on page 95 illustrates the situation when four match simulators are running.

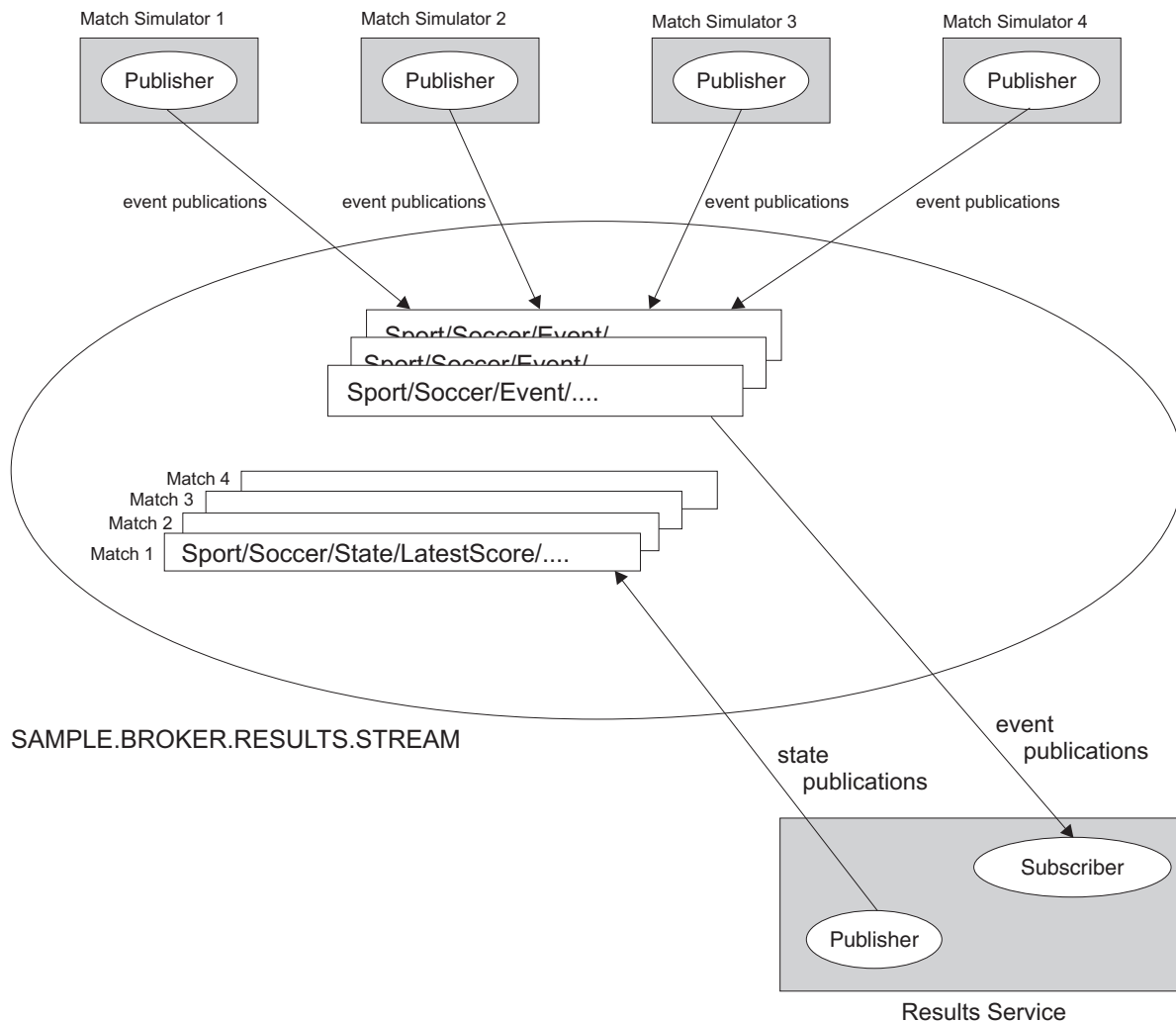


Figure 20. Results service running with four match simulators. The match simulators send event publications to three topics (MatchStarted, ScoreUpdate, MatchEnded). The results service subscribes to these, and sends state publications to four state topics (the LatestScore for each match).

When a match has ended, the retained publication that contains its latest score is deleted. After a period of inactivity (45 seconds), the results service deregisters the subscription from the `Sport/Soccer/Event/*` topic and the program ends with the message:

Results Service has ended

If the results service program (`amqsres`) is stopped and restarted while the match simulators are still running, the results are restored to their correct values and processing continues as before.

Possible extensions

The sample application illustrates many aspects of a WebSphere MQ Publish/Subscribe system. Possible extensions that might be implemented by the user include:

- Distribute the results service and match simulators across multiple connected brokers.
- Extend the application to handle more than one sport, and have a results service running for each sport.

Sample application

- Extend the results service to publish the final score when a match ends, and add another application that subscribes to these publications to produce a table of results.
- Extend the match simulator to confirm that a results service is subscribing to Sport/Soccer/Events/* topics before it starts publishing. This can be done using metatopics.
- Change the format of the user data in the publications, create a user defined format, and write a data conversion exit to enable the passing of publications between different platforms or languages.

Application Messaging Interface samples

For sample publisher and subscriber programs that use the Application Messaging Interface in C, C++, and Java, see the *WebSphere MQ Application Messaging Interface* book.

Part 3. Managing the broker

Chapter 10. Setting up a broker 99

Broker queues	99
System queues	99
Other stream queues	100
SYSTEM.BROKER.MODEL.STREAM	100
Internal queues	101
Dead-letter queue	101
Other considerations	101
Access control	101
Backup	101
Broker configuration stanza	102
Broker configuration parameters	102

Compiling a publish/subscribe routing exit program	137
Sample routing exit	137

Chapter 11. Controlling the broker 107

Starting a broker	107
Using triggering to start the broker	107
Stopping a broker	107
Displaying the status of a broker	107
Adding a stream	107
Creating a stream queue.	107
Informing other brokers about the stream	108
Deleting a stream	108
Deleting a stream on an isolated broker	108
Deleting a stream on a broker that is part of a network	108
Adding a broker to a network.	109
Deleting a broker from the network	109
Problems when deleting brokers	110
Deleting a broker that has a child broker	110
Sequence of commands for adding and deleting brokers	110

Chapter 12. Control commands 113

clrmqbrk (Clear broker's memory of a neighboring target broker)	114
dlmqbrk (Delete broker)	117
dspmqrk (Display broker status)	119
endmqbrk (End broker function)	121
migmqrk (Migrate broker to WebSphere Business Integration Brokers)	123
strmqbrk (Start broker function)	125

Chapter 13. Message broker exit 129

Publish/subscribe routing exit.	129
Parameters	129
Usage notes	129
C invocation.	130
Publish/subscribe routing exit parameter structure	130
Fields	131
C declaration	136
Writing a publish/subscribe routing exit program	136
Limitations on WebSphere MQ work done in the routing exit.	136
Security considerations	137

Chapter 10. Setting up a broker

Publishers, subscribers, and brokers communicate by using queues. Configuration and monitoring of these queues can be performed by whatever technique is currently in use for WebSphere MQ, whether supplied by WebSphere MQ or available from third parties.

Before you can use WebSphere MQ Publish/Subscribe you need to do the following things to set up your broker:

1. If necessary, define the queues that the broker needs to use.
2. Authorize applications to use these queues.
3. Review the default settings of the broker parameters in the queue manager initialization file (qm.ini) or review them using WebSphere MQ Explorer.

For information about managing your brokers when they have been set up, see Chapter 11, “Controlling the broker,” on page 107.

How to find out about publishers and subscribers registered with brokers, and how to write applications to manage a network of brokers is explained in Part 4, “System programming,” on page 139.

Broker queues

Brokers are event-driven; they wait for messages to arrive on their queues. The broker needs several system queues, and can also have any number of *stream* queues; these are described below.

System queues

The broker uses three system queues. These queues all have names beginning with SYSTEM.BROKER, and are used for the purposes described below. These queues are created automatically when the broker starts if they do not already exist. You might want to alter access authority to these queues.

SYSTEM.BROKER.CONTROL.QUEUE

This is the broker’s control queue. Publisher and subscriber applications, and other brokers, send all command messages (except publications and requests to delete publications) to this queue.

SYSTEM.BROKER.CONTROL.QUEUE is created as a predefined queue based on the SYSTEM.DEFAULT.LOCAL.QUEUE.

SYSTEM.BROKER.DEFAULT.STREAM

This is the queue that receives all publication messages for the default stream. Applications can also send requests to delete publications on the default stream to this queue.

SYSTEM.BROKER.DEFAULT.STREAM is created using SYSTEM.BROKER.MODEL.STREAM if it exists, otherwise the broker predefines it based on the SYSTEM.DEFAULT.LOCAL.QUEUE.

Note: SYSTEM.BROKER.DEFAULT.STREAM is created with a default persistence of yes. This means that an application using the

MQPER_AS_Q_DEF option in the message descriptor (the default) publishes persistent messages by default.

SYSTEM.BROKER.ADMIN.STREAM

This is the queue that the broker uses to publish its own broker configuration information (for example the identity of its parent). If you write your own administration applications, they can use the information published on this stream. You can also publish information on this stream (but not to topics with names beginning MQ/).

SYSTEM.BROKER.ADMIN.STREAM is created using SYSTEM.BROKER.MODEL.STREAM if it exists, otherwise the broker predefines it based on the SYSTEM.DEFAULT.LOCAL.QUEUE.

Other stream queues

Stream queues are used to process publications for all topics within a stream. Applications send publications (and requests to delete publications) to a stream queue. The stream queue must be a local queue at the broker, not an alias or remote queue. Applications can send messages to a stream queue through an alias or remote queue.

Publishing applications can register with the broker before they start sending publications. If the application specifies that it will be using a stream queue that does not yet exist, the broker might create a permanent dynamic queue with the same name as the stream specified, based on the SYSTEM.BROKER.MODEL.STREAM queue.

If the SYSTEM.BROKER.MODEL.STREAM queue does not exist, any message sent by an application that refers to a stream for which there is no stream queue, is rejected. The broker keeps information about which streams are known to it so that, when it is restarted, it can recognize the stream queues.

Applications can also specify the stream name in a publication message. If a publication message specifies the name of a stream that is different from the name of the queue to which it was sent, the message is rejected. If the application does not specify a stream name, it defaults to the name of the stream queue to which it is sent.

If you are using a network of brokers, and you want to restrict a certain stream to a particular sub-tree of the hierarchy, brokers immediately outside the sub-tree must not have a SYSTEM.MODEL.STREAM.QUEUE defined. All stream queues for streams that these brokers support must, therefore, be defined by the administrator.

SYSTEM.BROKER.MODEL.STREAM

The SYSTEM.BROKER.MODEL.STREAM is a model queue definition that can be used by the broker to create dynamic queues to receive publications for streams other than the default stream. This is used only if the stream queue does not already exist. This definition must specify that the dynamic queue to be created is a permanent-dynamic queue. If this queue does not exist, all stream queues must be defined by the administrator. (The administrator can also define stream queues manually, even if this queue does exist.)

This queue is supplied as sample `amqsfmda.tst` (see page 91). To create the queue from the sample, use the following command:

```
runmqsc QMgrName < amqsfmda.tst
```

where QMgrName is the name of the queue manager.

Internal queues

The broker creates several other queues for its own internal use. These queues also have names beginning with SYSTEM.BROKER. The broker uses them to store its persistent state, such as subscriptions and retained publications.

Dead-letter queue

You are recommended to set up a dead-letter queue for each queue manager that has a broker running on it. This enables the broker to continue operating when problems are encountered, such as a subscriber's queue being full. In this case publications for that subscriber are put to the dead-letter queue, and the broker continues to process publish command messages.

Without a dead-letter queue you might also have problems if you want to delete that broker from the network (see "Deleting a broker from the network" on page 109).

Other considerations

Some things you should consider when setting up a broker are:

- Access control
- Backup

Access control

Normal WebSphere MQ access control techniques apply to applications and brokers opening queues for Publish/Subscribe messages. These authorization checks are carried out using standard WebSphere MQ functions. The authority is tested before any message is sent to a particular identity after a broker restart, but not necessarily each subsequent time a message is put (see "Streams" on page 10).

Any application putting a message to the broker's SYSTEM.BROKER.CONTROL.QUEUE must have authorization to put messages to this queue.

A publisher must be authorized to put messages on the broker's appropriate stream queue.

Subscribers must be authorized to browse the broker's stream queue; this is checked by the broker because the subscriber does not try to open the broker's stream queue. In addition, a subscriber must have authority to put messages on the subscriber queue that the publications will be sent to.

There is no topic based security; the access check is for the stream and there are no further checks on topics within a particular stream.

Backup

Normal WebSphere MQ backup and restore procedures apply, as described in the *WebSphere MQ System Administration Guide*. When a queue manager is backed up, a broker installed on that queue manager is backed up as well.

Broker configuration stanza

On UNIX systems, broker parameters are controlled by the Broker stanza of the queue manager configuration file, `qm.ini`. Figure 21 shows an example of this stanza. On Windows, you can view and update these settings using the Broker page of the queue manager properties in WebSphere MQ Explorer.

1. Right-click the queue manager and select **Properties**.
2. Select **Broker** in the left-hand pane of the dialog. The Broker properties are displayed in the right-hand pane.
3. If you make any changes to the values, click **Apply** then **OK**.

The parameters are described in “Broker configuration parameters.”

```
Broker:
  MaxMsgRetryCount=5
  StreamsPerProcess=1
  OpenCacheSize=128
  OpenCacheExpiry=300
  PublishBatchSize=5
  PublishBatchInterval=0
  ChkPtMsgSize=100000
  ChkPtActiveCount=400
  ChkPtRestartCount=40
  RoutingExitPath=/opt/mqm/samp/bin/amqspsra(RoutingExit)
  RoutingExitConnectType=STANDARD
  RoutingExitAuthorityCheck=no
  RoutingExitData=My routing exit string data
  SyncPointIfPersistent=no
  DiscardNonPersistentInputMsg=no
  DLQNonPersistentResponse=yes
  DiscardNonPersistentResponse=no
  DLQNonPersistentPublication=yes
  DiscardNonPersistentPublication=no
  GroupId=nobody
  JmsStreamPrefix=JMS
```

Figure 21. Sample Broker stanza for `qm.ini`

Note: You do not need to list parameters if you are using their default values. Any parameters that you do list are checked for validity. A blank entry is not valid.

Broker configuration parameters

MaxMsgRetryCount=number

When the broker fails to process a command message under syncpoint (for example a publish message that cannot be delivered to a subscriber because the subscriber queue is full and it is not possible to put the publication to the dead-letter queue), the unit of work is backed out and the command retried this number of times before the broker attempts to process the command message according to its report options instead.

The default is `MaxMsgRetryCount=5`.

StreamsPerProcess=number

The broker consists of a broker main process (`runmqbrk`) and a number of broker worker processes (`amqfcxba`). Each worker process is capable of supporting one or more streams. Depending upon the broker configuration (for example the operating system, number of streams, number of publishers,

subscribers and retained messages, whether a non-fastpath routing exit is in use), varying this number can alter capacity or throughput (or both).

If you have a large number of lightly loaded streams, increase this value.

The defaults are:

AIX

StreamsPerProcess=10 (*RoutingExitConnectType*=Standard)
StreamsPerProcess=1 (*RoutingExitConnectType*=Fastpath)
StreamsPerProcess=1 (no routing exit)

HP-UX, Linux, and Solaris

StreamsPerProcess=10

Windows

StreamsPerProcess=10

OpenCacheSize=number

Each broker stream thread (2 threads for each stream) keeps a cache of recently used open queues. This parameter specifies the maximum number of queues in the cache.

The default is *OpenCacheSize*=128.

OpenCacheExpiry=number

Each broker stream thread (2 threads for each stream) keeps a cache of recently used open queues. If a queue in the cache is not used for approximately *OpenCacheExpiry* seconds, the queue is removed from the cache (closed).

The default is *OpenCacheExpiry*=300.

PublishBatchSize=number

The broker normally processes publish messages within syncpoint. It can be inefficient to commit each publication individually, and in some circumstances the broker can process multiple publish messages in a single unit of work. This parameter specifies the maximum number of publish messages that can be processed in a single unit of work.

The default is *PublishBatchSize*=5.

PublishBatchInterval=number

The broker normally processes publish messages within syncpoint. It can be inefficient to commit each publication individually, and in some circumstances the broker can process multiple publish messages in a single unit of work. This parameter specifies the maximum time (in milliseconds) between the first message in a batch and any subsequent publication included in the same batch. A batch interval of 0 indicates that up to *PublishBatchSize* messages can be processed, provided that the messages are available immediately.

The default is *PublishBatchInterval*=0.

ChkPtMsgSize=number

The broker stores individual publisher and subscriber registrations as messages on its internal queues. Periodically, it might consolidate a number of these registrations into a smaller number of larger messages called checkpoint messages. This action is called checkpointing and is performed to reduce the number of messages that need to be read to restore the publisher and subscriber registrations at broker and stream restart.

The *ChkPtMsgSize* parameter determines the default size of each checkpoint message in bytes, which in turn determines the number of registrations that each checkpoint message can contain.

Broker configuration

The default is *ChkPtMsgSize*=100000.

ChkPtActiveCount=*number*

The broker stores individual publisher and subscriber registrations as messages on its internal queues. Periodically it might consolidate a number of these registrations into a smaller number of larger messages called checkpoint messages. This action is called checkpointing and is performed to reduce the number of messages that need to be read to restore the publisher and subscriber registrations at broker and stream restart.

The number of changes that need to be made to part of the registration state of an individual stream during normal broker operation before checkpointing is considered for that part depends on the *ChkPtActiveCount* parameter.

The default is *ChkPtActiveCount*=400. A lower value makes checkpointing occur more frequently. A higher value makes checkpointing occur less frequently. A value of 0 disables checkpointing completely during normal operation and would be applicable if checkpoint activity was having an adverse effect on broker throughput.

ChkPtRestartCount=*number*

The broker stores individual publisher and subscriber registrations as messages on its internal queues. Periodically it might consolidate a number of these registrations into a smaller number of larger messages called checkpoint messages. This action is called checkpointing and is performed to reduce the number of messages that need to be read to restore the publisher and subscriber registrations at broker and stream restart.

The number of changes that need to have been made to part of the registration state of an individual stream during broker or stream restart before checkpointing is considered for that part depends on the *ChkPtRestartCount* parameter.

The default is *ChkPtRestartCount*=40. This is lower than the *ChkPtActiveCount* on the assumption that stream or broker restart is a more suitable time for the registration state to be checkpointed. A value of 0 disables checkpointing completely during restart.

RoutingExitPath=[*path*]*module_name*(*function_name*)

Before the broker sends a publication to a subscriber, the broker invokes the exit identified by the *RoutingExitPath* (if any).

The default is no routing exit.

RoutingExitConnectType=*FASTPATH*|*STANDARD*

If the broker is configured to use a routing exit, the exit runs within a broker process. If the exit conforms to the requirements of a fastpath application (MQCNO_FASTPATH_BINDING), the broker process can use a fastpath connection to the queue manager. This attribute informs the broker if the exit meets the standards necessary for a fastpath application.

Note: This attribute is relevant only if a *RoutingExitPath* is specified. For performance reasons *RoutingExitConnectType*=FASTPATH is desirable.

The default is *RoutingExitConnectType*=STANDARD.

RoutingExitAuthorityCheck=*yes*|*no*

Before the broker sends a publication to a subscriber the broker must have validated the subscribers authority to write to the subscriber queue. If the routing exit changes the message destination, the authority check already

performed by the broker is not valid. This attribute informs the broker if the authority check should be repeated for any changed destination.

Note: The performance implications of setting *RoutingExitAuthorityCheck=yes* are considerable if the routing exit frequently changes the destination.

The default is *RoutingExitAuthorityCheck=no*.

RoutingExitData=string

If the broker is using a routing exit, the broker invokes the routing exit passing an MQPXP structure as input. The data specified using this attribute is provided in the *ExitData* field. The string can be up to MQ_EXIT_DATA_LENGTH characters in length.

The default is 32 blank characters.

SyncPointIfPersistent=yes | no

If this attribute is specified, when the broker reads a publish or delete publication message from a stream queue during normal operation the broker specifies MQGMO_SYNCPOINT_IF_PERSISTENT. This makes the broker receive nonpersistent messages outside syncpoint. If the broker receives a publication outside syncpoint, the broker forwards that publication to subscribers known to the broker outside syncpoint.

When using *SyncPointIfPersistent=yes* it is possible that a nonpersistent publication might not be delivered to all subscribers known to a broker (for example, if an immediate broker shutdown occurred while a publish message was being processed). If *SyncPointIfPersistent=yes* is specified, the broker performance for publishing nonpersistent publications improves.

The default is *SyncPointIfPersistent=yes*.

DiscardNonPersistentInputMsg=yes | no

If the broker cannot process a nonpersistent input message, the broker might attempt to write the input message to the dead-letter queue (depending on the report options of the input message). If the attempt to write the input message to the dead-letter queue fails, and the MQRO_DISCARD_MSG report option was specified on the input message or *DiscardNonPersistentInputMsg=yes*, the broker discards the input message. If *DiscardNonPersistentInputMsg=no* is specified, the broker will only discard the input message if the MQRO_DISCARD_MSG report option was set in the input message.

The defaults are:

DiscardNonPersistentInputMsg=no if *SyncPointIfPersistent=no*.

DiscardNonPersistentInputMsg=yes if *SyncPointIfPersistent=yes*.

Note: If *SyncPointIfPersistent=yes* is set, *DiscardNonPersistentInputMsg=no* must not be set.

DLQNonPersistentResponse=yes | no

If the broker attempts to generate a response message in response to a nonpersistent input message, and the response message cannot be delivered to the reply-to queue, this attribute indicates whether the broker should write the undeliverable response message to the dead-letter queue.

The default is *DLQNonPersistentResponse=yes*.

DiscardNonPersistentResponse=yes | no

If the broker attempts to generate a response message in response to a nonpersistent input message, and the response message cannot be delivered to

Broker configuration

the reply-to queue or written to the dead-letter queue, this attribute indicates whether the broker can discard the undeliverable response message.

The default is:

DiscardNonPersistentResponse=no if *SyncPointIfPersistent=no*.

DiscardNonPersistentResponse=yes if *SyncPointIfPersistent=yes*.

Note: If *SyncPointIfPersistent=yes* is set, *DiscardNonPersistentResponse=no* must not be set.

DLQNonPersistentPublication=yes | no

If the broker fails to send a nonpersistent publication to a subscriber, this attribute indicates whether the broker should put the publication to the dead-letter queue.

The default is *DLQNonPersistentPublication=yes*.

DiscardNonPersistentPublication=yes | no

If the broker fails to send a nonpersistent publication to a subscriber and cannot write the publication to the dead-letter queue, this attribute indicates whether the broker can discard the publication.

The default is:

DiscardNonPersistentPublication=no if *SyncPointIfPersistent=no*.

DiscardNonPersistentPublication=yes if *SyncPointIfPersistent=yes*.

Note: If *SyncPointIfPersistent=yes* is set, *DiscardNonPersistentPublication=no* must not be set.

GroupId=group_identifier

Specifies the group that owns the stream queues created by the broker, except the admin stream (for example, SYSTEM.BROKER.DEFAULT.STREAM). Users in this group can access the stream queues. If this group does not exist, the broker cannot run.

If not specified, the following defaults are used (this normally means that all users can access the stream queues):

AIX, Linux, and Solaris

GroupId=nobody

HP-UX

GroupId=nogroup

iSeries

*GroupId=*PUBLIC*

Windows

GroupId=Users

Note: For WebSphere MQ for Windows, the *GroupId* is set to 'Users' or the national language equivalent.

JmsStreamPrefix=JMS

Identifies stream names. For example, stream names beginning with JMS are restricted to JMS semantics.

The default is *JMS*.

You can use JMS characteristics to improve your broker performance.

Chapter 11. Controlling the broker

This chapter describes the following broker operations:

- “Starting a broker”
- “Stopping a broker”
- “Displaying the status of a broker”
- “Adding a stream”
- “Deleting a stream” on page 108
- “Adding a broker to a network” on page 109
- “Deleting a broker from the network” on page 109

Part 4, “System programming,” on page 139 explains how to find out about publishers and subscribers registered with brokers, and how to write applications to manage a network of brokers.

Starting a broker

Use the **strmqbrk** command (**STRMQMBRK** on iSeries) to start a broker. This command starts the broker on the specified queue manager, either initially, or as a restart after an **endmqbrk** command (**ENDMQMBRK** on iSeries). This command is described in “strmqbrk (Start broker function)” on page 125.

Using triggering to start the broker

You can start a broker by enabling triggering on any of the broker’s queues. Specify triggering on the first message. However, if a broker is triggered on more than one of its stream queues, a trigger message is generated for each queue at startup.

Stopping a broker

Use the **endmqbrk** command (**ENDMQMBRK** on iSeries) to stop a broker. This command stops the broker on the specified queue manager and is described in “endmqbrk (End broker function)” on page 121.

Displaying the status of a broker

Use the **dspmqbrk** command (**DSPMQMBRK** on iSeries) to display the status of the broker for the specified queue manager. This command is described in “dspmqbrk (Display broker status)” on page 119.

Adding a stream

The following things need to happen for a stream to be created:

- A queue must be created to hold publications for that stream.
- Information about the stream has to be passed to other brokers in the network that need to support the stream.

Creating a stream queue

The stream queue has the same name as the stream, and is usually created by the operator. There should be one instance of the stream queue at each broker that supports the stream.

Adding a stream

Alternatively, you can let the broker create the stream queue dynamically when it is needed. The queue is based on the model queue definition `SYSTEM.BROKER.MODEL.STREAM` if this is available. If the model queue definition is not available, the broker will not create stream queues dynamically.

Note: If the queue is created dynamically, the operator must grant the required access authority to applications using the queue, so use dynamic stream queue creation only in a test environment.

Informing other brokers about the stream

When a stream is first referenced by a publisher or subscriber (for example, when a registration request is sent to the broker's control queue) the broker informs its neighbors that the stream exists. If the neighboring brokers also have a queue defined for the stream (or can create one using `SYSTEM.BROKER.MODEL.STREAM`), they also recognize the stream and pass information about it to their neighbors.

If a broker that is told about the stream does not have a queue for the stream and does not have the `SYSTEM.BROKER.MODEL.STREAM`, it does not pass information about the stream to its neighbors.

Deleting a stream

Before you delete a stream, quiesce all applications that use the stream.

To delete a stream, you need to delete the stream queue. To delete the queue, you must ensure that no applications (or channels) have the queue open. If there are messages on the queue, you must remove them from the queue, or purge them when you delete the queue.

You must also ensure that you do not have a definition of the `SYSTEM.BROKER.MODEL.STREAM` on the broker. If you do, and the old one is deleted, a new version of the stream queue is created dynamically when the broker is restarted.

Deleting a stream on an isolated broker

To delete a stream on a broker that is not part of a broker network:

1. Stop the broker using `endmqbrk` (`ENDMQMBRK` on iSeries).
2. Delete the queue.
3. Restart the broker using `strmqbrk` (`STRMQMBRK` on iSeries).

When the broker realizes that the queue no longer exists, it deregisters all subscriptions to the stream, and publishes a message to the `SYSTEM.BROKER.ADMIN.STREAM` advertising that the stream has been deleted. (For information about the format of this message see "Format of broker administration messages" on page 141.)

Deleting a stream on a broker that is part of a network

A stream on a broker that is part of a broker network is deleted in the same way as for an isolated broker. Other brokers in the network are advised that the stream has been deleted and stop sending publications and subscription requests to the broker for that stream. Messages sent from other brokers before they receive notification that the stream has been deleted are handled as follows:

- Publication messages are put to the dead-letter queue.

- Registration messages are put to the dead-letter queue.

Adding a broker to a network

You are recommended to define the broker topology from the root down.

Before you can add a broker to the network, channels in both directions must exist between the queue manager that hosts the new broker and the queue manager that hosts the parent. Brokers use explicit addressing when sending messages to queues that reside on another queue manager. When the queue is opened by the broker, both the queue and queue manager names are specified. To facilitate multi-broker operation, this queue manager name must resolve to the appropriate transmission queue. The simplest method of achieving this is for the transmission queue to have the same name as the remote queue manager name.

If you do not adopt this naming scheme, queue manager alias definition can be used to ensure that messages get placed on the appropriate transmission queue. For example, to specify that messages sent to queue manager PARENT are placed on transmission queue, PARENT.XMITQ:

```
DEFINE QREMOTE (PARENT) RNAME() RQMNAME(PARENT) XMITQ(PARENT.XMITQ)
```

To specify that messages sent to queue manager PARENT are placed on transmission queue, PARENT.XMITQ on iSeries:

```
CRTMQMQ QNAME(PARENT) QTYPE(*RMT) RMTMQMNAME(PARENT) TMQMNAME(PARENT.XMITQ)
```

To add a broker to the network, start the broker with the **strmqbrk** command (**STRMQMBRK** on iSeries), specifying the name of the parent broker if appropriate. When the broker has been started with a parent named you cannot change the name of its parent, even when the broker is restarted. You cannot change the parent of a broker as part of normal operational procedures without disrupting service. This command is described in “strmqbrk (Start broker function)” on page 125.

Deleting a broker from the network

Brokers must always be deleted from the bottom of the broker hierarchy. You cannot delete a broker if it has one or more child brokers. (See “Sequence of commands for adding and deleting brokers” on page 110 for more information.)

The broker needs to delete any queues that were created by the broker, so these queues need to be closed and empty.

1. Stop the broker using **endmqbrk** (**ENDMQMBRK** on iSeries).
2. Quiesce all applications that use the broker.
3. Applications and brokers can use channels to talk to the broker, so receiving channels might have queues open. If a channel has a queue open, stop and restart the channel.
4. Use the **dlmqbrk** command (**DLTMQMBRK** on iSeries) to delete the broker. This command is described in “dlmqbrk (Delete broker)” on page 117.

The broker performs the actions listed in “dlmqbrk (Delete broker)” on page 117 and sends a message to tell its parent broker that it is no longer active. **This message needs to be processed by its parent broker before the parent can be deleted.** The parent broker can process this message only while running.

Deleting a broker from a network

If you do not quiesce all your applications before deleting the broker, messages might be sent from other brokers before they receive notification that the broker has been deleted. Because there is no broker to handle these messages, the queue manager deals with them according to the report options set for these messages. This means that publication and registration messages are put to the dead-letter queue. Therefore, **ensure that a dead-letter queue has been set up for this queue manager before attempting to delete a broker.**

Problems when deleting brokers

If you cannot delete your broker, consider the following:

- Are any queues that are to be quiesced by the broker open to an application or a channel?

If so, you will receive an error message containing reason code 5840. The error log contains information about which queues cannot be quiesced.

- Does the broker have any children?

If it does, you will receive an error message containing reason code 5838. The error log contains information about the broker's children.

Deleting a broker that has a child broker

If you cannot delete a child of a broker you want to delete (for example, because the queue manager of the child broker has been deleted) you can use the **clrmqbrk** command to clear the broker's memory of the child broker. **Use this command in exceptional circumstances only, and do so with great care.** If it is not used correctly, brokers will see an inconsistent view of the hierarchy; this is likely to cause severe disruption to the service.

The command makes it appear as if the child broker has been deleted so that the parent broker can be deleted. If you use this command, you **must** remember to make sure that both ends have the same view of the relationship (see page 114).

Sequence of commands for adding and deleting brokers

This example shows the sequence of commands for adding and deleting brokers in a network. Queue manager A is to host the parent broker and queue manager B is to host the child broker. Channels are defined between the two queue managers. Broker A is the parent broker, so this must be created first. Broker B is then created as a child broker of broker A. The sequence of commands to achieve this is as follows:

```
START CHANNEL (B.to.A)
START CHANNEL (A.to.B)
strmqbrk -m A
strmqbrk -m B -p A
```

Use the following sequence for iSeries:

```
STRMQMCHL CHLNAME(B.to.A)
STRMQMCHL CHLNAME(A.to.B)
STRMQMBRK MQMNAME(A)
STRMQMBRK MQMNAME(B) PARENTMQM(A)
```

When both brokers are deleted, broker B must be deleted first, and broker A must be available for this to happen. Only when broker B has been deleted can broker A be deleted. The sequence of commands to achieve this is as follows.


```
endmqbrk -m B  
STOP CHANNEL (A.to.B)  
START CHANNEL (A.to.B)  
dlmqbrk -m B
```

```
endmqbrk -m A  
STOP CHANNEL (B.to.A)  
START CHANNEL (B.to.A)  
dlmqbrk -m A
```

Use the following sequence for iSeries:

```
ENDMQMBRK MQMNAME(B)  
ENDMQMCHL CHLNAME(A.to.B)  
STRMQMCHL CHLNAME(A.to.B)  
DLTMQMBRK MQMNAME(B)
```

```
ENDMQMBRK MQMNAME(A)  
ENDMQMCHL CHLNAME(B.to.A)  
STRMQMCHL CHLNAME(B.to.A)  
DLTMQMBRK MQMNAME(A)
```

Chapter 12. Control commands

This chapter describes the commands that you can use to manage your brokers. Chapter 11, "Controlling the broker," on page 107 discusses the circumstances under which you would use these commands. The commands are:

- "clrmqbrk (Clear broker's memory of a neighboring target broker)" on page 114
- "dlmqbrk (Delete broker)" on page 117
- "dspmqbrk (Display broker status)" on page 119
- "endmqbrk (End broker function)" on page 121
- "migmqbrk (Migrate broker to WebSphere Business Integration Brokers)" on page 123
- "strmqbrk (Start broker function)" on page 125

You can now use the following commands on iSeries and issue them using the command line. See chapter 2 "Managing WebSphere MQ for iSeries using CL commands" in the *WebSphere MQ for iSeries V6 System Administration Guide* for further information about using the iSeries command line.

- **CLRMQMBRK** - "clrmqbrk (Clear broker's memory of a neighboring target broker)" on page 114
- **DLTMQMBRK** - "dlmqbrk (Delete broker)" on page 117
- **DSPMQMBRK** - "dspmqbrk (Display broker status)" on page 119
- **ENDMQMBRK** - "endmqbrk (End broker function)" on page 121
- **STRMQMBRK** - "strmqbrk (Start broker function)" on page 125

clrmqbrk (Clear broker's memory of a neighboring target broker)

Purpose

Use the **clrmqbrk** command to clear the broker's memory of a neighboring (parent or child) target broker. On iSeries, the command name is **CLRMQMBRK**. **Use this command in exceptional circumstances only.**

The broker cancels all subscriptions from the target broker. The broker must be stopped when this command is issued. The command is synchronous, and when it has completed the broker can be restarted normally. No messages are read from any of the input queues.

After restart, the broker detects any messages on its input queues that came from this broker, and processes them according to their report options.

You need to clear the memory of the broker at each end of the connection. This means that you must issue this command (or an equivalent) to both the parent and the child broker. If you do not clear the memory of the broker at each end of the connection, one broker continues to send messages to the other broker, which are processed according to their report options. This might lead to a build-up of messages on the dead-letter queue, and unnecessary report messages being sent across the network.

Deleting a broker with **dltmqbrk** requires first deleting its children. If this is impractical (for example, if the child broker is no longer reachable) the **clrmqbrk** command can be used to make the child broker appear deleted to its parent so that the parent can be deleted. The child broker must be deleted whenever practical.

You can also use this command at the child with the **-p** parameter to break the link with its parent. Using such a pair of **clrmqbrk** commands, one at the child and one at the parent, causes the child and its descendants (if any), together with their publishers and subscribers, to be isolated from the rest of the network. The child now becomes the root node of a hierarchy. It can operate this way or be restarted with another parent (or even with its old parent) provided the new parent is not also a descendant.

Note: This command might mean that publications are not sent to subscribers that should receive them, even if the publishers or subscribers have registered with other brokers in the network. When making topology changes such as this to the broker hierarchy, it is the administrator's responsibility to ensure that publishers are quiesced, and not restarted until the effects of the topology change on the subscription state have propagated through the broker network.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows

```

>> clrmqbrk [-p QMgrName] [-c ChildQMGrName] -m QMgrName <<

```

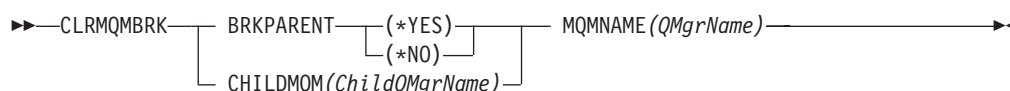
Required parameters

AIX, HP-UX, Linux, Solaris, and Windows

- p Specifies that the link is to be broken with the parent broker. If you specify this parameter, do not specify the -c parameter
- c *ChildQMgrName*
Specifies that the link is to be broken with a child broker; you also need to specify the name of the queue manager that hosts the child broker. If you specify this parameter, do not specify the -p parameter
- m *QMgrName*
The name of the queue manager hosting the broker whose link is to be broken.

Syntax

iSeries



Required parameters

iSeries

BRKPARENT

Specifies whether the link is to be broken with the parent broker. If you specify this parameter, do not specify the CHILDMQM parameter

(*YES) Breaks the link with the parent broker.

(*NO) Keeps the link with the parent broker.

CHILDMQM(*ChildQMgrName*)

Specifies that the link is to be broken with a child broker; you also need to specify the name of the queue manager that hosts the child broker. If you specify this parameter, do not specify the BRKPARENT parameter

MQMNAME(*QMgrName*)

The name of the queue manager hosting the broker whose link is to be broken.

Return codes

- 0 Command completed normally
- 10 Command completed with unexpected results
- 20 An error occurred during processing

Examples

In a broker network like this:

```

  grandparentQM
    |
  parentQM
    |
  childQM
  
```

remove the parentQM from the network like this:

1. `clrmqbrk -m grandparentQM -c parentQM`
breaks the link between the broker on grandparentQM and its child on parentQM.
2. `clrmqbrk -m parentQM -p`
breaks the link between the broker on parentQM and its parent.

3. `clrmqbrk -m parentQM -c childQM`
breaks the link between the broker on parentQM and its child on childQM.
4. `clrmqbrk -m childQM -p`
breaks the link between the broker on childQM and its parent.

If the broker on childQM is started by `strmqbrk -m childQM -p grandparentQM` at restart, the broker network now looks like this:

```

grandparentQM
|
childQM

```

Attention

If you do not issue the **clrmqbrk** command at both ends of a connection, for example, by omitting step 3 above, and you try to reconnect the brokers on parentQM and childQM at restart, reconnection fails with an AMQ5839 message at the parent, an AMQ5822 message at the child, and an AMQ5839 FDC file is generated. If you issue the **clrmqbrk** command at both ends of the connection now, it will not fix the problem. You must issue the following commands, assuming that the parent and child brokers are running with channels between them:

1. `endmqbrk -m childQM`
Wait a few seconds for the failed registration message to reach the child.
2. `clrmqbrk -m childQM -p`
3. `strmqbrk -m childQM`
Note that there is no parent argument. Wait a few seconds for the failed registration message to be removed.
4. `endmqbrk -m parentQM`
5. `endmqbrk -m childQM`
6. `clrmqbrk -m parentQM -c childQM`
7. `strmqbrk -m parentQM`
8. `strmqbrk -m childQM -p parentQM`

These commands restore the connection between the brokers on parentQM and childQM and leave the network looking like this:

```

parentQM
|
childQM

```

dlmqbrk (Delete broker)

Purpose

Use the **dlmqbrk** command to delete the broker. On iSeries, the command name is **DLTMQMBRK**. The broker must be stopped when this command is issued, and the queue manager running. If the broker is already started, you must issue the **endmqbrk** before issuing this command. To delete more than one broker in the hierarchy, it is essential that you stop (using the **endmqbrk** command) and delete each broker one at a time. Do not attempt to stop all the brokers in the hierarchy that you want to delete first and then try to delete them.

The broker must not have children when this command is issued, because they might be cut off from the rest of the network as a result. If the broker has children and this command is issued, an error message naming at least one child broker is received. Delete any children before you delete the broker or, in exceptional circumstances, before you clear the broker using the **clrmqbrk** command.

The broker performs the following actions:

1. Put-inhibits its input queues (SYSTEM.BROKER.CONTROL.QUEUE and all stream queues).
2. Deregisters all its subscribers and publishers.
3. Sends Delete Publication commands to its parent for its metatopics.
4. Deregisters all its subscriptions with the parent.
5. Processes any messages on its input queues according to their report options.

Note: You must have a dead-letter queue, because any input messages are processed according to their report options. If there is no dead-letter queue, commands might fail.

6. Deletes internal queues (purging any messages on the queues).
7. Deletes any empty input queues. that were created by the broker in question.
8. Terminates.

If the queue manager terminates before the broker has finished deleting itself (the finish is indicated by a message to the operator), the operator must issue **dlmqbrk** again when the queue manager has been restarted.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows

```
►►dlmqbrk— -m QMgrName—————►◄
```

Required parameters

AIX, HP-UX, Linux, Solaris, and Windows

-m *QMgrName*

The name of the queue manager for which the broker function is to be deleted.

Syntax

iSeries

►►—DLTMQMBRK— MQMNAME (QMgrName)—►►

Required parameters

iSeries

MQMNAME (QMgrName)

The name of the queue manager for which the broker function is to be deleted.

Return codes

- 0 Command completed normally
- 10 Command completed with unexpected results
- 20 An error occurred during processing

Examples

dlmqbrk -m exampleQM	Deletes the broker on exampleQM.
----------------------	----------------------------------

dspmqrk (Display broker status)

Purpose

Use the **dspmqrk** command to display the status of a broker. On iSeries, the command name is **DSPMQMBRK**. The status value returned from this command can be one of:

- Starting
- Running
- Stopping (immediate shutdown)
- Quiescing (controlled shutdown)
- Not active
- Ended abnormally

Syntax

AIX, HP-UX, Linux, Solaris, and Windows

```

>> dspmqrk [-m QMgrName]

```

Optional parameters

AIX, HP-UX, Linux, Solaris, and Windows

-m *QMgrName*

The name of the queue manager for which the broker status is to be displayed. If you do not specify this parameter, the command applies to the default queue manager.

Syntax

iSeries

```

>> DSPMQMBRK [MQMNAME(QMgrName)]

```

Optional parameters

iSeries

MQMNAME (*QMgrName*)

The name of the queue manager for which the broker status is to be displayed. If you do not specify this parameter, the command applies to the default queue manager.

Return codes

- | | |
|----|---|
| 0 | Command completed normally |
| 10 | Command completed with unexpected results |
| 20 | An error occurred during processing |

Examples

dspmqrk	Displays information about the broker on the default queue manager.
---------	---

dspmqbrk

dspmqbrk -m exampleQM	Displays information about the broker on exampleQM.
-----------------------	---

endmqbrk (End broker function)

Purpose

Use the **endmqbrk** command to stop a broker. On iSeries, the command name is **ENDMQMBRK**.

Control information is retained and registrations for publishers and subscribers remain in force. Messages are queued by the queue manager until the broker is restarted using the **strmqbrk** command.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows

```

>> endmqbrk [ -c ] [ -i ] [ -m QMgrName ]

```

Optional parameters

AIX, HP-UX, Linux, Solaris, and Windows

- c** Requests a controlled shutdown. This is the default value.
- i** Requests an immediate shutdown. The broker does not attempt any further gets or puts, and backs out any in-flight units-of-work. This might mean that a nonpersistent input message is published only to a subset of subscribers, or lost, depending on the broker configuration parameters. (See the description of *SyncPointIfPersistent* in “Broker configuration parameters” on page 102.)
- m QMgrName**
The name of the queue manager for which the broker function is to be ended. If you do not specify this parameter, the command is routed to the default queue manager.

Syntax

iSeries

```

>> ENDMQMBRK [ OPTION(*CNTRLD) ] [ OPTION(*IMMED) ] [ MQMNAME(QMgrName) ]

```

Optional parameters

iSeries

OPTION(*CNTRLD)

Requests a controlled shutdown. This is the default value.

OPTION(*IMMED)

Requests an immediate shutdown. The broker does not attempt any further gets or puts, and backs out any in-flight units-of-work. This might mean that a nonpersistent input message is published only to a subset of subscribers, or lost, depending on the broker configuration parameters. (See the description of *SyncPointIfPersistent* in “Broker configuration parameters” on page 102.)

endmqbrk

MQMNAME(*QMgrName*)

The name of the queue manager for which the broker function is to be ended.
If you do not specify this parameter, the command is routed to the default queue manager.

Return codes

- 0** Command completed normally
- 10** Command completed with unexpected results
- 20** An error occurred during processing

Examples

endmqbrk	Stops the broker on the default queue manager with a controlled shutdown.
endmqbrk -i -m exampleQM	Stops the broker on exampleQM immediately.

migmqbrk (Migrate broker to WebSphere Business Integration Brokers)

Purpose

Use the **migmqbrk** command to migrate a WebSphere MQ Publish/Subscribe broker to a WebSphere Business Integration Message Broker broker or a WebSphere Business Integration Event Broker broker. This command is available only on platforms that support MQSeries® Integrator Version 2.0 and above. Make sure that you have applied any necessary CSD or Fix Pack to the WebSphere MQ base product before running this command. For instructions on applying a CSD or Fix Pack, see the appropriate *Quick Beginnings* for your platform.

Read the topics in the “Migrating publish/subscribe applications” of the WebSphere Business Integration Message Broker or WebSphere Business Integration Event Broker Help System before deciding to migrate. In particular, read the “Product differences” topic, which outlines the impact that migration will have on your current broker network.

Migration exports the following state to a replacement WebSphere Business Integration Message Broker or WebSphere Business Integration Event Broker broker. This broker must reside on the same queue manager as the Publish/Subscribe broker.

Subscriptions

All client subscriptions are exported from all streams except SYSTEM.BROKER.ADMIN.STREAM

Retained publications

All retained publications in MQRFH format are exported from all streams except SYSTEM.BROKER.ADMIN.STREAM

Local publishers

Registrations for all publishers that produce local publications are exported from all streams except SYSTEM.BROKER.ADMIN.STREAM

Related brokers

If the broker is part of a multi-broker hierarchy, details of all its relations are exported. This includes the names of all streams that the broker to be migrated has in common with each relation.

The WebSphere Business Integration Message Broker broker or WebSphere Business Integration Event Broker broker and the WebSphere MQ Publish/Subscribe broker that it is to replace must have been created on the same queue manager.

When migration is complete, the Publish/Subscribe broker will be deleted automatically. Therefore, you are advised to back up the queue manager that hosts the Publish/Subscribe broker before you start the migration. If migration fails, the Publish/Subscribe broker remains operational and you can restart it.

Syntax

```
►► migmqbrk -m QMgrName ◀◀
```

Optional parameters

-m *QMgrName*

The name of the queue manager for which the broker function is to be migrated. This must match the queue manager that hosts the replacement WebSphere Business Integration Message Broker or WebSphere Business Integration Event Broker broker.

Return codes

- 0 Command completed normally
- 10 Command completed with unexpected results
- 20 An error occurred during processing

Examples

migmqbrk -m exampleQM	Migrates the broker on exampleQM.
-----------------------	-----------------------------------

strmqbrk (Start broker function)

Purpose

Use the **strmqbrk** command to start a broker, either as a restart after an **endmqbrk** command (in which case control information is maintained) or initially. On iSeries, the command name is **STRMQMBRK**.

On WebSphere MQ for Windows, the **strmqbrk** command can be added to the reference command file used when starting a queue manager automatically.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows

```

>> strmqbrk [-p ParentQMGrName] [-m QMgrName]

```

Optional parameters

AIX, HP-UX, Linux, Solaris, and Windows

-p *ParentQMGrName*

The name of the queue manager that provides the parent broker function.

Before you can add a broker to the network, channels in both directions must exist between the queue manager that hosts the new broker, and the queue manager that hosts the parent. See “Adding a broker to a network” on page 109 for more details.

On restart, this parameter is optional. If present, it must be the same as it was when previously specified. If this is the root-node broker, the queue manager specified becomes its parent. You cannot specify the name of the parent broker when you use triggering to start a broker.

When a parent has been specified, it is possible to change parentage only in exceptional circumstances in conjunction with the **clrmqbrk** command.

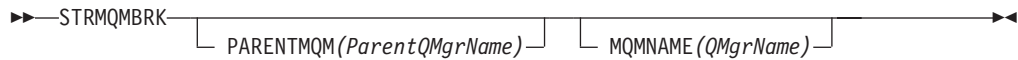
By changing a root node to become the child of an existing broker, two hierarchies can be joined. This propagates subscriptions across the two hierarchies, which now become one. After that, publications start to flow across them. To ensure predictable results, it is essential that you quiesce all publishing applications at this time. If the changed broker detects a hierarchical error (that is, if the new parent is found also to be a descendant), it immediately shuts down. The administrator must then use **clrmqbrk** at both the changed broker and the new, false parent to restore the previous status. Note that a hierarchical error is detected by propagating a message up the hierarchy, which can complete only when the relevant brokers and links are available.

-m *QMGrName*

The name of the queue manager for which the broker function is to be started. If you do not specify this parameter, the command is routed to the default queue manager.

Syntax

iSeries



Optional parameters

iSeries

PARENTMQM (*ParentQMGrName*)

The name of the queue manager that provides the parent broker function.

Before you can add a broker to the network, channels in both directions must exist between the queue manager that hosts the new broker, and the queue manager that hosts the parent. See “Adding a broker to a network” on page 109 for more details.

On restart, this parameter is optional. If present, it must be the same as it was when previously specified. If this is the root-node broker, the queue manager specified becomes its parent. You cannot specify the name of the parent broker when you use triggering to start a broker.

When a parent has been specified, it is possible to change parentage only in exceptional circumstances in conjunction with the **clrmqbrk** command.

By changing a root node to become the child of an existing broker, two hierarchies can be joined. This propagates subscriptions across the two hierarchies, which now become one. After that, publications start to flow across them. To ensure predictable results, it is essential that you quiesce all publishing applications at this time. If the changed broker detects a hierarchical error (that is, if the new parent is found also to be a descendant), it immediately shuts down. The administrator must then use **clrmqbrk** at both the changed broker and the new, false parent to restore the previous status. Note that a hierarchical error is detected by propagating a message up the hierarchy, which can complete only when the relevant brokers and links are available.

MQMNAME (*QMGrName*)

The name of the queue manager for which the broker function is to be started. If you do not specify this parameter, the command is routed to the default queue manager.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

strmqbrk -p parentQM	Starts the broker on the default queue manager specifying that it is a child of the broker on parentQM.
strmqbrk -m exampleQM	Starts the broker on exampleQM.

You can use the **runmqbrk** command to run the broker synchronously in the foreground. **runmqbrk** takes the same parameters as **strmqbrk**.

strmqbrk

Chapter 13. Message broker exit

An exit can be configured at the broker to customize publications. This exit can be used, for example, to cause traffic for different streams to be sent along different channels.

The exit is invoked after the broker has decided to send a publication to a particular broker or subscriber, and the exit can modify the publication and message descriptor. Do not change the message descriptor for a publication that is being sent between brokers.

Exits are configured in the queue manager configuration file, qm.ini or in the Broker page of the queue manager properties in WebSphere MQ Explorer (described in “Broker configuration stanza” on page 102).

The following topics are discussed in this chapter:

- “Publish/subscribe routing exit”
- “Writing a publish/subscribe routing exit program” on page 136
- “Compiling a publish/subscribe routing exit program” on page 137
- “Sample routing exit” on page 137

Publish/subscribe routing exit

MQ_PUBSUB_ROUTING_EXIT (*ExitParms*)

This call definition describes the parameters that are passed to the publish/subscribe routing exit called by the publish/subscribe broker.

Note: No entry point called MQ_PUBSUB_ROUTING_EXIT is actually provided by the broker. This is because the name of the publish/subscribe routing exit is defined by the *RoutingExitPath* parameter in the *Broker* stanza of the queue manager’s initialization file qm.ini. You can also update this parameter in the Broker page of the queue manager properties in WebSphere MQ Explorer

Parameters

ExitParms (MQPXP) – input/output
Exit parameter block.

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate the destination to which the message should be sent.

Usage notes

1. The function performed by the publish/subscribe routing exit is defined by the provider of the exit. The exit, however, must conform to the rules defined in the associated control block MQPXP.
2. No entry point called MQ_PUBSUB_ROUTING_EXIT is actually provided by the publish/subscribe broker. However, a **typedef** is provided for the name

Publish/subscribe routing exit

MQ_PUBSUB_ROUTING_EXIT in the C programming language, and this can be used to declare the user-written exit, to ensure that the parameters are correct. The following example illustrates how this can be used:

```
#include "cmqc.h"
#include "cmqxc.h"

MQ_PUBSUB_ROUTING_EXIT MyRoutingExit;

void MQENTRY MyRoutingExit(PMQXPX pExitParms)
{
    /* C language statements to perform the function of the exit */
}
```

C invocation

exitname (&ExitParms);

Declare the parameters as follows:

PMQXPX ExitParms; /* Exit parameter block */

Publish/subscribe routing exit parameter structure

The following table summarizes the fields in the structure.

Table 6. Fields in MQXPX

Field	Description	Page
<i>DestinationQMgrName</i>	Name of destination queue manager	131
<i>DestinationQName</i>	Name of destination queue	131
<i>DestinationType</i>	Type of destination	131
<i>ExitData</i>	Exit data	131
<i>ExitId</i>	Type of exit	131
<i>ExitNumber</i>	Exit number	132
<i>ExitReason</i>	Reason for invoking exit	132
<i>ExitResponse</i>	Response from exit	132
<i>ExitResponse2</i>	Reserved field	133
<i>ExitUserArea</i>	Exit user area	133
<i>Feedback</i>	Feedback code	133
<i>HeaderLength</i>	Reserved field	134
<i>MsgDescPtr</i>	Address of message descriptor (MQMD)	134
<i>MsgInLength</i>	Length of input message	134
<i>MsgInPtr</i>	Address of input message	134
<i>MsgOutLength</i>	Length of output message	134
<i>MsgOutPtr</i>	Address of output message	134
<i>QMgrName</i>	Name of local queue manager	135
<i>StreamName</i>	Name of stream	135
<i>StrucId</i>	Structure identifier	135
<i>Version</i>	Structure version number	135

The MQPXP structure describes the information that is passed to the publish/subscribe routing exit. The exit is invoked each time a broker sends a publication to a subscriber or to another broker. The exit is also invoked when a stream is initialized or terminated.

This structure is supported in the following environments: AIX, HP-UX, Linux, Solaris, and Windows.

Fields

DestinationQMgrName (MQCHAR48)

Name of destination queue manager.

This is the name of the queue manager to which the message is being sent. The name is padded with blanks to the full length of the field. The name can be altered by the exit.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. This is an input/output field to the exit.

DestinationQName (MQCHAR48)

Name of destination queue.

This is the name of the queue to which the message is being sent. The name is padded with blanks to the full length of the field. The name can be altered by the exit.

The length of this field is given by MQ_Q_NAME_LENGTH. This is an input/output field to the exit.

DestinationType (MQLONG)

Type of destination for message.

This is the type of the destination to which the message is being sent. It is one of the following:

MQDT_APPL

Application.

MQDT_BROKER

Broker.

This is an input field to the exit.

ExitData (MQCHAR32)

Exit data.

This is the fixed exit data defined by the *RoutingExitData* parameter of the *Broker* stanza in the queue manager's initialization file. The data is padded with blanks to the full length of the field. If there is no fixed exit data defined in the initialization file, this field is completely blank.

The length of this field is given by MQ_EXIT_DATA_LENGTH. This is an input field to the exit.

ExitId (MQLONG)

Type of exit.

This indicates the type of exit being called. The value is:

MQXT_PUBSUB_ROUTING_EXIT

Publish/subscribe routing exit.

This is an input field to the exit.

Publish/subscribe routing exit

ExitNumber (MQLONG)

Exit number.

This is the sequence number of the exit. The value is one.

This is an input field to the exit.

ExitReason (MQLONG)

Reason for invoking exit.

This indicates the reason why the exit is being called. Possible values are:

MQXR_INIT

Exit initialization.

This indicates that the exit for the stream identified by the *StreamName* field is being invoked for the first time. It allows the exit to acquire and initialize any resources that it might need (for example: main storage).

MQXR_TERM

Exit termination.

This indicates that the exit for the stream identified by the *StreamName* field is about to be terminated. The exit should free any resources that it has acquired since it was initialized (for example: main storage).

MQXR_MSG

Process a message.

This indicates that the exit is being invoked to process a message.

This is an input field to the exit.

ExitResponse (MQLONG)

Response from exit.

This is set by the exit to indicate how processing should continue. It must be one of the following:

MQXCC_OK

Continue normally.

This indicates that processing should continue normally. It is valid for all values of *ExitReason*.

When *ExitReason* has the value **MQXR_MSG**, *DestinationQName* and *DestinationQMGrName* identify the destination to which the message should be sent.

MQXCC_SUPPRESS_FUNCTION

Suppress function.

This indicates that the normal processing of the message should be discontinued. It is valid only when *ExitReason* has the value **MQXR_MSG**.

The processing performed on the message is determined by the **MQRO_DISCARD_MSG** option in the *Report* field of the message descriptor of the message:

- If the exit specifies **MQRO_DISCARD_MSG**, the message is discarded.
- If the exit does not specify **MQRO_DISCARD_MSG**, the message is placed on the dead-letter queue (undelivered-message queue). If

there is no dead-letter queue, or the message cannot be placed successfully on the dead-letter queue:

- The message is discarded if the *Persistence* field in the message descriptor has the value `MQPER_NOT_PERSISTENT` and the *DiscardNonPersistentPublication* parameter in the queue manager's initialization file has the value *yes*.
- In all other cases, the message is retried intermittently.

MQXCC_SUPPRESS_EXIT

Suppress exit.

This indicates that the exit should not be invoked again until termination of the stream. It is valid only when *ExitReason* has the value `MQXR_INIT` or `MQXR_MSG`.

The broker processes subsequent messages as if no publish/subscribe routing exit were defined. Processing of the current message (if there is one) continues normally; *DestinationQName* and *DestinationQMGrName* identify the destination to which the current message should be sent.

If any other value is returned by the exit, the broker processes the message as if `MQXCC_OK` had been specified.

This is an output field from the exit.

ExitResponse2 (MQLONG)

Reserved.

This is a reserved field. The value is zero.

ExitUserArea (MQBYTE16)

Exit user area.

This is a field that is available for the exit to use. It is initialized to `MQXUA_NONE` (binary zero) on the first invocation of the exit for the stream, and thereafter any changes made to this field by the exit are preserved across invocations of the exit. The first invocation of the exit is indicated by the *ExitReason* field having the value `MQXR_INIT`. There is a separate *ExitUserArea* for each stream.

The following value is defined:

MQXUA_NONE

No user information.

The value is binary zero for the length of the field.

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

The length of this field is given by `MQ_EXIT_USER_AREA_LENGTH`. This is an input/output field to the exit.

Feedback (MQLONG)

Feedback code.

This is the feedback code to be used if the exit returns `MQXCC_SUPPRESS_FUNCTION` in the *ExitResponse* field.

Publish/subscribe routing exit

On input to the exit, this field always has the value MQFB_NONE. If the exit decides to return MQXCC_SUPPRESS_FUNCTION, the exit should set *Feedback* to the value to be used for the message when the broker places it on the dead-letter queue.

If MQXCC_SUPPRESS_FUNCTION is returned by the exit, but *Feedback* still has the value MQFB_NONE, the following feedback code is used:

MQFB_STOPPED_BY_PUBSUB_EXIT

Message stopped by publish/subscribe routing exit.

This is an input/output field to the exit.

HeaderLength (MQLONG)

Reserved.

This is a reserved field. The value is zero.

This is an input field to the exit.

MsgDescPtr (PMQMD)

Address of message descriptor.

This is the address of the message descriptor (MQMD) of the message being processed. The exit can change the contents of the message descriptor, so use it with care. In particular:

- If *DestinationType* has the value MQDT_BROKER, the *CorrelId* field in the message descriptor must *not* be changed.

No message descriptor is passed to the exit if *ExitReason* is MQXR_INIT or MQXR_TERM; in these cases, *MsgDescPtr* is the null pointer.

This is an input field to the exit.

MsgInLength (MQLONG)

Length of input message data.

This is the length in bytes of the message data passed to the exit. The address of the data is given by *MsgInPtr*.

This is an input field to the exit.

MsgInPtr (PMQVOID)

Address of input message data.

This is the address of a buffer containing the message data that is input to the exit. The contents of this buffer can be modified by the exit; see *MsgOutPtr*.

This is an input field to the exit.

MsgOutLength (MQLONG)

Length of output message data.

This is the length in bytes of the message data returned by the exit. On input to the exit, this field is always zero. On output from the exit, this field is ignored if *MsgOutPtr* is the null pointer. See the description of the *MsgOutPtr* field for information about modifying the message data.

This is an input/output field to the exit.

MsgOutPtr (PMQVOID)

Address of output message data.

This is the address of a buffer containing the message data that is output from the exit. On input to the exit, this field is always the null pointer. On output

from the exit, if the value is still the null pointer, the broker sends the message specified by *MsgInPtr*, with the length given by *MsgInLength*.

If the exit needs to modify the message data, use one of the following procedures:

- If the length of the data does not change, the data can be modified in the buffer addressed by *MsgInPtr*. In this case, do not change *MsgOutPtr* and *MsgOutLength*.
- If the modified data is shorter than the original data, the data can be modified in the buffer addressed by *MsgInPtr*. In this case *MsgOutPtr* must be set to the address of the input message buffer, and *MsgOutLength* set to the new length of the message data.
- If the modified data is (or might be) longer than the original data, the exit must obtain a buffer of the required size and copy the modified data into it. In this case *MsgOutPtr* must be set to the address of the new buffer, and *MsgOutLength* set to the new length of the message data. The exit is responsible for freeing the buffer on a subsequent invocation of the exit.

Note: Because *MsgOutPtr* is always the null pointer on input to the exit, the exit must save the address of the buffer it obtains, either in *ExitUserArea*, or in a control block whose address is saved in *ExitUserArea*.

This is an input/output field to the exit.

QMgrName (MQCHAR48)

Name of local queue manager.

This is the name of the local queue manager. The name is padded with blanks to the full length of the field.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. This is an input field to the exit.

StreamName (MQCHAR48)

Name of stream.

This is the name of the stream to which the message belongs. The name is padded with blanks to the full length of the field.

The length of this field is given by MQ_OBJECT_NAME_LENGTH. This is an input field to the exit.

StrucId (MQCHAR4)

Structure identifier.

Possible values are:

MQPXP_STRUC_ID

Identifier for publish/subscribe routing-exit parameter structure.

For the C programming language, the constant

MQPXP_STRUC_ID_ARRAY is also defined; this has the same value as MQPXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)

Structure version number.

The value is:

Publish/subscribe routing exit

MQPXP_VERSION_1

Version-1 publish/subscribe routing-exit parameter structure.

MQPXP_CURRENT_VERSION

Current version of publish/subscribe routing-exit parameter structure.

This is an input field to the exit.

C declaration

```
typedef struct tagMQPXP {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;           /* Structure version number */
    MQLONG    ExitId;            /* Type of exit */
    MQLONG    ExitReason;        /* Reason for invoking exit */
    MQLONG    ExitResponse;      /* Response from exit */
    MQLONG    ExitResponse2;     /* Reserved */
    MQLONG    Feedback;          /* Feedback code */
    MQLONG    ExitNumber;        /* Exit number */
    MQBYTE16  ExitUserArea;      /* Exit user area */
    MQCHAR32  ExitData;          /* Exit data */
    MQLONG    HeaderLength;      /* Reserved */
    MQLONG    MsgInLength;       /* Length of input message data */
    MQLONG    MsgOutLength;      /* Length of output message data */
    MQLONG    DestinationType;   /* Type of destination for message */
    PMQMD     MsgDescPtr;        /* Address of message descriptor */
    PMQVOID   MsgInPtr;          /* Address of input message data */
    PMQVOID   MsgOutPtr;         /* Address of output message data */
    MQCHAR48  StreamName;        /* Name of stream */
    MQCHAR48  QMgrName;          /* Name of local queue manager */
    MQCHAR48  DestinationQName;  /* Name of destination queue */
    MQCHAR48  DestinationQMgrName; /* Name of destination queue
                                   manager */
} MQPXP;
```

Writing a publish/subscribe routing exit program

The WebSphere MQ Publish/Subscribe routing exit is a stream related exit; the parameters (for example, *ExitUserArea*) passed to the exit have the scope of a stream.

The broker uses two threads for each stream and the exit can be invoked under either thread. The broker does not call the exit for a single stream under two threads concurrently (that is, the exit does not need to serialize access to the *ExitUserArea* or other stream related data).

If the exit uses thread related resources (for example, a connection handle or queue handle) the exit must manage these resources on a thread basis. The connection handle obtained by a thread is not usable by any other thread. The exit can use operating system thread services such as **pthread_set_specific** and **pthread_get_specific** on Unix, or **TlsSetValue** and **TlsGetValue** on Windows to manage thread related resources.

The routing exit is called before a broker sends a publication to a subscriber or another broker. It is also called at initialization and termination of a stream.

Limitations on WebSphere MQ work done in the routing exit

When writing routing exit programs, be aware of the following restrictions on MQI calls:

- Do not issue **MQDISC**.

- Do not issue **MQCMIT** or **MQBACK** within the exit:
 - If you are using *SyncPointIfPersistent=yes* (described in “Broker configuration stanza” on page 102), do not take recoverable action within the exit when processing nonpersistent messages.
 - If you are using *SyncPointIfPersistent=no*, or persistent messages, the exit is invoked within the scope of the publication unit of work.

Security considerations

If the routing exit changes the destination queue or queue manager name, by default no new authority check is carried out.

Compiling a publish/subscribe routing exit program

The routing exit is a dynamically loaded library; it can be thought of as a channel-exit. For information on writing and compiling channel-exit programs see *WebSphere MQ Intercommunication*.

Sample routing exit

A sample routing exit program is provided with WebSphere MQ Publish/Subscribe (see Chapter 9, “Sample programs,” on page 91). The exit sample is invoked using the *RoutingExitPath* in the Broker stanza of queue manager initialization file (see “Broker configuration stanza” on page 102).

The sample program changes either the destination queue or queue manager, depending upon the parameters supplied, as follows:

- If the destination of the message is an application, and the stream name is the default stream:
 - If the destination queue name is Q1, change it to Q2
 - If the destination queue name is Q2, change it to Q3
 - If the destination queue name is Q3, change it to Q4
- If the destination of the message is a broker, and the stream name is `MY.ROUTING.STREAM`:
 - If the destination queue manager is queue manager 1, change it to queue manager 2.
 - If the destination queue manager is queue manager 2, change it to queue manager 3.
 - If the destination queue manager is queue manager 3, change it to queue manager 4.

Sample routing exit

Part 4. System programming

Chapter 14. Writing system management

applications	141
Format of broker administration messages.	141
Subscription deregistered message	142
Stream deleted message	142
Broker deleted message	142
Stream support messages	143
Children messages	143
Parent messages	143
MQCFH - PCF header	143
Reason codes returned from publish/subscribe messages	145
PCF Command Messages	146
Delete Publication	147
Deregister Publisher	147
Deregister Subscriber	147
Publish	148
Register Publisher	148
Register Subscriber	149
Request Update	150

Chapter 15. Finding out about other publishers

and subscribers	151
Metatopics	151
Subscribing to metatopics	152
Using wild cards	153
Example requests	153
Authorized metatopics	153
Finding out about brokers	154
Message format for metatopics	154
Parameters	155
Sample program for administration information	157
Operation	158
Example of metatopic information	159

Chapter 14. Writing system management applications

Brokers communicate with their neighbors in the hierarchy to establish the topology, and to inform their neighbors about the streams they support. They do this by publishing broker administration messages, as retained messages, using the WebSphere MQ Programmable Command Format (PCF).

Note that the format of administration information (including metatopics) might be changed in future products.

A PCF message starts with an MQCFH structure, which includes a definition of the type of command the message represents. This is followed by a succession of MQCFIN (integer parameter) and MQCFST (string parameter) structures. The PCF format is described in *WebSphere MQ Programmable Command Formats and Administration Interface*. The WebSphere MQ administration interface (MQAI) has been provided to help you write PCF applications. It is also described in *WebSphere MQ Programmable Command Formats and Administration Interface*.

The SYSTEM.BROKER.ADMIN.STREAM queue is used for broker administration messages. System management applications can subscribe to these messages, provided that they have the correct security authorization. Subscription requests for these topics are sent to the SYSTEM.BROKER.CONTROL.QUEUE in the normal way.

Topics starting 'MQ/' are reserved for WebSphere MQ use, but other topics can be defined. The broker passes these publications to subscribers in the same way as for other streams.

Brokers publish on the 'MQ/QMgrName/Children' and 'MQ/QMgrName/Parent' topics if applicable. This enables applications to build a view of the broker topology.

The 'MQ/QMgrName/StreamSupport' topic is published on by all brokers. This enables applications to build a view of the stream topology in relation to the broker topology.

Brokers also publish messages to this queue when a stream or broker has been deleted, and when a subscription has been deregistered by the broker because it is no longer valid.

This chapter discusses the following topics:

- "Format of broker administration messages"
- "MQCFH - PCF header" on page 143
- "PCF Command Messages" on page 146

Metatopics are published on the stream to which they relate so the relevant ones are published on SYSTEM.BROKER.ADMIN.STREAM. For information about metatopics see "Metatopics" on page 151.

Format of broker administration messages

The broker sends administration messages as **Publish** messages in PCF format. The following parameters are always present:

Broker administration messages

PublicationOptions (MQCFIN)

MQPUBO_RETAIN_PUBLICATION is set if the publication is retained.

StreamName (MQCFST)

Set to the reserved stream name 'SYSTEM.BROKER.ADMIN.STREAM'.

Topic (MQCFST)

This is one of the following:

- 'MQ/*QMgrName*/Event/SubscriptionDeregistered'
- 'MQ/*QMgrName*/Event/StreamDeleted'
- 'MQ/*QMgrName*/Event/BrokerDeleted'
- 'MQ/*QMgrName*/StreamSupport'
- 'MQ/*QMgrName*/Children'
- 'MQ/*QMgrName*/Parent'

where *QMgrName* is the queue manager name of the broker sending the message (this is 48 characters long, padded with blanks if necessary).

PublishTimestamp (MQCFST)

Set to the time of publication (Universal time).

Subscription deregistered message

An 'MQ/*QMgrName*/Event/SubscriptionDeregistered' message is published when a subscription is deregistered by the broker because it has become invalid (for example, it is no longer authorized).

For 'MQ/*QMgrName*/Event/SubscriptionDeregistered' messages, the following group of parameters is published to identify the subscription that has been removed by the broker:

- *RegistrationStreamName*
- *RegistrationTopic*
- *RegistrationQMgrName*
- *RegistrationQName*
- *RegistrationCorrelId* (if applicable)
- *RegistrationUserIdentifier*
- *RegistrationRegistrationOptions*

These additional parameters are described in "Message format for metatopics" on page 154.

Stream deleted message

An 'MQ/*QMgrName*/Event/StreamDeleted' message is published when a stream is deleted. The following additional parameter is present:

RegistrationStreamName (MQCFST)

Name of deleted stream (parameter identifier:
MQCACF_REG_STREAM_NAME).

Broker deleted message

When a broker is deleted with the *dlmqbrk* command, it publishes an 'MQ/*QMgrName*/Event/BrokerDeleted' message.

The administrator is advised to stop affected application programs before making changes to broker network and stream topology. However, a program could be written to subscribe to these administrative event topics and take appropriate action. In the case of the BrokerDeleted event, such a program cannot rely on this

message being propagated to the parent, but the program will receive the message if it has subscribed to this topic at the affected broker.

Stream support messages

An 'MQ/QMgrName/StreamSupport' message (a retained publication) gives information about which streams the broker supports. The following parameter is repeated for each stream supported:

SupportedStreamName (MQCFST)

Name of supported stream (parameter identifier: MQCACF_SUPPORTED_STREAM_NAME).

Children messages

An 'MQ/QMgrName/Children' message (a retained publication) gives information about a broker's children. It is published only by those brokers that have children. The following parameter is repeated for each child:

QMgrName (MQCFST)

Queue manager name of child broker (parameter identifier: MQCACF_CHILD_Q_MGR_NAME).

This list gives all the broker's immediate children in the hierarchy.

Parent messages

An 'MQ/QMgrName/Parent' message (a retained publication) gives information about a broker's parent. It is published only by those brokers that have a parent. The following parameter occurs once:

QMgrName (MQCFST)

Queue manager name of parent broker (parameter identifier: MQCACF_PARENT_Q_MGR_NAME).

MQCFH - PCF header

Each message or response in PCF format starts with an MQCFH structure. The field contents of the MQCFH structure for WebSphere MQ Publish/Subscribe are as follows:

Type (MQLONG)

Structure type.

The following values are valid:

MQCFT_COMMAND

Command message (for example, Publish, Register Subscribers).

MQCFT_RESPONSE

Message is a response to a command.

StrucLength (MQLONG)

Structure length. The value must be MQCFH_STRUC_LENGTH.

Version (MQLONG)

Structure version number. The value must be MQCFH_VERSION_1.

Command (MQLONG)

Command identifier.

For a command message, this identifies the function to be performed. For a response message, it identifies the command to which this is the reply. The following values are valid:

MQCMD_DELETE_PUBLICATION

Delete Publication

MQCMD_DEREGISTER_PUBLISHER

Deregister Publisher

MQCMD_DEREGISTER_SUBSCRIBER

Deregister Subscriber

MQCMD_PUBLISH

Publish

MQCMD_REGISTER_PUBLISHER

Register Publisher

MQCMD_REGISTER_SUBSCRIBER

Register Subscriber

MQCMD_REQUEST_UPDATE

Request Update

MQCMD_BROKER_INTERNAL

Used internally by brokers

MsgSeqNumber (MQLONG)

Message sequence number. The value must be 1 for WebSphere MQ Publish/Subscribe messages and responses.

Control (MQLONG)

Control options.

The value must be MQCFC_LAST for WebSphere MQ Publish/Subscribe messages and responses.

CompCode (MQLONG)

Completion code.

This field is meaningful only for a response; its value is not significant for a command. The following values are possible:

MQCC_OK

Command completed successfully.

MQCC_WARNING

Command completed with warning.

MQCC_FAILED

Command failed.

Reason (MQLONG)

Reason code qualifying completion code.

This field is meaningful only for a response; its value is not significant for a command.

The reason codes that might be returned in response to a command are listed in “Reason codes returned from publish/subscribe messages” on page 145.

ParameterCount (MQLONG)

Count of parameter structures (MQCFIN, MQCFST) following.

The value of this field is zero or greater.

Reason codes returned from publish/subscribe messages

The following reason codes can be returned by a broker in response to any command message in PCF format. They are described in *WebSphere MQ Programmable Command Formats and Administration Interface*.

MQRCCF_CFH_COMMAND_ERROR

Command identifier not valid.

MQRCCF_CFH_CONTROL_ERROR

Control option not valid.

MQRCCF_CFH_LENGTH_ERROR

Structure length not valid.

MQRCCF_CFH_MSG_SEQ_NUMBER_ERROR

Message sequence number not valid.

MQRCCF_CFH_PARM_COUNT_ERROR

Parameter count not valid.

MQRCCF_CFH_TYPE_ERROR

Type not valid.

MQRCCF_CFH_VERSION_ERROR

Structure version number not valid.

MQRCCF_CFIN_DUPLICATE_PARM

Duplicate MQCFIN parameter.

MQRCCF_CFIN_LENGTH_ERROR

MQCFIN structure length not valid.

MQRCCF_CFIN_PARM_ID_ERROR

Parameter identifier not valid.

MQRCCF_CFST_DUPLICATE_PARM

Duplicate MQCFST parameter.

MQRCCF_CFST_LENGTH_ERROR

MQCFST structure length not valid.

MQRCCF_CFST_PARM_ID_ERROR

Parameter identifier not valid.

MQRCCF_CFST_STRING_LENGTH_ERR

MQCFST string length not valid.

MQRCCF_COMMAND_FAILED

Command failed.

MQRCCF_ENCODING_ERROR

Encoding error.

MQRCCF_INCORRECT_Q

Command sent to wrong broker queue.

MQRCCF_MD_FORMAT_ERROR

Format not valid.

MQRCCF_MSG_LENGTH_ERROR

Message length not valid.

MQRCCF_PARM_COUNT_TOO_SMALL

Mandatory parameter for command missing.

MQRCCF_STRUCTURE_TYPE_ERROR

Structure type invalid.

The following reason codes might be returned by a broker in response to a command message in PCF format, depending on the parameters used in that message. They are described in *WebSphere MQ Messages*.

MQRCCF_CORREL_ID_ERROR

Correlation identifier used as part of identity but is all binary zero.

MQRCCF_DEL_OPTIONS_ERROR

Invalid delete options supplied.

MQRCCF_DUPLICATE_IDENTITY

Publisher or subscriber identity already assigned to another user ID.

MQRCCF_INCORRECT_STREAM

Stream name different from queue name.

MQRCCF_NO_RETAINED_MSG

No retained message exists for this topic.

MQRCCF_NOT_AUTHORIZED

Subscriber not authorized to browse broker's stream queue or subscriber queue.

MQRCCF_NOT_REGISTERED

Publisher or subscriber not registered.

MQRCCF_PUB_OPTIONS_ERROR

Invalid publication options supplied.

MQRCCF_Q_MGR_NAME_ERROR

Queue manager name invalid.

MQRCCF_Q_NAME_ERROR

Queue name invalid.

MQRCCF_REG_OPTIONS_ERROR

Invalid registration options supplied.

MQRCCF_STREAM_ERROR

Stream name too long or contains invalid characters.

MQRCCF_TOPIC_ERROR

Topic name has an invalid length or contains invalid characters.

MQRCCF_UNKNOWN_STREAM

Stream not defined to broker and cannot be created.

PCF Command Messages

This section lists the parameters and options that are relevant for each command message in PCF format. Parameter identifiers and types (MQCFIN or MQCFST) are shown. Broker administration and metatopic messages use the PCF format.

The usage of each parameter and option is the same as for the corresponding command messages in RFH format, which are described in Chapter 7, "Publish/Subscribe command messages," on page 57. In principle, publishers and subscribers could send command messages to a broker in PCF format but **this is not recommended**. Use the RFH format for command messages to ensure interoperability with other WebSphere business integration functions.

Delete Publication

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

DeleteOptions (MQCFIN)

Delete options (parameter identifier: MQIACF_DELETE_OPTIONS).

The following option can be set:

MQDELO_LOCAL

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

Deregister Publisher

RegistrationOptions (MQCFIN)

Registration options (parameter identifier:

MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_DEREGISTER_ALL

MQREGO_CORREL_ID_AS_IDENTITY

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

QMgrName (MQCFST)

Publisher's queue manager name (parameter identifier:

MQCA_Q_MGR_NAME).

QName (MQCFST)

Publisher's queue name (parameter identifier: MQCA_Q_NAME).

Deregister Subscriber

RegistrationOptions (MQCFIN)

Registration options (parameter identifier:

MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_DEREGISTER_ALL

MQREGO_CORREL_ID_AS_IDENTITY

MQREGO_LEAVE_ONLY

MQREGO_VARIABLE_USER_ID

MQREGO_FULL_RESPONSE

SubName (MQCFST)

Subscription name (parameter identifier: MQCACF_SUBSCRIPTION_NAME).

SubIdentity (MQCFST)

Subscription identity (parameter identifier:

MQCACF_SUBSCRIPTION_IDENTITY).

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

PCF Command Messages

QMgrName (MQCFST)

Subscriber's queue manager name (parameter identifier: MQCA_Q_MGR_NAME).

QName (MQCFST)

Subscriber's queue name (parameter identifier: MQCA_Q_NAME).

Publish

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

RegistrationOptions (MQCFIN)

Registration options (parameter identifier: MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_ANONYMOUS
MQREGO_LOCAL
MQREGO_DIRECT_REQUESTS
MQREGO_CORREL_ID_AS_IDENTITY

PublicationOptions (MQCFIN)

Publication options (parameter identifier: MQIACF_PUBLICATION_OPTIONS).

The following options can be set:

MQPUBO_NO_REGISTRATION
MQPUBO_RETAIN_PUBLICATION
MQPUBO_IS_RETAINED_PUBLICATION
MQPUBO_OTHER_SUBSCRIBERS_ONLY
MQPUBO_CORREL_ID_AS_IDENTITY

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

QMgrName (MQCFST)

Publisher's queue manager name (parameter identifier: MQCA_Q_MGR_NAME).

QName (MQCFST)

Publisher's queue name (parameter identifier: MQCA_Q_NAME).

PublishTimestamp (MQCFST)

Publication timestamp (parameter identifier: MQCACF_PUBLISH_TIMESTAMP).

SequenceNumber (MQCFIN)

Publication sequence number (parameter identifier: MQIACF_SEQUENCE_NUMBER).

StringData (MQCFST)

String publication data (parameter identifier: MQCACF_STRING_DATA).

IntegerData (MQCFIN)

Integer publication data (parameter identifier: MQIACF_INTEGER_DATA).

Register Publisher

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

RegistrationOptions (MQCFIN)

Registration options (parameter identifier: MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_ANONYMOUS
 MQREGO_LOCAL
 MQREGO_DIRECT_REQUESTS
 MQREGO_CORREL_ID_AS_IDENTITY

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

QMgrName (MQCFST)

Publisher's queue manager name (parameter identifier: MQCA_Q_MGR_NAME).

QName (MQCFST)

Publisher's queue name (parameter identifier: MQCA_Q_NAME).

Register Subscriber

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

RegistrationOptions (MQCFIN)

Registration options (parameter identifier: MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_ANONYMOUS
 MQREGO_LOCAL
 MQREGO_NEW_PUBLICATIONS_ONLY
 MQREGO_PUBLISH_ON_REQUEST_ONLY
 MQREGO_CORREL_ID_AS_IDENTITY
 MQREGO_INCLUDE_STREAM_NAME
 MQREGO_INFORM_IF_RETAINED
 MQREGO_DUPLICATES_OK
 MQREGO_NON_PERSISTENT
 MQREGO_PERSISTENT
 MQREGO_PERSISTENT_AS_PUBLISH
 MQREGO_PERSISTENT_AS_Q
 MQREGO_ADD_NAME
 MQREGO_VARIABLE_USER_ID
 MQREGO_NO_ALTERATION
 MQREGO_JOIN_SHARED
 MQREGO_JOIN_EXCLUSIVE
 MQREGO_LOCKED
 MQREGO_FULL_RESPONSE

SubName (MQCFST)

Subscription name (parameter identifier: MQCACF_SUBSCRIPTION_NAME).

SubIdentity (MQCFST)

Subscription identity (parameter identifier: MQCACF_SUBSCRIPTION_IDENTITY).

SubUserData (MQCFST)

Subscription user data (parameter identifier: MQCACF_SUBSCRIPTION_USER_DATA).

PCF Command Messages

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

QMgrName (MQCFST)

Subscriber's queue manager name (parameter identifier: MQCA_Q_MGR_NAME).

QName (MQCFST)

Subscriber's queue name (parameter identifier: MQCA_Q_NAME).

Request Update

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

RegistrationOptions (MQCFIN)

Registration options (parameter identifier: MQIACF_REGISTRATION_OPTIONS).

The following options can be set:

MQREGO_CORREL_ID_AS IDENTITY

MQREGO_VARIABLE_USER_ID

SubName (MQCFST)

Subscription name (parameter identifier: MQCACF_SUBSCRIPTION_NAME).

StreamName (MQCFST)

Stream name (parameter identifier: MQCACF_STREAM_NAME).

QMgrName (MQCFST)

Subscriber's queue manager name (parameter identifier: MQCA_Q_MGR_NAME).

QName (MQCFST)

Subscriber's queue name (parameter identifier: MQCA_Q_NAME).

Chapter 15. Finding out about other publishers and subscribers

Brokers publish information about the publishers and subscribers that are registered with them. The information is published as a special set of topics, known as *metatopics*, within each supported stream. They are published as persistent messages, and use the default priority for the stream queue (at the last time the broker started).

Applications can subscribe to this information in the same way as they can register any other subscription. Whenever the information changes, brokers publish the changed information in the form of retained publications so that new subscribers to it receive the current state.

Metatopic command messages can be sent to a broker using the WebSphere MQ Programmable Command Format (PCF), which is described in Chapter 14, “Writing system management applications,” on page 141. Publications containing the metatopic data are sent in PCF format, as are any broker response messages. This is illustrated in the “Sample program for administration information” on page 157.

Alternatively, metatopic command messages can be sent with the Rules and Formatting header (RFH), which is described in Chapter 6, “Format of command messages,” on page 47. In this case any broker response messages are in RFH format, but publications containing the metatopic data are sent in PCF format.

This chapter discusses the following topics:

- “Metatopics”
- “Subscribing to metatopics” on page 152
- “Authorized metatopics” on page 153
- “Finding out about brokers” on page 154
- “Message format for metatopics” on page 154

Metatopics

Brokers publish information about the publishers and subscribers that are registered with them. The information is published as a special set of topics, known as metatopics, within each supported stream.

Each broker publishes on metatopics to each stream to describe the publishers, subscribers and topics on that stream. Metatopics include subscribers to metatopics. All metatopic publications are global.

Metatopics always begin with ‘MQ/’, and topics starting with ‘MQ/’ are reserved for all streams. These metatopic strings are of the form:

- ‘MQ/S/QMgrName/Publishers/Topics’
- ‘MQ/S/QMgrName/Publishers/Summary’
- ‘MQ/S/QMgrName/Publishers/Summary/Topic’
- ‘MQ/S/QMgrName/Publishers/Identities’
- ‘MQ/S/QMgrName/Publishers/Identities/Topic’
- ‘MQ/SA/QMgrName/Publishers/AllIdentities’
- ‘MQ/SA/QMgrName/Publishers/AllIdentities/Topic’

Metatopics

- 'MQ/S/QMgrName/Subscribers/Topics'
- 'MQ/S/QMgrName/Subscribers/Summary'
- 'MQ/S/QMgrName/Subscribers/Summary/Topic'
- 'MQ/S/QMgrName/Subscribers/Identities'
- 'MQ/S/QMgrName/Subscribers/Identities/Topic'
- 'MQ/SA/QMgrName/Subscribers/AllIdentities'
- 'MQ/SA/QMgrName/Subscribers/AllIdentities/Topic'

Where:

- *QMGrName* is the name of the broker's queue manager. This is 48 characters long padded with blanks if necessary.
- *Topic* is any topic for which the broker has a registered publisher or subscriber (depending on whether the subscription is for publishers or subscribers).

Metatopics that do not include *Topic* each represent a single metatopic (for one broker), so a broker receiving a **Register Subscriber** message for one of these metatopics generates one retained **Publish** message as a result (additional retained **Publish** messages are generated whenever the information changes). However, for metatopics that do include *Topic*, one retained **Publish** message is generated for each registered topic that matches the *Topic* specification (and again further messages are generated as the information changes).

The strings in the fifth part of the metatopic offer varying levels of detail, as follows:

Summary

Minimal information including counts. If *Topic* is included, one message is generated for each matching topic.

Topics A list of registered topics in a single message.

Identities

Identities of publishers or subscribers, including user ID and time of registration. If *Topic* is included, one message is generated for each matching topic, otherwise all identities are packaged into a single message. Anonymous publishers or subscribers are not included (this means that no message is generated for topics that have only anonymous publishers and subscribers registered).

AllIdentities

This is the equivalent of *Identities* for authorized metatopics (see "Authorized metatopics" on page 153) and gives the same information, but also includes anonymous publishers and subscribers.

If an application subscribes to an 'AllIdentities' metatopic, the application requires **altusr** authority for the queue manager, as well as the normal **browse** authority for that stream queue.

Subscribing to metatopics

Applications should use this facility carefully because it can produce a large amount of data. Applications using metatopics are recommended to register subscriptions with each broker individually (using the `SYSTEM.BROKER.ADMIN.STREAM` to determine the queue manager names of the brokers). These applications are recommended to subscribe only to the information from that broker and set the `MQREGO_PUBLISH_ON_REQUEST` option in the **Register Subscriber** message and use **Request Update** to minimize network traffic.

Using wild cards

Metatopics describing subscribers include information about wild card subscriptions. This is an exception to the rule that publications should not include wild cards in their topics.

In subscriptions to metatopics, wild cards can be used in the usual way. For example, if the metatopic 'MQ/S/QM1/Publishers/S*' is specified, this matches 'MQ/S/QM1/Publishers/Summary' plus all the 'MQ/S/QM1/Publishers/Summary/*Topic*' metatopics (one for each topic registered implicitly or explicitly for publishers, except those published anonymously), and the broker sends this number of retained **Publish** messages as a result.

If the metatopic 'MQ/S/QM1/Subscribers/S*' is specified, the resultant messages show all the topics registered for subscribers (except those registered anonymously), including wildcard subscriptions. (The wildcard characters in metatopics match only wildcard subscriptions.)

A wildcard subscription of the form '*' gives all topics on a stream except the metatopics. You need to specify at least the first five characters ('MQ/S/') to receive publications about metatopics.

On WebSphere MQ for UNIX systems, you need to prevent the shell from interpreting the meaning of special characters, for example: *. Depending on the shell you are using, you might have to enclose the wildcard characters in single quotation marks or double quotation marks, or use a backslash.

Example requests

The following examples show valid metatopic requests:

- To find out what topics are being published on QM22 in a single **Publish** message:
MQ/S/QM22/Publishers/Topics
No information is returned about publishers' identities.
- To find the identities of each subscriber (except anonymous subscribers) on all brokers in the network, for any topics starting with 'Trade/':
MQ/S/*/Subscribers/Identities/Trade/*

One **Publish** message is generated for each matching topic, by each broker. Requesting this much information could have an adverse effect on the performance of your system.

Authorized metatopics

There is a subclass of metatopics, called *authorized metatopics*, that are available only to users with **altusr** authority for that queue manager. These show the identities of all publishers and subscribers, including the anonymous ones. Subscribers (who must be authorized) receive only authorized metatopics by specifying at least the first six characters 'MQ/SA/'. A wildcard subscription of the form 'MQ/S*' gives no metatopics at all, 'MQ/SA/*' gives all the authorized metatopics and 'MQ/S/*' gives all the others.

Finding out about brokers

To find out about all brokers, a subscriber can specify a *Topic* parameter of, say, 'MQ/S/*/Publishers/Summary'. If no *StreamName* parameter is specified, this defaults to the default stream, which all brokers support. At least one message is received from each broker that is connected. More than one message might be received from a broker if the state changes. For efficiency, however, it is recommended to register a subscription with each broker individually.

To determine which brokers support a particular stream, the program can issue the **Register Subscriber** command to the SYSTEM.BROKER.ADMIN.STREAM at its local broker and specify an appropriate StreamSupport topic.

Message format for metatopics

These messages are sent as **Publish** messages in PCF format with MQPUBO_RETAIN_PUBLICATION (for ongoing subscriptions registered with **Register Subscriber**). In these messages, *Command* is MQCMD_PUBLISH, and *Type* is MQCFT_COMMAND.

The following table summarizes which parameters are included for which metatopics. An explanation of each parameter follows the table.

Table 7. Parameters for publisher and subscriber information messages

	Topics	Summary	Summary /<Topic>	Identities ¹	Identities /<Topic> ¹
Number of messages sent	1	1	1 for each topic	1	1 for each topic
<i>StreamName</i>	Y	Y	Y	Y	Y
<i>Topic</i>	Y	Y	Y	Y	Y
<i>PublishTimestamp</i>	Y	Y	Y	Y	Y
<i>BrokerCount</i>	Y	Y	Y	Y	Y
<i>ApplCount</i>	Y	Y	Y	Y	Y
<i>AnonymousCount</i>	Y	Y	Y	Y	Y
<i>RegistrationTopic</i>	Y ²	N	N ³	N	N ³
<i>RegistrationQMgrName</i>	N	N	N	Y	Y
<i>RegistrationQName</i>	N	N	N	Y	Y
<i>RegistrationCorrelId</i>	N	N	N	Y	Y
<i>RegistrationUserIdentifier</i>	N	N	N	Y	Y
<i>RegistrationRegistrationOptions</i>	N	N	N	N	Y
<i>RegistrationTime</i>	N	N	N	N	Y
<i>RegistrationSubName</i>	N	N	N	N	Y
<i>RegistrationSubUserData</i>	N	N	N	N	Y
<i>RegistrationSubIdentity</i>	N	N	N	N	Y ⁴
Notes: 1. 'AllIdentities' subscriptions are the same except that they include anonymous as well as non-anonymous publishers and subscribers. 2. Repeated for each registered topic. 3. <i>Topic</i> parameter contains the registered topic. 4. Repeated.					

Parameters

These parameters might be included in publisher and subscriber information messages sent by the broker. Table 7 summarizes the parameters that are used for each metatopic.

SubName (MQCFST)

Subscription Name (parameter identifier: MQCACF_SUBSCRIPTION_NAME).

Name of the subscription for which this information applies.

SubIdentity (MQCFST)

Subscription identity (parameter identifier: MQCACF_SUBSCRIPTION_IDENTITY).

Identity registered to this subscription for which this information applies. Can be repeated.

SubUserData (MQCFST)

Subscription user data (parameter identifier: MQCACF_SUBSCRIPTION_USER_DATA).

User data of the subscription for which this information applies.

Metatopic message format

StreamName (MQCFST)

Stream Name (parameter identifier: MQCACF_STREAM_NAME).

Name of the stream for which this information applies.

Topic (MQCFST)

Topic (parameter identifier: MQCACF_TOPIC).

The metatopic under which this publication is published. These are listed in “Metatopics” on page 151.

PublishTimestamp (MQCFST)

Time this **Publish** message was generated (parameter identifier: MQCACF_PUBLISH_TIMESTAMP).

This is of length 16 characters, in the format YYYYMMDDHHMMSSTH, using Universal Time.

BrokerCount (MQCFIN)

Number of broker publishers or subscribers (parameter identifier: MQIACF_BROKER_COUNT).

Count of publisher or subscriber registrations from brokers, for the specified topic if this is a ‘MQ/QMgrName/.../Topic’ message.

For publishers, this count is normally zero, because brokers do not register as publishers. The role of a broker in acting as a publisher itself for metatopics on stream queues is not counted, nor is its role as a publisher for administrative topics on the SYSTEM.BROKER.ADMIN.STREAM stream.

ApplCount (MQCFIN)

Number of application publishers or subscribers (parameter identifier: MQIACF_APPL_COUNT).

Count of publisher or subscriber registrations from applications, for the specified topic if this is a ‘MQ/S/QMgrName/.../Topic’ or ‘MQ/SA/QMgrName/.../Topic’ message. ‘MQ/SA/QMgrName/.../Topic’ includes anonymous registrations.

AnonymousCount (MQCFIN)

Number of anonymous publishers or subscribers (parameter identifier: MQIACF_ANONYMOUS_COUNT).

Count of anonymous publisher or subscriber registrations from applications, for the specified topic if this is a ‘MQ/SA/QMgrName/.../Topic’ message.

RegistrationTopic (MQCFST)

Topic (parameter identifier: MQCACF_REG_TOPIC).

A topic for which at least one publisher or subscriber is registered. Wild cards are not present for publishers, but might be for subscribers.

This parameter is repeated for as many topics as necessary for ‘MQ/S/QMgrName/Publishers/Topics’ and ‘MQ/S/QMgrName/Subscribers/Topics’ messages. Each topic is present only once, even if there are several publishers or subscribers registered for the same topic.

RegistrationQMGrName (MQCFST)

Publisher’s or subscriber’s queue manager name (parameter identifier: MQCACF_REG_Q_MGR_NAME).

RegistrationQName (MQCFST)

Publisher's or subscriber's queue name (parameter identifier: MQCACF_REG_Q_NAME).

RegistrationCorrelId (MQCFST)

Publisher's or subscriber's correlation identifier (parameter identifier: MQCACF_REG_CORREL_ID).

This is a 48-byte character string of hexadecimal characters representing the contents of the 24-byte binary correlation identifier. Each character in the string is in the range 0 through 9 or A through F.

This parameter is present only if the publisher's or subscriber's identity includes a correlation identifier.

RegistrationUserIdentifier (MQCFST)

Publisher's or subscriber's user ID (parameter identifier: MQCACF_REG_USER_ID).

RegistrationRegistrationOptions (MQCFST)

Publisher's or subscriber's registration options (parameter identifier: MQIACF_REG_REG_OPTIONS).

RegistrationOptions parameter as specified (or defaulted) by the publisher or subscriber when it registered.

RegistrationTime (MQCFST)

Registration time (parameter identifier: MQCACF_REG_TIME).

This is of length 16 characters, in the format YYYYMMDDHHMSSSTH, using Universal Time.

Sample program for administration information

The sample administration program attaches to a broker, subscribes to the appropriate streams to obtain the required metatopic information, and then detaches from the broker. The following *RegistrationOptions* are used:

```
MQREGO_ANONYMOUS
MQREGO_PUBLISH_ON_REQUEST_ONLY
```

The information listed below can be dumped into a file or to standard output.

- The parent and children for the broker.
- All the streams supported at the broker (unless overridden by the **-s** option).
- All the subscribers and publishers registered for these streams (unless overridden by the **-p** or **-u** options), with the following parameters:

```
StreamName
Topic (max 255 chars)
BrokerCount
ApplCount
AnonymousCount
RegistrationQMgrName
RegistrationQName
RegistrationCorrelId
RegistrationUserIdentifier
RegistrationRegistrationOptions
RegistrationTime
```

- All retained messages at the broker for the given topic (only if the **-r** option is set), with the following parameters:

```
StreamName
```

Sample administration program

Topic (max 255 chars)
StringData (max 255 chars, PCF only)
IntegerData (PCF only)
QMgrName
QName
SequenceNumber
PublishTimestamp
Expiry

Operation

To run the sample administration program, first run `amqspda.tst` on the queue manager. Then enter the following:

`amqspsd options`

where *options* are any of the following:

-l *LogFileName*

The name of the log file that the information is sent to.

The default is that output is sent to the screen (stdio).

-m *QMgrName*

The queue manager name.

The default is that the default queue manager is used.

-q *QName*

The name of the queue that is subscribed to.

The default is that the program attempts to create a permanent-dynamic queue based on AMQSPSDA.PERMDYN.MODEL.QUEUE. This queue is deleted at program termination.

-s *StreamName*

The stream name.

The default is that all streams are dumped.

-t *Topic*

The topic.

The default is that * (all topics) are used as the topic.

-r *Topic*

Dump retained messages for this topic (* can be used for all topics).

The default is not to dump retained messages.

-p Dump information for publishers only.

The default is to dump information for publishers and subscribers.

-u Dump information for subscribers only.

The default is to dump information for publishers and subscribers.

-a Dump information for anonymous publishers and subscribers.

The default is to dump information for non-anonymous publishers and subscribers.

On successful termination, zero is returned to any calling application.

Example of metatopic information

Here is an example of output from the sample administration program, which was obtained from a newly created broker using the command:

```
amqspdsd -m PubSub -r *
```

It shows two retained messages, one in RFH and one in PCF format.

```
WebSphere MQ Message Broker Dumper
Start time Wed-22-Dec-2004 10:35:31
```

Broker Hierarchy

```
-----
QMgrName:
  PubSub
Parent:
  None
Children:
  None
```

Streams supported

```
-----
SYSTEM.BROKER.DEFAULT.STREAM
SYSTEM.BROKER.ADMIN.STREAM
```

Publishers

```
-----
StreamName: SYSTEM.BROKER.ADMIN.STREAM
  None
StreamName: SYSTEM.BROKER.DEFAULT.STREAM
  Topic: Topic 3
    BrokerCount: 0
    ApplCount: 1
    AnonymousCount: 0
    RegistrationQMgrName: PubSub
    RegistrationQName: Q2
    RegistrationUserIdentifier: hgdd
    RegistrationRegistrationOptions: 0 : MQREGO_NONE
    RegistrationTime: 1998111810350435
  Topic: Topic 2
    BrokerCount: 0
    ApplCount: 1
    AnonymousCount: 0
    RegistrationQMgrName: PubSub
    RegistrationQName: Q2
    RegistrationUserIdentifier: hgdd
    RegistrationRegistrationOptions: 0 : MQREGO_NONE
    RegistrationTime: 1998111810350435
  Topic: Topic 1
    BrokerCount: 0
    ApplCount: 1
    AnonymousCount: 0
    RegistrationQMgrName: PubSub
    RegistrationQName: Q1
    RegistrationUserIdentifier: hgdd
    RegistrationRegistrationOptions: 0 : MQREGO_NONE
    RegistrationTime: 1998111810341148
```

Subscribers

```
-----
StreamName: SYSTEM.BROKER.ADMIN.STREAM
  Topic: MQ/PubSub /StreamSupport
    BrokerCount: 0
    ApplCount: 1
    AnonymousCount: 0
    RegistrationQMgrName: PubSub
    RegistrationQName: SYSTEM.BROKER.INTER.BROKER.COMMUNICATIONS
```

Sample administration program

```
RegistrationCorrelId: 414D5159010100000000000000000000000000000000
RegistrationUserIdentifier: mqm
RegistrationRegistrationOptions: 17 : MQREGO_CORREL_ID_AS_IDENTITY
                                   MQREGO_NEW_PUBLICATIONS_ONLY

RegistrationTime: 1998111810330750
Topic: MQ/S/PubSub                /Subscribers/Identities/*
BrokerCount: 0
ApplCount: 1
AnonymousCount: 1
StreamName: SYSTEM.BROKER.DEFAULT.STREAM
Topic: MQ/S/PubSub                /Subscribers/Identities/*
BrokerCount: 0
ApplCount: 1
AnonymousCount: 1

Retained messages
-----
StreamName: SYSTEM.BROKER.DEFAULT.STREAM
RFH Message
  Expiry: -1
  Topic: Topic 2
  Topic: Topic 3
  QMgrName: PubSub
  QName: Q2
  SequenceNumber: None
  PublishTimestamp: None

**** Message **** length - 92 bytes

0000: 0100 0000 2400 0000 0100 0000 0000 0000 '....¢.....'
0010: 0100 0000 0100 0000 0000 0000 0000 0000 '.....'
0020: 0200 0000 0400 0000 2800 0000 DB0B 0000 '.....(....³...'
0030: 0000 0000 1200 0000 4D79 2072 6574 6169 '.....My retai'
0040: 6E65 6420 7374 7269 6E67 0000 0300 0000 'ned string.....'
0050: 1000 0000 3804 0000 15CD 5B07 '....8...."£.'

PCF Message
  Expiry: -1
  Topic: Topic 1
  QMgrName: PubSub
  QName: Q1
  SequenceNumber: None
  PublishTimestamp: None
  IntegerData: 123456789
  StringData: My retained string
```

Part 5. Appendixes

Appendix A. Header files

This appendix lists the C-language header files necessary for publish/subscribe applications:

cmqpsc.h

Contains string constants for publish/subscribe messages using the MQRFH header.

cmqc.h

Contains elementary data types, and some named constants for events and PCF commands.

cmqcfc.h

Contains named constants specific to publish/subscribe messages, definitions for PCF structures, and additional named constants for events and PCF commands.

cmqbc.h

Contains definitions unique to the MQAI. This file is required only for metatopics and system management functions.

Appendix B. 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:

AIX	IBM	IBMLink™
iSeries	MQSeries	OS/2
SupportPac	Tivoli	WebSphere
z/OS		

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

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

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be the trademarks or service marks of others.

Index

A

- access control
 - defining 101
 - using streams 11
- AccountingToken parameter
 - publications forwarded by broker 31
- adding a broker to a network 109
- adding a stream 107
- adding and removing brokers 110
- amqsfmda.tst sample 100
- application
 - sample program 92
- Application Messaging Interface 32
- application programming 21
- applications, system management 141
- AppIdentityData parameter
 - publications forwarded by broker 31
- AppOriginData parameter
 - publications forwarded by broker 31
- authorization checks 41, 101

B

- backup 101
- broker
 - adding to a network 109
 - administration messages 141
 - backup 101
 - configuration parameters 102
 - controlling 107
 - deleting from a network 109
 - deregistering as a publisher 39
 - deregistering as subscriber 44
 - exit program 129
 - finding children 143
 - finding out about 154
 - finding parent 143
 - finding supported streams 143
 - interactions with subscriber and publisher 22
 - introduction 3
 - registering as a publisher 35
 - registering as a subscriber 41
 - response message 86
 - routing exit 129
 - setting up 99
 - stanza of qm.ini 102
- broker deleted message 142
- broker hierarchy, example 11
- broker networks 11
- broker queues, defining 99
- broker response message 86

C

- child broker 11
- children messages 143
- ChkPtActiveCount parameter 104
- ChkPtMsgSize parameter 103
- ChkPtRestartCount parameter 104
- class of service 11

- clear broker's memory, control command 114
- clrmqbrk command 114
- cluster queues 31, 99
- CodedCharSetId field
 - MQRFH structure 49
- Command field
 - MQCFH structure 144
- command message
 - name/value pairs 57
 - PCF format 141
 - RFH format 47
 - structure 21
- Command parameter
 - Broker response message 86
 - Delete Publication command 58
 - Deregister Publisher command 60
 - Deregister Subscriber command 62
 - Publish command 65
 - Register Publisher command 70
 - Register Subscriber command 72
 - Request Update command 80
- CompCode field
 - MQCFH structure 144
- CompCode parameter
 - Broker response message 86
- compiling, routing exit 137
- configuration file 102
- control commands
 - clear broker's memory (clrmqbrk) 114
 - deregister or delete broker function (dlmqbrk) 117
 - display broker status (dspmqbrk) 119
 - end broker function (endmqbrk) 121
 - migrate broker to WebSphere Business Integration Brokers function (migmqbrk) 123
 - start broker function (strmqbrk) 125
- Control field
 - MQCFH structure 144
- controlling brokers 107
- CorrelId parameter
 - message sent to broker 29
 - publications forwarded by broker 30
 - response messages 84
- creating queues 99

D

- data conversion 31
- data, publication 53
- dead-letter queue 101
- dead-letter queue processing 83
- defining queues 99
- Delete Publication command 58, 147
- DeleteOptions parameter
 - Broker response message 87
 - Delete Publication command 58
- deleting a broker from a network 109
- deleting a stream 108
- deleting publications 38
- deregister or delete broker function, control command 117
- Deregister Publisher command 60, 147
- Deregister Subscriber command 62, 147

- deregistering as a publisher 39
- deregistering as a subscriber 44
- DestinationQMGrName field
 - MQPXP structure 131
- DestinationQName field
 - MQPXP structure 131
- DestinationType field
 - MQPXP structure 131
- DiscardNonPersistentInputMsg parameter 105
- DiscardNonPersistentPublication parameter 106
- DiscardNonPersistentResponse parameter 105
- display broker status, control command 119
- DLQNonPersistentPublication parameter 106
- DLQNonPersistentResponse parameter 105
- dltmqbrk command 117
- double-byte character sets 53
- dspmqbrk command 119

E

- Encoding field
 - MQRFH structure 48
- end broker function, control command 121
- endmqbrk command 121
- error codes
 - Broker response message 88
 - Delete Publication command 58
 - Deregister Publisher command 61
 - Deregister Subscriber command 64
 - Publish command 69
 - Register Publisher command 71
 - Register Subscriber command 79
 - Request Update command 81
- error handling 83
- error response 85
- ErrorId parameter
 - Broker response message 87
- ErrorPos parameter
 - Broker response message 87
- event publications 14
- example
 - administration information program 157
 - application program 92
 - broker hierarchy 11
 - Broker response message 88
 - clrmqbrk command 115
 - Delete Publication command 58
 - Deregister Publisher command 61
 - Deregister Subscriber command 64
 - dltmqbrk command 118
 - dspmqbrk command 119
 - endmqbrk command 122
 - metatopic information 159
 - metatopic requests 153
 - migmqbrk command 124
 - multiple broker configuration 4
 - multiple subscriptions 12
 - NameValueString 51
 - propagation of publications 13
 - propagation of subscriptions 12
 - publication data 53
 - Publish command 69
 - qm.ini broker stanza 102
 - Register Publisher command 71
 - Register Subscriber command 78
 - Request Update command 81
 - routing exit 137

- example (*continued*)
 - simple broker configuration 4
 - strmqbrk command 126
- exit program 129
- ExitData field
 - MQPXP structure 131
- ExitId field
 - MQPXP structure 131
- ExitNumber field
 - MQPXP structure 132
- ExitParms parameter 129
- ExitReason field
 - MQPXP structure 132
- ExitResponse field
 - MQPXP structure 132
- ExitResponse2 field
 - MQPXP structure 133
- ExitUserArea field
 - MQPXP structure 133
- Expiry parameter
 - message sent to broker 30
 - publications forwarded by broker 30

F

- Feedback field
 - MQPXP structure 133
- Flags field
 - MQRFH structure 49
- Format field
 - MQRFH structure 49
- Format parameter
 - message sent to broker 29
 - publications forwarded by broker 30
 - response messages 84

G

- global publications
 - introduction 14
 - publishing 38
- group messages 31
- GroupId parameter 106

H

- header files 163
- HeaderLength field
 - MQPXP structure 134

I

- identity of publisher and subscriber 27
- identity of subscription 28
- initialization file 102
- IntegerData parameter
 - Publish command 65
- internal queues 101

J

- JmsStreamPrefix parameter 106

L

- limitations 6, 7, 31
- local publications
 - introduction 14
 - publishing 38

M

- managing brokers 107
- MaxMsgRetryCount parameter 102
- message descriptor (MQMD)
 - message sent to broker 29
 - publications forwarded by broker 30
 - response messages 84
- message flow 22
- message format
 - broker response 86
 - commands 47, 57
 - metatopic 154
- message order 26
- messages
 - broker administration 141
 - group 31
 - response 84
 - segmented 31
- metatopics 151
 - example 159
 - sample program 157
- migmqbrk command 123
- migrate broker to WebSphere Business Integration Brokers
 - function, control command 123
- MQ_PUBSUB_ROUTING_EXIT call 129
- MQBACK, routing exit 137
- MQCFH structure 143
- MQCFT_* values 143
- MQCMIT, routing exit 137
- MQDISC, routing exit 136
- MQFB_* values 134
- MQMD (message descriptor)
 - message sent to broker 29
 - publications forwarded by broker 30
 - response messages 84
- MQPXP structure 130
- MQPXP_* values 135
- MQRFH 47
- MQRFH_* values 48, 49
- MQRFH_DEFAULT 50
- MQXCC_* values 132
- MQXR_* values 132
- MQXUA_* values 133
- MsgDescPtr field
 - MQPXP structure 134
- MsgId parameter
 - response messages 85
- MsgInLength field
 - MQPXP structure 134
- MsgInPtr field
 - MQPXP structure 134
- MsgOutLength field
 - MQPXP structure 134
- MsgOutPtr field
 - MQPXP structure 134
- MsgSeqNumber field
 - MQCFH structure 144
- MsgType parameter
 - message sent to broker 29

- MsgType parameter (*continued*)
 - publications forwarded by broker 30
 - response messages 85
- multiple subscriptions, example 12

N

- name of subscription 28
- NameValueString 51
- NameValueString field 49
- network
 - adding a broker 109
 - broker 11
 - deleting a broker 109

O

- OK response 85
- OpenCacheExpiry parameter 103
- OpenCacheSize parameter 103

P

- ParameterCount field
 - MQCFH structure 144
- ParameterId parameter
 - Broker response message 87
- parent broker 11
- parent messages 143
- PCF definitions
 - command messages 146
 - Delete Publication 147
 - Deregister Publisher 147
 - Deregister Subscriber 147
 - Publish 148
 - Register Publisher 148
 - Register Subscriber 149
 - Request Update 150
- persistence 31
- Persistence parameter
 - publications forwarded by broker 30
 - response messages 85
- Priority parameter
 - publications forwarded by broker 30
 - response messages 85
- problem determination 88
- publication data 53
- publication propagation, example 13
- PublicationOptions parameter
 - Broker response message 87
 - Publish command 65
- publications
 - customizing 129
 - deleting 38
- Publish command 65, 148
- publish/subscribe
 - command messages 47, 57
 - exit structure 130
- PublishBatchInterval parameter 103
- PublishBatchSize parameter 103
- publisher
 - broker restart 37
 - changing registration 37
 - deregistering with the broker 39
 - exit program 129
 - identity 27

- publisher (*continued*)
 - interactions with subscriber and broker 22
 - introduction 3
 - registering with the broker 35
 - writing applications 35
- publisher information messages 151
- publishing information 37
- PublishTimestamp parameter
 - Publish command 66
- PutApplName parameter
 - publications forwarded by broker 31
 - response messages 85
- PutApplType
 - publications forwarded by broker 31
- PutApplType parameter
 - response messages 85
- PutDate parameter
 - publications forwarded by broker 31
- PutTime parameter
 - publications forwarded by broker 31

Q

- qm.ini 102
- QMGrName field
 - MQPXP structure 135
- QMGrName parameter
 - Broker response message 87
 - Deregister Publisher command 60
 - Deregister Subscriber command 62
 - Publish command 67
 - Register Publisher command 70
 - Register Subscriber command 72
 - Request Update command 80
- QName parameter
 - Broker response message 87
 - Deregister Publisher command 60
 - Deregister Subscriber command 62
 - Publish command 67
 - Register Publisher command 70
 - Register Subscriber command 72
 - Request Update command 80
- queue manager initialization file 102
- queues
 - cluster 31
 - dead letter 101
 - internal 101
 - stream 100
 - SYSTEM.BROKER.ADMIN.STREAM 100
 - SYSTEM.BROKER.CONTROL.QUEUE 99
 - SYSTEM.BROKER.DEFAULT.STREAM 99
 - SYSTEM.BROKER.MODEL.STREAM 100

R

- reason codes
 - PCF messages 145
- Reason field
 - MQCFH structure 144
- Reason parameter
 - Broker response message 86
- ReasonText parameter
 - Broker response message 86
- Register Publisher command 70, 148
- Register Subscriber command 72, 149
- registering as a publisher 35

- registering as a subscriber 41
- registration
 - changing for a publisher 37
 - changing for subscriber 43
- RegistrationOptions parameter
 - Broker response message 87
 - Deregister Publisher command 60
 - Deregister Subscriber command 62
 - Publish command 67
 - Register Publisher command 70
 - Register Subscriber command 72
 - Request Update command 80
- ReplyToQ parameter
 - message sent to broker 30
 - publications forwarded by broker 30
- ReplyToQMGr parameter
 - message sent to broker 30
 - publications forwarded by broker 30
- Report parameter
 - message sent to broker 29
 - publications forwarded by broker 30
 - response messages 85
- request update
 - message flow 23
- Request Update command 80, 150
- requesting information 43
- response messages 84
- retained publication
 - introduction 14
 - publishing 38
- return codes
 - clrmqbrk command 115
 - dltmqbrk command 118
 - dspmqbrk command 119
 - endmqbrk command 122
 - migmqbrk command 124
 - strmqbrk command 126
- RFH definitions
 - Delete Publication 58
 - Deregister Publisher 60
 - Deregister Subscriber 62
 - Publish 65
 - Register Publisher 70
 - Register Subscriber 72
 - Request Update 80
- root broker 11
- routing exit 129
- RoutingExitAuthorityCheck parameter 104
- RoutingExitConnectType parameter 104
- RoutingExitData parameter 105
- RoutingExitPath parameter 104
- rules and formatting header
 - definition 47
 - use of 51

S

- sample program
 - administration information 157
 - application 92
 - Application Messaging Interface 96
 - routing exit 137
- security, setting up 101
- segmented messages 31
- SequenceNumber parameter
 - Publish command 68
- start broker function, control command 125

- starting a broker 107
- state publications 14
- stopping a broker 107
- stream
 - adding 107
 - deleting 108
 - finding which are supported 143
 - implementation 10
 - introduction 3
 - reasons for using 10
- stream deleted message 142
- stream queues 100
- stream support messages 143
- StreamName field
 - MQPXP structure 135
- StreamName parameter
 - Broker response message 88
 - Delete Publication command 58
 - Deregister Publisher command 60
 - Deregister Subscriber command 63
 - Publish command 68
 - Register Publisher command 71
 - Register Subscriber command 77
 - Request Update command 81
- StreamsPerProcess parameter 102
- StringData parameter
 - Publish command 68
- strmqbrk command 125
- StrucId field
 - MQPXP structure 135
 - MQRFH structure 48
- StrucLength field
 - MQCFH structure 143
 - MQRFH structure 48
- structures
 - MQCFH 143
 - MQPXP 130
 - MQRFH 47
- SubIdentity parameter
 - Deregister Subscriber command 63
 - Register Subscriber command 77
- SubName parameter
 - Deregister Subscriber command 63
 - Register Subscriber command 78
 - Request Update command 81
- subscriber
 - broker restart 43
 - changing registration 43
 - deregistering with the broker 44
 - identity 27
 - interactions with publisher and broker 22
 - introduction 3
 - message arrival order 26
 - registering with the broker 41
 - writing applications 41
- subscriber information messages 151
- subscribing to metatopics 152
- subscription
 - identity 28
 - name 28
- subscription deregistered message 142
- subscription propagation, example 12
- subscriptions
 - passing between brokers 12
- SubUserData parameter
 - Register Subscriber command 78
- SyncPointIfPersistent parameter 105

- system design 9
- system management programs 141
- SYSTEM.BROKER.ADMIN.STREAM 100
- SYSTEM.BROKER.CONTROL.QUEUE 99
- SYSTEM.BROKER.DEFAULT.STREAM 99
- SYSTEM.BROKER.MODEL.STREAM 100

T

- threads, routing exit 136
- Topic parameter
 - Broker response message 88
 - Delete Publication command 58
 - Deregister Publisher command 61
 - Deregister Subscriber command 64
 - Publish command 65
 - Register Publisher command 70
 - Register Subscriber command 72
 - Request Update command 80
- topics
 - introduction 3
 - using wildcards 9
- triggering a broker 107
- Type field
 - MQCFH structure 143

U

- unit of work 31
- UserId parameter
 - Broker response message 88
 - publications forwarded by broker 30

V

- Version field
 - MQCFH structure 143
 - MQPXP structure 135
 - MQRFH structure 48

W

- Warning response 85
- WebSphere Business Integration Brokers, relationship with 7
- WebSphere MQ, relationship with 6
- wild cards
 - using with metatopics 153
- wildcards 9
- writing applications 21

Sending your comments to IBM

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 make comments about the functions of IBM products or systems, 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, to this address:
User Technologies Department (MP095)
IBM United Kingdom Laboratories
Hursley Park
WINCHESTER,
Hampshire
SO21 2JN
United Kingdom
- By fax:
 - From outside the U.K., after your international access code use 44-1962-816151
 - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
 - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
 - IBMLink: HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.



SC34-6606-00



Spine information:



WebSphere MQ

WebSphere MQ Publish/Subscribe User's Guide

Version 6.0