WebSphere MQ

Publish/Subscribe User's Guide

Version 7.0

SC34-6950-00

WebSphere MQ

Publish/Subscribe User's Guide

Version 7.0

SC34-6950-00

Note
Before using this information and the product it supports, be sure to read the general information under notices at the back of this book.

First edition (April 2008)

This edition of the book applies to the following products:

- IBM WebSphere MQ for Windows, Version 7.0

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

Contents

Figures

Tables

Chapter 1. What's new in publish/subscribe in WebSphere MQ Version 7.0?

Publish/subscribe has been changed significantly for WebSphere® MQ Version 7.0. In previous versions, publish/subscribe messaging was controlled using a command message interface. This interface no longer exists in Version 7.0 publish/subscribe. Instead, publish/subscribe messaging is now controlled using new function in the WebSphere MQ API and as a result, publish/subscribe messaging is much more consistent with point-to-point messaging. This new way of doing publish/subscribe messaging is documented in the main body of this book.

Applications written using previous versions of WebSphere MQ publish/subscribe and make use of the command message interface are encouraged to move to the new WebSphere MQ publish/subscribe API – however, the command message interface continues to be supported by means of a process which runs on all platforms (including z/OS®). As such, if you are already a user of publish/subscribe you can continue to use your current configuration after installing WebSphere MQ Version 7.0 without making extensive changes to your applications or configuration.

Similarly, JMS applications do not have to be modified, although if you do not chose to use the new Version 7.0 publish/subscribe you will not benefit from the simplified administration that is now available when using WebSphere MQ as the provider. Since the command message interface method of doing publish/subscribe is still supported in Version 7.0 using the PSMODE function, this interface continues to be documented in the WebSphere MQ Version 7.0 library. This information is grouped together here Chapter 7, "Publish/subscribe deprecated function," on page 81.

Benefits of WebSphere Version 7.0 publish/subscribe

Publish/subscribe messaging is now performed using the WebSphere MQ API, there are a number of benefits of using this method.

Benefits of the new method of publish/subscribe include:

- In WebSphere MQ Version 7.0, support has been added for publish/subscribe messaging on z/OS.
- To perform some WebSphere MQ publish/subscribe functions in previous versions you required WebSphere Event Broker, WebSphere Message Broker or the MA0C WebSphere MQ SupportPac™ (if you were using WebSphere MQ Version 5.3), none of these applications are required now unless you want to route messages according to their content, in which case you can use WebSphere MQ in combination with WebSphere Message Broker.
- In WebSphere MQ Version 6.0, if you were using WebSphere MQ publish/subscribe you had to use PCFs or RFH2 headers, this is no longer the case.
- In WebSphere MQ Version 6.0 if you were using WebSphere Event Broker or WebSphere Message Broker you had to use RFH1 headers, this is no longer the case.

- In WebSphere MQ Version 7.0, the way you publish and subscribe is consistent with the rest of the WebSphere MQ API, making publish/subscribe more intuitive and easier to use.
- Native support for non-durable subscriptions has been added, allowing unconsumed messages and unnecessary subscriptions to be cleaned up at disconnection. This removes the need that existed in WebSphere MQ Version 6.0 for JMS to tidy up non-durable subscriptions in order to meet JMS specification requirements.
- By using non-durable subscriptions with JMS publish/subscribe messaging performance can be improved and resource usage made more efficient.
- The interlock between JMS and the queue manager is improved by using the new WebSphere MQ publish/subscribe API (rather than a command message interface).

Chapter 2. Introduction to WebSphere MQ publish/subscribe messaging

Publish/subscribe messaging allows you to decouple the provider of information, from the consumers of that information. The sending application and receiving application do not need to know anything about each other for the information to be sent and received.

Before a point-to-point WebSphere MQ application can send a message 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 the sending application has to do, is put a WebSphere MQ message, containing the information that it wants, and assign it a topic, that denotes the subject of the information, and let WebSphere MQ handle the distribution of that information. Similarly, the target application does not have to know anything about the source of the information it receives.

Figure 1 shows the simplest publish/subscribe system. There is one publisher, one queue manager, and one subscriber. A subscription is sent from the subscriber to the queue manager, a publication is sent from the publisher to the queue manager, and the publication is then forwarded by the queue manager to the subscriber.

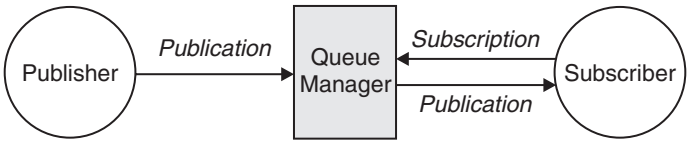


Figure 1. Simple publish/subscribe configuration

A typical publish/subscribe system has more than one publisher and more than one subscriber, and often, more than one queue manager. An application can be both a publisher and a subscriber.

Overview of publish/subscribe components

Publish/subscribe is the mechanism by which subscribers can receive information, in the form of messages, from publishers. The interactions between publishers and subscribers are controlled by queue managers, using standard WebSphere MQ facilities.

A typical publish/subscribe system has more than one publisher and more than one subscriber, and often, more than one queue manager. An application can be both a publisher and a subscriber.

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

interested in that information. Publishers generate this information in the form of messages, called *publications* that they want to publish and define the topic of these messages.

The consumer of the information is called a *subscriber*. Subscribers create *subscriptions* that describe the topic that the subscriber is interested in. Thus, the subscription determines which publications are forwarded to the subscriber. Subscribers can have make multiple subscriptions and can receive information from many different publishers.

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

It is the existence of topics that allows the providers and consumers of information to be decoupled in publish/subscribe messaging by removing the need to include a specific destination in each message as is required in point-to-point messaging.

Interactions between publishers and subscribers are all controlled by a queue manager. The queue manager receives messages from publishers, and subscriptions from subscribers (to a range of topics). The queue manager's job is to route the published messages to the subscribers that have registered an interest in the topic of the messages.

Standard WebSphere MQ facilities are used to distribute messages, 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 queue manager publish/subscribe configuration

Figure 2 on page 5 illustrates a basic single queue manager publish/subscribe configuration. The example shows the configuration for a news service, where information is available from publishers about several topics:

- 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 queue manager 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.

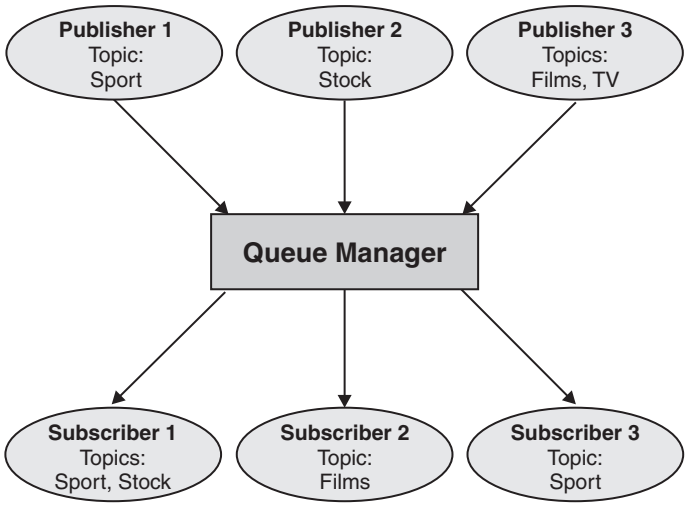


Figure 2. Single queue manager publish/subscribe example. This shows the relationship between publishers, subscribers, and queue managers.

Publishers and publications

In WebSphere MQ publish/subscribe a publisher is an application that makes information about a specified topic available to a queue manager in the form of a standard WebSphere MQ message called a publication. A publisher can publish information about more than one topic.

Publishers use the MQPUT verb to put a message to a previously opened topic, this message is a publication. The local queue manager then routes the publication to any subscribers who have subscriptions to the topic of the publication. A published message can be consumed by more than one subscriber.

In addition to distributing publications to all local subscribers that have appropriate subscriptions, a queue manager can also distribute the publication to any other queue managers connected to it, either directly or through a network of queue managers that have subscribers to the topic.

In a WebSphere MQ publish/subscribe network, a publishing application can also be a subscriber.

State and event information

Publications can be categorized as either state publications, such as the current price of a stock, or event publications, such as a trade in that stock.

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.

If a publication contains state information, it is often published as a retained publication. A new subscriber typically wants the current state information immediately; the subscriber does not want to wait for an event that causes the information to be republished. Subscribers will automatically receive a topic's retained publication when it subscribes unless the subscriber uses the MQSO_PUBLICATIONS_ON_REQUEST or MQSO_NEW_PUBLICATIONS_ONLY options.

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, after a publication is sent to all interested subscribers it is discarded. However, a publisher can specify that a copy of a publication should be retained so that it can be sent to future subscribers who register an interest in the topic.

Deleting publications after they have been sent to all interested subscribers is suitable for event information, but is not always suitable for state information. By retaining a message, new subscribers do not have to wait for information to be published again before they receive initial state information. For example, a subscriber with a subscription to a stock price would receive the current price straight away, without waiting for the stock price to change (and hence be re-published).

The queue manager can retain only one publication for each topic, so a topic's existing retained publication is deleted when a new retained publication arrives at the queue manager. Wherever possible, have no more than one publisher sending retained publications on any topic.

Subscribers can specify that they do not want to receive retained publications by using the MQSO_NEW_PUBLICATIONS_ONLY subscription option. Existing subscribers can ask for duplicate copies of retained publications to be sent to them.

There are times when you might not want to retain publications, even for state information:

- If all subscriptions to a topic are made before any publications are made on that topic, and you do not expect, or will not allow, new subscriptions, there is no need to retain publications because they will be delivered to the complete set of subscribers the first time they are published.
- If publications occur very frequently, such as every second, a new subscriber (or a subscriber recovering from a failure) receives the current state almost immediately after their initial subscription, so there is no need to retain these publications.
- If the publications are quite large, you could end up needing a considerable amount of storage space to store the retained publication for each topic. In a multiple queue manager environment, retained publications are stored by all queue managers in the network that have a matching subscription.

When deciding whether to use retained publications, consider how subscribing applications recover from a failure. If the publisher does not use retained publications, the subscriber application might need to store its current state locally.

To ensure that a publication is retained use the MQPMO_RETAIN put-message option. If this option is used and the publication cannot be retained, the message will not be published and the call will fail with MQRC_PUT_NOT_RETAINED.

If a message is a retained publication this will be indicated by the MQIsRetained message property.

Subscribers and subscriptions

In WebSphere MQ publish/subscribe, a subscriber is an application that requests information about a specific topic from a queue manager in a publish/subscribe network. A subscriber can receive messages, about the same or different topics, from more than one publisher.

Subscriptions can be created manually using an MQSC command or by applications. These subscriptions are issued to the local queue manager and contain information about the publications the subscriber wants to receive:

- The topic the subscriber is interested in; this can resolve to multiple topics if wildcards are used.
- An optional selection string to be applied to published messages.
- A handle to the a queue (known as the *subscriber queue*), on which selected publications should be placed, and optional CorrelId.

The local queue manager stores subscription information and when it receives a publication, scans the information to determine whether there is a subscription that matches the publication's topic and selection string. For each matching subscription, the queue manager directs the publication to the subscriber's subscriber queue. The information that a queue manager stores about subscriptions can be viewed by using the DIS SBSTATUS command.

A subscription is deleted only when one of the following events occurs:

- The subscriber unsubscribes using the MQCLOSE call (if the subscription was made non-durably).
- The subscription expires.
- The subscription is deleted by the system administrator using the DELETE SUB command.
- The subscriber application ends (if the subscription was made non-durably).
- The queue manager is stopped or restarted (if the subscription was made non-durably).

Managed queues and publish/subscribe

When you create a subscription you can choose to use managed queuing. If you use managed queueing a subscription queue is automatically created when you create a subscription. Managed queues are tidied up automatically in accordance with the durability of the subscription. Using managed queues means that you do not have to worry about creating queues to receive publications and any unconsumed publications are removed from subscriber queues automatically if a non-durable subscription connection is closed.

If an application has no need to use a particular queue as its subscriber queue, the destination for the publications it receives, it can make use of the *managed subscriptions* using the MQSO_MANAGED subscription option. If you create a managed subscription, the queue manager returns an object handle to the subscriber for a subscriber queue that the queue manager creates where publications will be received. The queue's object handle will be returned allowing you to browse, get or inquire on the queue (it is not possible to put to or set attributes of a managed queue unless you have been explicitly given access to temporary dynamic queues).

The durability of the subscription determines whether the managed queue remains when the subscribing application's connection to the queue manager is broken.

Managed subscriptions are particularly useful when used with non-durable subscriptions because when the application's connection is ended, unconsumed messages would otherwise remain on the subscriber queue taking up space in your queue manager indefinitely. If you are using a managed subscription, the managed queue will be a temporary dynamic queue and as such will be deleted along with any unconsumed messages when the connection is broken for any of the following reasons:

- MQCLOSE with MQCO_REMOVE_SUB is used and the managed Hobj is closed.
- a connection is lost to an application using a non-durable subscription (MQSO_NON_DURABLE).
- a subscription is removed because it has expired and the managed Hobj is closed.

Managed subscriptions can also be used with durable subscriptions but it is possible that you would want to leave unconsumed messages on the subscriber queue so that they can be retrieved when the connection is reopened. For this reason, managed queues for durable subscriptions take the form of a permanent dynamic queue and will remain when the subscribing application's connection to the queue manager is broken.

You can set an expiry on your subscription if you want to use permanent dynamic managed queue so that although the queue will still exist after the connection is broken, it will not continue to exist indefinitely.

If you delete the managed queue you will receive an error message.

The managed queues that are created are named with numbers at the end (timestamps) so that each is unique.

Subscription durability

Subscriptions can be configured to be durable or non-durable. Subscription durability determines what happens to subscriptions when subscribing applications disconnect for a queue manager.

Durable subscriptions

Durable subscriptions continue to exist when a subscribing application's connection to the queue manager is closed. If a subscription is durable, when the subscribing application disconnects, the subscription remains in place and can be used by the subscribing application when it reconnects requesting the subscription again using the SubName that was returned when the subscription was created.

When subscribing durably, a subscription name (SubName) is required. Subscription names must be unique within a queue manager so that it can be used to identify a subscription. This means of identification is necessary when specifying a subscription you want to resume, if you have either deliberately closed the handle to the subscription (using the MQCO_KEEP_SUB option) or have been disconnected from the queue manager. You can resume an existing subscription by using the MQSUB call with the MQSO_RESUME option. Subscription names are also displayed if you use the DISPLAY SBSTATUS command with SUBTYPE ALL or ADMIN.

When an application no longer requires a durable subscription it can be removed using the MQCLOSE command with the MQCO_REMOVE_SUB option or it can be deleted manually use the MQSC DELETE SUB call.

Whether durable subscriptions can be made to a topic can be controlled using the **DURSUB** topic attribute.

On return from an MQSUB call using the MQSO_RESUME option, subscription expiry will be set to the original expiry of the subscription and not the remaining expiry time.

A queue manager will continue to send publications to satisfy a durable subscription even if that subscriber application is not connected. This will lead to a build up of messages on the subscriber queue. The easiest way to avoid this problem is to use a non-durable subscription wherever appropriate. However, where it is necessary to use durable subscriptions, a build up of messages can be avoided if the subscriber subscribes using the MQSO_PUBLICATIONS_ON_REQUEST option. A subscriber can then control when it receives publications by using the MQSUBRQ call.

Non-durable subscriptions

Non-durable subscriptions exist only as long as the subscribing application's connection to the queue manager remains open. The subscription is removed when the subscribing application disconnects from the queue manager either deliberately or by loss of connection. When the connection is closed, the information about the subscription is removed from the queue manager, and will no longer be shown if you display subscriptions using the DISPLAY SBSTATUS command. No more messages will be put to the subscriber queue.

What happens to any unconsumed publications on the subscriber queue for non-durable subscriptions is determined as follows.

- If a subscribing application is using a managed destination, any publications that have not been consumed are automatically removed.
- If the subscribing application provides a handle to its own subscriber queue when it subscribes, unconsumed messages are not removed automatically. It is the responsibility of the application to clear the queue if that is appropriate. If the queue is shared by more than one subscriber, or other point-to-point applications, it might not be appropriate to clear the queue completely.

Although not required for non durable subscriptions, a subscription name if provided, will be used by the queue manager. Subscription names must be unique within the queue manager so that it can be used to identify a subscription.

Selection strings

A *selection string* is an expression that is applied to a publication to determine whether it matches a subscription. Selection strings can include wildcard characters.

When you subscribe, in addition to specifying a topic, you can specify a selection string to select publications according to their message properties.

WebSphere MQ topics

A topic is a character string that describes the subject of the information that is published in a publish/subscribe message.

Topics are key to the successful delivery of messages in a publish/subscribe system. Instead of including a specific destination address in each message, a publisher assigns a topic to each message. The queue manager matches the topic with a list of subscribers who have subscribed to that topic, and delivers the message to each of those subscribers.

Note that a publisher can control which subscribers receive a publication by choosing carefully the topic that is specified in the message.

Topics can be defined using MQSC or PCF commands. However, the topic of a message does not have to be defined before a publisher can use it; a topic is created when it is specified in a publication or subscription for the first time.

A topic string can include any character from the Unicode character set, including the space character. However, there are three characters that have special meanings. These characters (″/″, ″#″, and ″+″) are described in “Special characters in topics” on page 11.

Although a null character does not cause an error, do not use null characters in your topic strings.

Topic strings are case sensitive.

Topic names

The full topic name is created by the concatenation of two fields used in publish/subscribe MQI calls, in the order listed.

1. The value of the TOPICSTR parameter of the topic object named in *ObjectName* field.
2. The value of the *ObjectString* field, if the *VSLength* provided for that variable length string is non-zero

A ’/’ character is inserted between the two elements in the resultant combined topic name.

These fields are considered to be completed if the first character of the field is neither a blank nor a null character. If only one of the fields is completed, it is used unchanged as the topic name. If neither field has a value the call fails with reason code MQRC_UNKNOWN_OBJECT_NAME.

Table 1 shows examples of topic string concatenation:

Table 1. Topic string concatenation examples

TOPICSTR	ObjectString	Full topic name	Comment
/Football	Scores	/Football/Scores	A '/' character is added at the concatenation point
/Football/	Scores	/Football//Scores	An 'empty node' is produced between the two strings
/Football	/Scores	/Football//Scores	An 'empty node' is produced between the two strings
/Football/	/Scores	/Football///Scores	Two 'empty nodes' are produced between the two strings

Notes:

1. The '/' character is considered to be a special character providing structure to the full topic name in the topic tree and should not be used for any other reason as the structure of the topic tree will not be effected. This means that the topic '/Football' is not the same as the topic 'Football'. However, topic '/Football' is the same as the topic '/Football/'.
2. A full topic name with two repeated '/' characters is not valid.
3. If the full topic name is not valid, the call fails with reason code MQRC_TOPIC_STRING_ERROR.
4. Wild card characters, +, #, * and ? are special characters. Do not use these characters in your topic strings when publishing. They are not considered invalid, however, if using them is unavoidable you should take care to understand the behavior when using them.
   - Publishing on a topic string with # or + mixed in with other characters (including themselves) within a topic level can be subscribed on, with either wildcard scheme.
   - Publishing on a topic string with # or + as the only character between two '/' characters will produce a topic string that cannot be subscribed on explicitly by an application using the wildcard scheme MQSO_WILDCARD_TOPIC. This will result in the application getting more publications than expected.
   - Publishing on a topic string containing either * or ? anywhere will produce a topic string that cannot be subscribed on explicitly by an application using the wildcard scheme MQSO_WILDCARD_CHAR. This will result in the application getting more publications than expected.

Special characters in topics

WebSphere MQ supports two different wildcard schemas. Wildcard characters are determined differently according to the schema the subscription uses. This topic details the wildcards used in the Version 7.0 implementation of publish/subscribe messaging.

Topics that were created prior to WebSphere MQ Version 7.0 use the schema described in "Version 6 wild card schema" on page 84.

A topic can contain any character in the Unicode character set. However, the following three characters have a special meaning in the Version 7.0 publish/subscribe:

The topic level separator "/".
The multilevel wildcard "#".
The single-level wildcard "+".

The topic level separator is used to introduce structure into the topic, and can therefore be specified within the topic for that purpose.

Wildcards are a powerful feature of the topic system in WebSphere MQ publish/subscribe. Wildcards allow subscribers to subscribe to more than one topic at a time. The multilevel wildcard and single level wildcard can be used for subscriptions, but they cannot be used within a topic by the publisher of a message.

However, if a publisher uses the characters "+" or "#" together with other characters in any topic level within a topic, these characters are not treated as wildcards, and they do not have any special meaning.

The topic level separator

WebSphere MQ publish/subscribe does not recognize that the '/' character is being used in a special way. However, it is allows you to introduce structure into the topic, providing a hierarchical structure to the topic string. It can be used by applications to separate levels within a topic tree.

The use of the topic level separator is significant when the two wildcard characters are encountered in topics specified by subscribers.

Topic hierarchy is important in the administration of access control.

The multilevel wildcard

The multilevel wildcard can be used for subscriptions, but cannot be used within a topic string by the publisher of a message.

The multilevel wildcard character "#" is used to match any number of levels within a topic. For example, using the example topic tree shown above, if you subscribe to "USA/Alaska/#", you receive messages on topics "USA/Alaska" and "USA/Alaska/Juneau".

The multilevel wildcard can represent zero or more levels. Therefore, "USA/#" can also match the singular "USA", where # represents zero levels. The topic level separator is meaningless in this context, because there are no levels to separate.

The multilevel wildcard is only effective when specified on its own or next to the topic level separator character. Therefore, "#" and "USA/#" are valid topics where the "#" character is treated as a wildcard. However, although "USA#" is also a valid topic string, the "#" character is not regarded as a wildcard and does not have any special meaning. See "When wildcards are not wild" on page 13 for more information.

The single-level wildcard

The multilevel wildcard can be used for subscriptions, but cannot be used within a topic string by the publisher of a message.

The single-level wildcard character ″+″ matches one, and only one, topic level. For example, ″USA/+″ matches ″USA/Alabama″, but not ″USA/Alabama/Auburn″. Because the single-level wildcard matches only a single level, ″USA/+″ does not match ″USA″.

The single-level wildcard can be used at any level in the topic tree, and in conjunction with the multilevel wildcard. The single-level wildcard must be specified next to the topic level separator, except when it is specified on its own. Therefore, ″+″ and ″USA/+″ are valid topics where the ″+″ character is treated as a wildcard. However, although ″USA+″ is also a valid topic string, the ″+″ character is not regarded as a wildcard and does not have any special meaning. See “When wildcards are not wild” for more information.

When wildcards are not wild

The wildcard characters ″+″ and ″#″ have no special meaning when they are mixed with other characters (including themselves) in a topic level.

This means that topics that contain ″+″ or ″#″ together with other characters in a topic level can be published.

For example, consider the following two topics:

1. level0/level1/+/level4/#
2. level0/level1/#+/level4/level#

In the first example, the characters ″+″ and ″#″ are treated as wildcards and are therefore not valid in a topic string that is to be published to but are valid in a subscription.

In the second example, the characters ″+″ and ″#″ are not treated as wildcards and therefore the topic string can be both published and subscribed to.

Topic trees

Each topic that you define is an element, or node, in the topic tree. The topic tree can either be empty to start with or contain topics that have been defined previously using MQSC or PCF commands. You can define a new topic either by using the create topic commands or by specifying the topic for the first time in a publication or subscription.

Although you can use any character string to define a topic’s topic string, it is advisable to choose a topic string that fits into a hierarchical tree structure. Thoughtful design of topic stings and topic trees can help you with the following operations:

- Subscribing to multiple topics.
- Establishing security policies.

Although you can construct a topic tree as a flat, linear structure, it is better to build a topic tree in a hierarchical structure with one or more root topics.

Figure 3 on page 14 shows an example of a topic tree with one root topic.

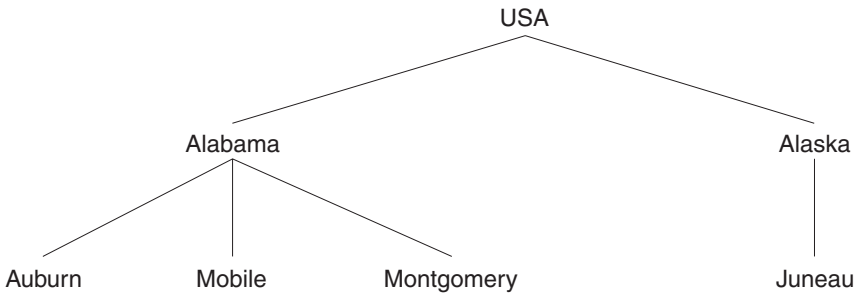


Figure 3. Example of a topic tree

Each character string in the figure represents a node in the topic tree. A complete topic string is created by aggregating nodes from one or more levels in the topic tree. Levels are separated by the "/" character. The format of a fully specified topic string is: "root/level2/level3".

The valid topics in the topic tree shown in Figure 3 are:
"USA"
"USA/Alabama"
"USA/Alaska"
"USA/Alabama/Auburn"
"USA/Alabama/Mobile"
"USA/Alabama/Montgomery"
"USA/Alaska/Juneau"

When you design topic strings and topic trees, remember that the queue manager does not interpret, or attempt to derive meaning from, the topic string itself. It simply uses the topic string to send selected messages to subscribers of that topic.

The following principles apply to the construction and content of a topic tree:

- There is no limit to the number of levels in a topic tree.
- There is no limit to the length of the name of a level in a topic tree.
- There can be any number of "root" nodes; that is, there can be any number of topic trees.

Administrative topic objects

An *administrative topic object* is a WebSphere MQ object that allows you to assign specific, non-default attributes to *topics*.

Figure 4 shows how a high-level topic of 'Sport' divided into separate topics covering different sports can be visualized as a topic tree:

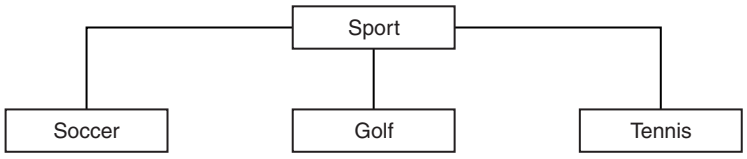


Figure 4. Visualization of a topic tree

Figure 5 on page 15 shows how the topic tree can be divided further, to separate different types of information about each sport:

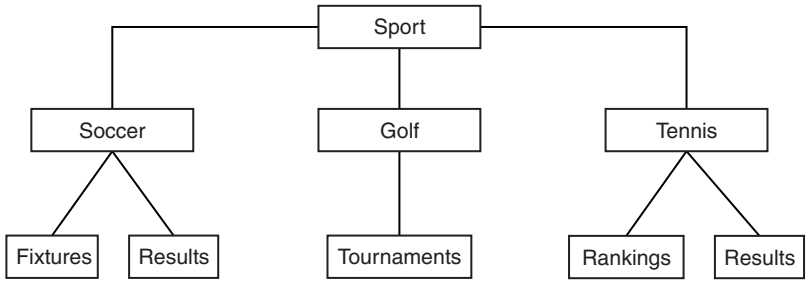


Figure 5. Extended topic tree

To create the topic tree illustrated, no administrative topic objects need be defined. If each of the nodes in this tree are defined by a topic string created in a publish or subscribe operation, each topic in the tree inherits its attributes from its parent. Attributes are inherited from the parent topic object because by default all attributes are set to ASPARENT. In this example, therefore, every topic has the same attributes as the 'Sport' topic, which again, assuming no administrative topic object exists for this node, inherits its attributes from SYSTEM.BASE.TOPIC.

Administrative topic objects can be used to define specific attributes for particular nodes in the topic tree. In the following example, the administrative topic object is defined to set the durable subscriptions attribute (DURSUB) of the soccer topic to NO:

```
DEFINE TOPIC(FOOTBALL.EUROPEAN)
       TOPICSTR('Sport/Soccer')
       DURSUB(NO)
       DESCR('Administrative topic object to disallow durable subscriptions')
```

The topic tree can now be visualized as:

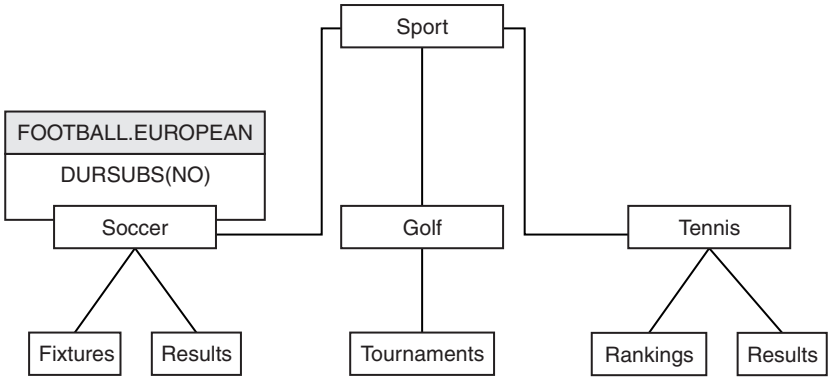


Figure 6. Visualization of an administrative topic object associated with the Sport/Soccer topic

Any applications subscribing to topics beneath Soccer in the tree can still use the topic strings they used before the administrative topic object was added. However, an application can now be written to subscribe using the object name FOOTBALL.EUROPEAN, instead of the string /Sport/Soccer. For example, to subscribe to /Sport/Soccer/Results, an application can specify MQSD.ObjectName as FOOTBALL.EUROPEAN and MQSD.ObjectString as Results.

This feature allows you to hide part of the topic tree from application developers. If you define an administrative topic object at a particular node in the topic tree, application developers can define their own topics below this, without needing to have knowledge of topics above the administrative topic object.

Inheriting attributes

If a topic tree has many administrative topic objects, each administrative topic object, by default, inherits its attributes from its closest parent administrative topic node. The previous example has been extended in Figure 7:

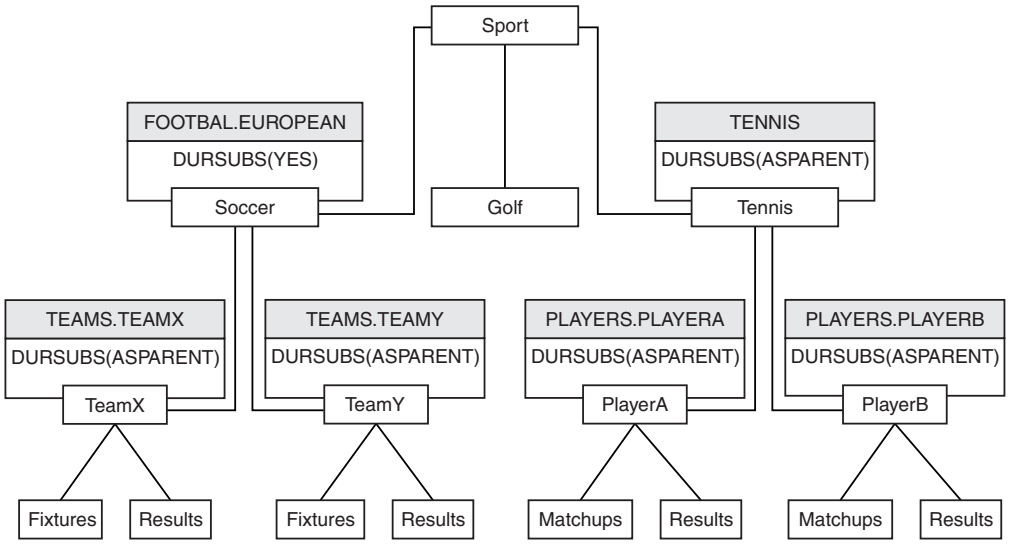


Figure 7. Topic tree with several administrative topic objects

If all topics at and below `/Sport/Soccer` need to have the attribute `DURSUB` set to `NO`, the only change that needs to be made is to alter the `DURSUB` attribute of `FOOTBALL.EUROPEAN` to `NO`.

This attribute can be set using the following command:

```
ALTER TOPIC(FOOTBALL.EUROPEAN) DURSUB(NO)
```

Because all the administrative topic objects below `Sport/Soccer` have the `DURSUB` attribute set to the default value `ASPARENT`, all topics below `Sport/Soccer` will inherit the value `NO` for their `DURSUB` attribute.

All the administrative topic objects at and below `Sport/Tennis` have the value `ASPARENT` for the attribute `DURSUB`. All topics at and below `Sport/Tennis`, therefore, will inherit `DURSUB` from the `SYSTEM.BASE.TOPIC` object and will have the value of `YES`.

Trying to make a durable subscription to the topic `Sport/Soccer/TeamX/Results` would now fail; however, trying to make a durable subscription to `Sport/Tennis/PlayerB/Results` would succeed.

SYSTEM.BASE.TOPIC

Base topic for ASPARENT resolution. If a particular topic has no parent administrative topic objects, or those parent objects also have ASPARENT, any remaining ASPARENT attributes are inherited from this object.

The default values of the SYSTEM.BASE.TOPIC are:

Table 2. Default values of SYSTEM.BASE.TOPIC

Parameter	**Value**
TOPICSTR	″
DEFPRTY	0
DEFPRESP	SYNC
DEFPSIST	NO
DESCR	'Base topic for resolving attributes'
DURSUB	YES
MDURMDL	SYSTEM.DURABLE.MODEL.QUEUE
MNDURMDL	SYSTEM.NDURABLE.MODEL.QUEUE
MASTER	YES
NPMSGDLV	ALLAVAIL
PMSGDLV	ALLDUR
PUB	ENABLE
SUB	ENABLE

If this object does not exist, its default values are still used by WebSphere MQ for ASPARENT attributes that are not resolved by parent topics further up the topic tree.

Chapter 3. Distributed publish/subscribe

This section discusses how publish/subscribe messaging can be performed between queue managers, and the 2 different queue manager topologies that can be used to connect queue managers, clusters and hierarchies.

Queue managers can communicate with other queue managers in your WebSphere MQ publish/subscribe system, so that subscribers can subscribe to one queue manager and receive messages that were initially published to another queue manager. This is illustrated in Figure 8.

Figure 8 shows a publish/subscribe system with two queue managers.

- Queue manager 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 queue manager, 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 queue manager from the publisher. This is possible because the queue managers are linked to each other.

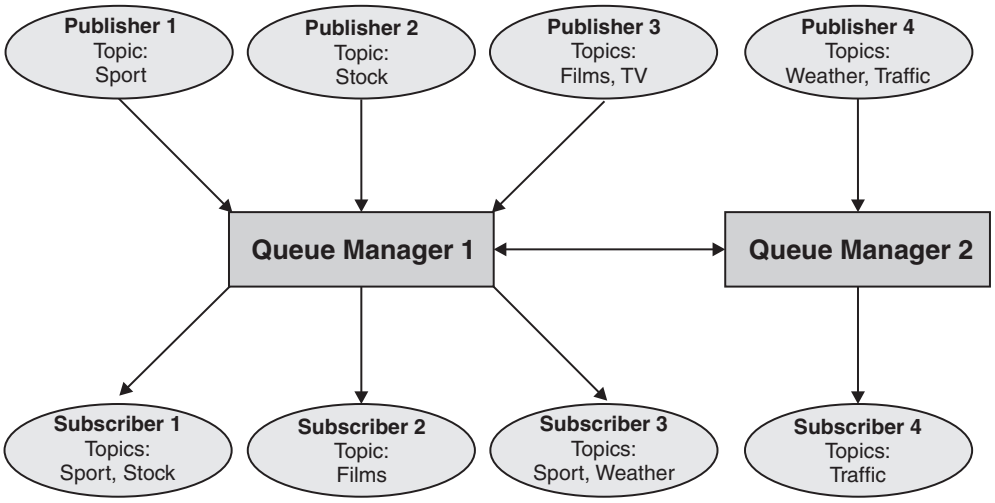


Figure 8. Publish/subscribe example with two queue managers

How does distributed publish/subscribe work?

WebSphere MQ publish/subscribe uses proxy subscriptions to ensure that subscribers can receive messages that are published to remote queue managers.

Distributed publish/subscribe uses the same components as distributed queuing to connect networks of queue managers and consequently, the applications that connect to those queue managers. To find out more about messaging between queue managers and the components involved making connections between queue managers see the *Intercommunication* documentation.

Subscribers need not do anything beyond the standard subscription operation in a distributed publish/subscribe system. When a subscription is made on a queue manager, the queue manager manages the process by which the subscription is propagated to connected queue managers. A subscriptions flows to all queue managers in the network, where proxy subscriptions are created to ensure that publications get routed back to the queue manager where the subscription was created originally. This is shown in Figure 9.

A publication is propagated to a remote queue manager only if a subscription to that topic exists on that remote queue manager.

A queue manager consolidates all the subscriptions that are created on it, whether from local applications or from remote queue managers. In turn, the queue manager creates subscriptions for these topics with its neighbors, unless a subscription already exists. This is shown in Figure 10 on page 21.

When an application publishes information, the receiving queue manager forwards it (possibly through one or more intermediate queue managers) using transmission queues to any applications that have valid subscriptions on remote queue managers. This is shown in Figure 11 on page 21.

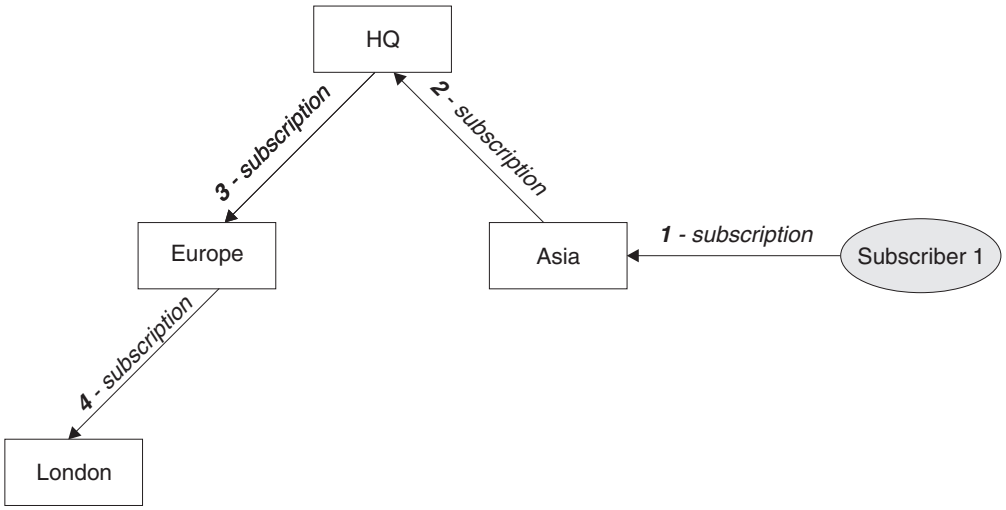


Figure 9. Propagation of subscriptions through a queue manager network. Subscriber 1 registers a subscription for a particular topic on the Asia queue manager (1). The subscription for this topic is forwarded to all other queue managers in the network (2,3,4).

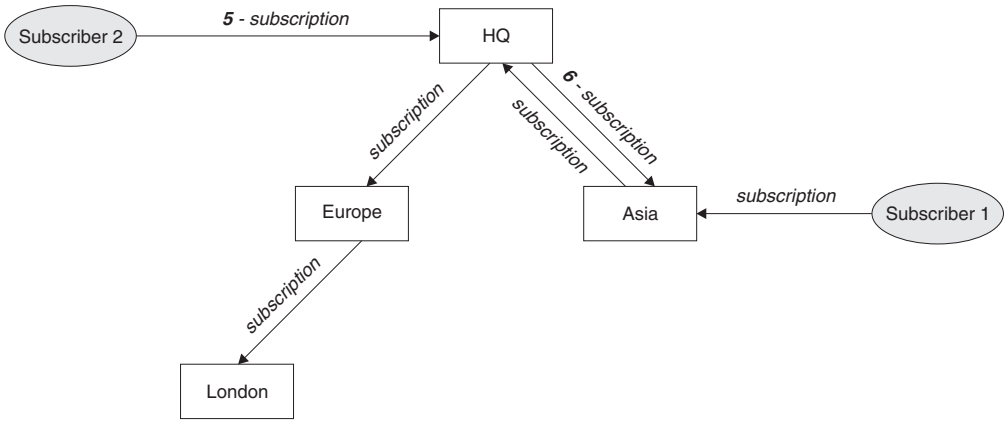


Figure 10. Multiple subscriptions. Subscriber 2 registers a subscription, to the same topic as in Figure 9 on page 20, on the HQ queue manager (5). The subscription for this topic is forwarded to the Asia queue manager, so that it is aware that subscriptions exist elsewhere on the network (6). The subscription does not have to be forwarded to the Europe queue manager, because a subscription for this topic has already been registered (step 3 in Figure 9 on page 20).

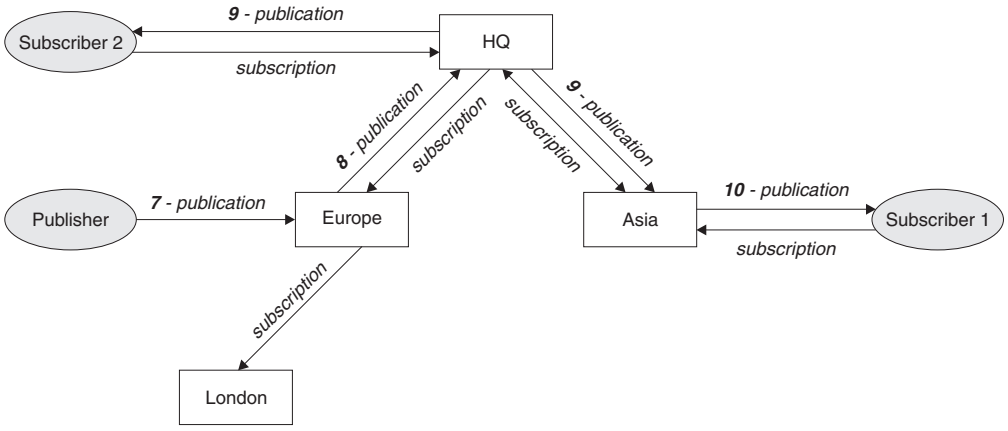


Figure 11. Propagation of publications through a queue manager network. A publisher sends a publication, on the same topic as in Figure 10, to the Europe queue manager (7). A subscription for this topic exists from HQ to Europe, so the publication is forwarded to the HQ queue manager (8). However, no subscription exists from London to Europe (only from Europe to London), so the publication is not forwarded to the London queue manager. The HQ queue manager sends the publication directly to subscriber 2 and to the Asia queue manager (9), from where it is forwarded to subscriber 1 (10).

When a queue manager sends any publications or subscriptions to another queue manager, it sets its own user ID in the message, and uses its own authority to put the message. This means that the queue manager must have the authority to put messages onto other queue managers' 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 queue manager.

The interconnected nature of publish/subscribe queue managers means that it takes some time for the proxy subscription to propagate around all nodes in the network. The consequence of this is that once a subscription has been made, remote publications are not necessarily received immediately; this can be addressed by using PROXYSUB(FORCE) as described in "More on routing mechanisms" on page 22.

The subscription operation completes when the proxy subscription has been put on the appropriate transmission queue for each directly connected queue manager, and will not include the propagation of the proxy subscription out to the rest of the topology. Proxy subscriptions are associated with the queue manager name that created them. If one queue manager is attached, by a hierarchical connection or as part of a publish/subscribe cluster, to more than one queue manager with the same queue manager name, this can result in publications failing to reach one or all of the identically named remote queue managers. To avoid this problem, as with point-to-point messaging, give queue managers unique names, especially if they are directly or indirectly connected in a WebSphere MQ network.

Within a distributed publish/subscribe network the flow of publications and subscriptions can be controlled, and if appropriate, restricted, using publication and subscription scope.

Proxy subscription aggregation and publication aggregation

Distributed publish/subscribe publications and proxy subscriptions are aggregated to minimize the quantity of messages passing between publish/subscribe queue managers.

Proxy subscription aggregation

Proxy subscriptions are aggregated using a simple duplicate elimination system. For a given resolved topic string, a proxy subscription is sent to directly connected publish/subscribe queue managers on the first local subscription or received proxy subscription.

Subsequent subscriptions make use of this existing proxy subscription. The proxy subscription is cancelled only after the last local subscription or received proxy subscription is cancelled.

Note: If PROXYSUB(FORCE) is set, a proxy subscription might be sent before the first local subscription or received proxy subscription, and will not be cancelled even after the last local subscription or received proxy subscription is cancelled.

Publication aggregation

It is possible for more than one proxy subscription to match the topic string of a single publication when the proxy subscriptions contain wildcards. If a message is published on a queue manager that matches two or more proxy subscriptions created by a single connected queue manager, only one copy of the publication is forwarded to the remote queue manager to satisfy the multiple proxy subscriptions.

More on routing mechanisms

Publish everywhere is an alternative routing mechanism to proxy subscription-forwarding. Publish everywhere works by publishing to all directly connected queue managers regardless of proxy subscriptions. Publish everywhere is not supported in publish/subscribe clusters or hierarchies, but a similar technique is available by using the PROXYSUB attribute for a high-level topic object.

PROXYSUB attribute for a high-level topic object is explained in the following comparison:

Publish everywhere

If publish everywhere routing is available in a publish/subscribe cluster,

there is no need for any proxy subscriptions and all publications are published to every member of the publish/subscribe clusters.

The advantages of publish everywhere are the removal of latency introduced by the propagation of proxy subscriptions, and the removal of the network overhead caused by proxy subscription propagation where the subscription is frequently created and deleted.

Proxy-subscription forwarding

To achieve a similar behavior to publish everywhere, alter the topic object, as follows:

```
ALTER TOPIC("SYSTEM.BASE.TOPIC") PROXYSUB(FORCE)
```

This forces the sending of a wildcard proxy subscription, for the topic string associated with this topic object, to every directly connected member of the publish/subscribe topology, regardless of whether any local subscriptions have been made.

When this forced proxy subscription has been propagated throughout the topology, any new subscriptions immediately receive any publications from other connected queue manager, without suffering latency. Proxy subscriptions for these new subscriptions are still propagated to each of the directly connected publish/subscribe queue managers; preventing a break in flow of publications if this behavior is turned off later.

Wildcard rules

Wildcards in proxy subscriptions are converted to use topic wildcards.

When a subscription for a wildcard is received, it can be either a character, as used by WebSphere MQ Version 6.0, or a topic, as used by WebSphere Message Broker Version 6.0 and WebSphere MQ Version 7.0 as follows:

- Character wildcards use '*' to represent any character (including '/').
- Topic wildcards use '#' to represent a portion of the topic space between '/' characters.

In WebSphere MQ Version 7.0, all proxy subscriptions are converted to use topic wildcards. To achieve this, if a character wildcard is found, it is replaced with a '#' character, back to the nearest '/'. For example, '/aaa/bbb/c*d' is converted to '/aaa/bbb/#'. This results in remote queue managers sending slightly more publications than were explicitly subscribed to, but these are filtered out by the local queue manager as it delivers the publications to its local subscribers.

Controlling the flow of publications and subscriptions

Scope is separated into publication and subscription scope so that queue managers can pass publications into, but not out of the publish/subscribe cluster, or out of, but not into the publish/subscribe cluster.

Publication scope

The scope of a publication controls whether queue managers distribute the publication to remote subscribers.

The PUBSCOPE topic attribute can be used to determine the scope of publications made to a specific topic. You can set the attribute to one of the following values:

QMGR
The publication is delivered only to local subscribers. These publications are called *local publications*. Local publications are not forwarded to remote queue managers and therefore are not received by remote queue managers' subscribers.

ALL The publication is delivered to local subscribers and remote subscribers through directly connected queue managers. These publications are called *global publications*.

Publishers can also specify whether a publication is local or global using the MQPMO_SCOPE_QMGR put message option, if this option is used, it overrides any behavior that has been set using the PUBSCOPE topic attribute.

Subscription scope

The scope of a subscription controls whether a subscription receives publications made on remote queue managers. You use the SUBSCOPE topic attribute to administer the scope of subscriptions.

Subscribers can decide to receive only local publications using the MQSO_SCOPE_QMGR subscription option. The MQSO_SCOPE_QMGR option determines whether a proxy subscription is created on remote queue managers in the network so that they are aware of the subscription and route publications to the local queue manager. If this option is not used, the subscriber will receive both local and global publications.

You can set the attribute to one of the following values:

QMGR
The subscription is not propagated to directly connected queue managers, and receives publications only from local publishers.

ALL The subscription is propagated to directly connected queue managers, and receives publications from local publishers and remote publishers through directly connected queue managers.

Overlapping topics

The scope of publications and subscriptions is defined in both local topic objects, as shown in the following information, and cluster topic objects.

For the following local topic definitions, a local application that subscribes using topic string '/football/#' will not receive remote publications on 'football/myteam':

```
DEFINE TOPIC(A) TOPICSTR('/football') SUBSCOPE(ALL)
DEFINE TOPIC(B) TOPICSTR('/football/myteam') SUBSCOPE(QMGR)
```

Note: Subscribers can restrict SUBSCOPE, so that remote publications are not received, by using MQSO_SCOPE_QMGR.

Retained publications

It is not good practice for two or more applications to publish retained publications to the same topic on the same or different queue managers within a single publish/subscribe topology.

It is possible that different retained publications could be active at different queue managers for the same topic, leading to unexpected behavior. As multiple proxy subscriptions are distributed, multiple retained publications could be received.

Distributed publish/subscribe security

Distributed publish/subscribe internal messages such as proxy subscriptions, publications, and so on, are put to distributed publish/subscribe system queues (SYSTEM.INTER.QMGR.CONTROL, for example) by the receiving channel using normal channel security rules. The information and diagrams in this topic highlight the various processes and user IDs involved in the delivery of these messages.

Local access control

Access to topics for publication and subscriptions is governed by local security definitions and rules that are described in “Topic objects” on page 71. On z/OS, no local topic object is required to establish access control. This is also true on distributed systems, so administrators can choose to apply access control to clustered topic objects irrespective of whether they exist in the cluster yet.

System administrators are responsible for access control on their local system and trust other members of the hierarchy or cluster collectives to which they are attached to be responsible for their own access control policy. It might not be necessary to impose any access control, or access control can be defined on high level objects in the topic tree, or fine level access control can be defined for each subdivision of the topic name space. Because access control is defined for each separate machine it is likely to be burdensome if fine level control is needed.

Making a proxy subscription

Trust for an organization to connect its queue manager to your queue manager is confirmed by normal channel authentication means. If that trusted organization is then allowed to do distributed publish/subscribe, an authority check is done when the channel puts the message to a distributed publish/subscribe queue; for example, SYSTEM.INTER.QMGR.CONTROL. The user ID for the queue authority check depends on the PUTAUT value of the receiving channel (for example, the user ID of the channel, MCAUSER, message context, and so on, depending on value and platform). For more information on channel security, see WebSphere MQ Security.

Proxy subscriptions will be made with the user ID of the distributed publish/subscribe agent on the remote queue manager (QM2 in Figure 12 on page 26) which can then easily be granted access to local topic object profiles, because that user ID is defined in the system and there are therefore no domain conflicts.

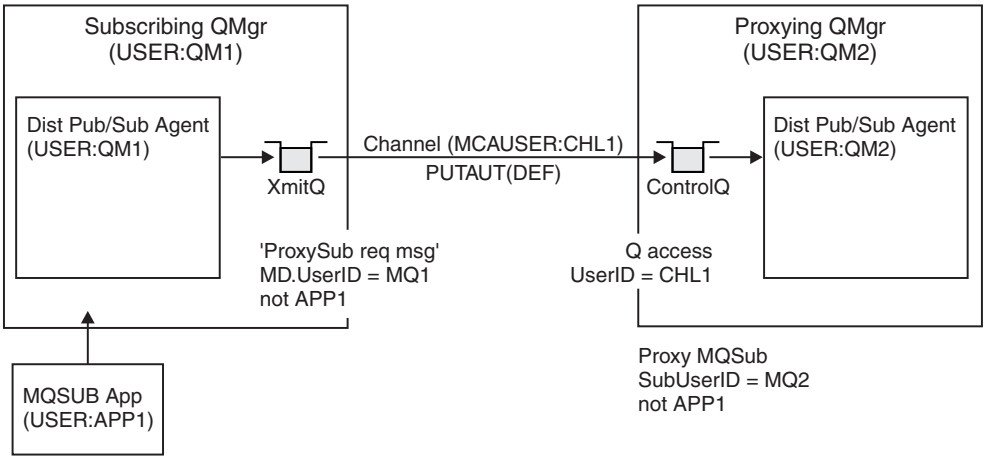


Figure 12. Proxy subscription security, making a subscription

Sending back remote publications

When a publication is made on the publishing queue manager, a copy satisfies the proxy subscription that was made, and the context of that message contains the context of the user ID which made the subscription, QM2 in Figure 13 on page 27. The proxy subscription is made with a destination queue that is a remote queue, so the publication message is resolved onto a transmission queue.

Again, trust for an organization to connect its queue manager, QM2, to another queue manager, QM1, is confirmed by normal channel authentication means. If that trusted organization is then allowed to do distributed publish/subscribe, an authority check is done when the channel puts the publication message to the distributed publish/subscribe publication queue SYSTEM.INTER.QMGR.PUBS. The user ID for the queue authority check depends on the PUTAUT value of the receiving channel (for example, the user ID of the channel, MCAUSER, message context, and so on, depending on value and platform). For more information on channel security, see WebSphere MQ Security.

When the publication message reaches the subscribing queue manager, another MQPUT to the topic is done under the authority of that queue manager and the context with the message is replaced by the context of each of the local subscribers as they are each given the message.

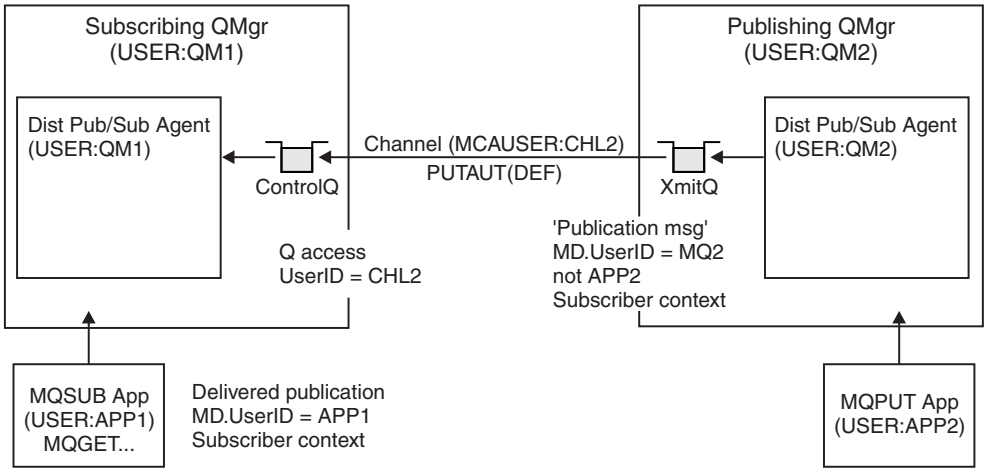


Figure 13. Proxy subscription security, forwarding publications

This means that, on a system where little has been considered regarding security, the distributed publish/subscribe processes are likely to be running under a user ID in the mqm group, the MCAUSER parameter on a channel will be blank (the default), and messages are delivered to the various system queues as required. This makes it easy to set up a proof of concept to demonstrate distributed publish/subscribe.

On a system where security is more seriously considered, these internal messages are subject to the same security controls as any message going over the channel.

If the channel is set up with a non-blank MCAUSER and a PUTAUT value specifying that MCAUSER should be checked, then the MCAUSER in question must be granted access to SYSTEM.INTER.QMGR.* queues. Where there are multiple different remote queue managers with channels running under different MCAUSER ids (for instance, when multiple hierarchical connections are configured on a single queue manager), then all those user IDs need to be granted access to the SYSTEM.INTER.QMGR.* queues.

If the channel is set up with a PUTAUT value specifying that the context of the message is used, then access to the SYSTEM.INTER.QMGR.* queues are checked based on the user ID inside the internal message. Because all these messages are put by the distributed publish/subscribe agent's user ID from the queue manager that is sending the internal message, or publication message (see Figure 13), it is not too large a set of user IDs to grant access to the various system queues (one per remote queue manager), should you want to set up your distributed publish/subscribe security in this way. It still has all of the same issues that channel context security always has; that of the different user ID domains and the fact that the user ID in the message might not be defined on the receiving system. However, it is a perfectly acceptable way to run if required.

System queue security in the *WebSphere MQ z/OS System Setup Guide* provides a list of queues and the access that is required to securely set up your distributed publish/subscribe environment. If any internal messages or publications fail to be put due to security violations, the channel writes a message to the log in the normal manner and the messages can be sent to the dead-letter queue according to normal channel error processing.

All inter-queue manager messaging for the purposes of distributed publish/subscribe runs using normal channel security. No special casing is required in the security manager on behalf of the distributed publish/subscribe component.

For information on restricting publications and proxy subscriptions at the topic level, see "Topic objects" on page 71.

Distributed publish/subscribe system queues

Four system queues are used by queue managers when they do publish/subscribe messaging. You normally need to be aware of their existence only for problem determination or capacity planning purposes.

Table 3. Publish/subscribe system queues

System queue	Purpose
SYSTEM.INTER.QMGR.CONTROL	WebSphere MQ distributed publish/subscribe control queue
SYSTEM.INTER.QMGR.FANREQ	WebSphere MQ distributed publish/subscribe internal proxy subscription fan-out process input queue
SYSTEM.INTER.QMGR.PUBS	WebSphere MQ distributed publish/subscribe publications
SYSTEM.HIERARCHY.STATE	WebSphere MQ distributed publish/subscribe hierarchy relationship state

The attributes of the distributed publish/subscribe system queues are as displayed in Table 4.

Table 4. Attributes of publish/subscribe system queues

Attribute	Value
DEFPSIST	Yes
DEFSOPT	This takes the value EXCL.
MAXMSGL	On AIX®, HP-UX, Linux®, i5/OS®, Solaris and Windows® this takes the value of MAXMSGL parameter of the ALTER QMGR command. On z/OS this takes the value 100 MB (104 857 600 bytes).
MAXDEPTH	On AIX, HP-UX, Linux, i5/OS, Solaris, Windows and z/OS this takes the value 999 999 999.
SHARE	This is a keyword that specifies that the queue can be shared for GET.
STGCLASS	On z/OS this takes the value 'SYSTEM'. On other platforms this attribute is not used.

Publish/subscribe system queue errors

Errors can occur when distributed publish/subscribe queue manager queues are unavailable.

If the fan-out request queue SYSTEM.INTER.QMGR.FANREQ is unavailable, the MQSUB API receives reason codes and error messages written to the error log, on occasions where proxy subscriptions need to be delivered to directly connected queue managers.

If the hierarchy relationship state queue SYSTEM.HIERARCHY.STATE is unavailable, an error message is written to the error log and the publish/subscribe engine is put into COMPAT mode.

If any other of the SYSTEM.INTER.QMGR queues are unavailable, an error message is written to the error log, and although function is not disabled, it is likely that publish/subscribe messages will build up on queues on remote queue managers.

If the transmission queue to a parent, child or publish/subscribe cluster queue manager is unavailable:

1. The MQPUT API receives reason codes and the publications are not delivered.
2. Received inter-queue manager publications are backed out to the input queue, and subsequently re-attempted, being placed on the dead letter queue if the backout threshold is reached.
3. Proxy subscriptions are backed out to the fanout request queue, and subsequently attempted again, being placed on the dead letter queue if the backout threshold is reached; in which case the proxy subscription will not be delivered to any connected queue manager.
4. Hierarchy relationship protocol messages fail, and the connection status is marked as ERROR on the PUBSUB command.

Publish/subscribe topologies

A *publish/subscribe topology* consists of queue managers and the connections between them, that support publish/subscribe applications.

A publish/subscribe application can consist of a network of queue managers connected together. The queue managers can all be on the same physical system, or they can be distributed over several physical systems. By connecting queue managers together, publications can be received by an application using any queue manager in the network.

This provides the following benefits:

- Client applications can communicate with a nearby queue manager rather than with a distant queue manager, thereby getting better response times.
- By using more than one queue manager, more subscribers can be supported.

You can arrange queue managers that are doing publish/subscribe messaging in two different ways, clusters and hierarchies. For more information about these two topologies and to find out which is most appropriate for you, refer to the information in this chapter.

It is possible to use both topologies in combination by joining clusters together in a hierarchy.

Publish/subscribe clusters

You can improve the performance of your publish/subscribe network by arranging your queue managers in a publish/subscribe cluster. A publish/subscribe cluster consists of a set of queue managers connected together, with direct channel links between all members, to form all or part of a publish/subscribe network.

A *publish/subscribe cluster* is a set of queue managers that are fully interconnected and form part of a multi-queue manager network for publish/subscribe

applications. A cluster that is used for publish/subscribe messaging is no different from a standard WebSphere MQ cluster. As such, the queue managers within the publish/subscribe cluster can exist on physically separate computers and each pair of queue managers is connected together by a pair of channels. For information about how to plan and configure a WebSphere MQ cluster refer to *WebSphere MQ Queue Manager Clusters*.

Using clusters in a publish/subscribe topology provides the following benefits:

- Messages destined for a specific queue manager in the same cluster are transported directly to that queue manager and do not need to pass through an intermediate queue manager. This improves performance and optimizes inter-queue manager publish/subscribe traffic, in comparison with a hierarchical topology.
- There is no single point of failure in this topology. If one queue manager is not available, publications and subscriptions are still able to flow through the rest of the publish/subscribe system because each queue manager is directly connected with each other.
- If your clients are geographically dispersed, you can set up a cluster in each location, and connect the clusters (by joining a single queue manager in each cluster) to optimize the flow of publications and subscriptions through the network.
- You can group clients according to the topics to which they publish and subscribe.

  Clients that share common topics can connect to queue managers within a cluster. The common publications are transported efficiently within the cluster, because they pass through only queue managers that have at least one client with an interest in those common topics.
- A subscribing application can connect to its nearest queue manager, to improve its own performance. The queue manager receives all messages that match the subscription registration of the client from all queue managers within the cluster.

  The performance of a client application is also improved for other services that are requested from this queue manager. A client application can use both publish/subscribe and point-to-point messaging.
- The number of clients per queue manager can be reduced by adding more queue manager to the cluster to share workload. This makes a publish/subscribe cluster topology highly scalable.

When you create a cluster it is possible to create a loop causing messages to cycle forever within the network, nothing will prevent you from doing this but you will be made aware of it because of the fingerprint that is added by the queue manager (stored as a message property).

A publish/subscribe cluster is created when a clustered topic is defined. This definition is shared with all members of the cluster. This means that publications on the clustered topic are shared with all members of the cluster.

When at least one clustered topic object is defined, all queue managers within the cluster will be notified about each other.

If you have several queue managers in your publish/subscribe system, many channels are required to connect these queue managers together. However, the connections between queue managers can be created automatically to reduce the administrative work load.

Cluster topics

You can cluster topics in a similar manner to cluster queues, although an individual administrative topic object can be a member of only one cluster. Topic objects do not have an equivalent to the CLUSNL (cluster namelist) attribute.

When a cluster topic is defined, the cluster topic object is published to the full repositories. The full repositories then push all cluster topic definitions to all queue managers within the cluster.

At each queue manager a single topic space is constructed from the local and cluster topic definitions. When an application subscribes to a topic that resolves to a clustered topic, WebSphere MQ creates a proxy subscription and sends it, from the queue manager to which the subscriber connected, to all members of the cluster in which the clustered topic object is defined.

If a local and cluster topic definition exists for a single topic string, the local definition is used. Where two or more cluster topic definitions, for a single topic string, have differing attributes or exist in more than one cluster, a message is written to the log and the most recently received cluster topic definition is used. It is acceptable to define two or more cluster topic definitions with identical attributes for a single topic string.

If you are working in clusters, and a single queue manager defines a local topic object to override the behavior of a cluster topic object, this does not prevent other queue managers in the cluster from sending proxy subscriptions to the queue manager that defined the local topic object. To prevent publications being sent to those proxy subscriptions, you need to specify PUBSCOPE(QMGR) on the local topic object.

If the queue manager on which a cluster topic is defined is unavailable, you cannot alter the cluster topic definition remotely. However, you can use the RESET CLUSTER command to remove the queue manager from the cluster. You can define an additional cluster topic definition on the same topic string at a different queue manager within the cluster; if defined with differing attributes, this overrides the previous definition and a message is written to the log. If the original queue manager subsequently becomes available, its clustered topic object must either be deleted or its definition updated to match the additional cluster definition.

Cluster topic names

Cluster topic names are character strings. For example, you could have high-level cluster topics named 'Sport', 'Stock', 'Films', and 'TV', and you could divide the 'Sport' cluster topic into separate, more specific cluster topics covering different sports:

```
Sport/Soccer Sport/Golf Sport/Tennis
```

These cluster topics could then be divided further, to separate different types of information about each sport:

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

WebSphere MQ publish/subscribe does not recognize that the forward slash (/) character is being used in a special way, but if you use the forward slash (/) character as a separator, you can ensure compatibility with other WebSphere business integration applications.

You can use any character in the single-byte character set for which the machine is configured in a character string. Consider, however, whether the cluster topic string might need to be translated to a different character representation, in which case you must use only those characters that are available in the configured character set of all relevant machines.

Cluster topic strings are case sensitive, and a blank character has no special meaning. As a subscriber, you can specify a cluster topic or range of cluster topics using wildcards to receive the information in which you are interested.

Key roles for publish/subscribe cluster queue managers

There are two key roles for queue managers in publish/subscribe clusters that you should consider when designing a publish/subscribe cluster.

Full repositories

You can define full repositories on any queue manager in the publish/subscribe cluster. As with a normal cluster, a publish/subscribe cluster should ideally have two full repositories, hosted in highly available machines.

Cluster topic host

A cluster topic host is a queue manager where a clustered topic object is defined. You can define clustered topic objects on any queue manager in the publish/subscribe cluster. When at least one clustered topic exists within a cluster, the cluster is a publish/subscribe cluster. Ideally, all clustered topic objects should be identically defined on two queue managers and these machines should be highly available.

If a single host of a clustered topic object is lost, for example, because of disk failure, any cluster topic cache records that are based on the clustered topic object, that already exist in the cluster cache on other queue managers, are usable within the cluster for a period of up to 30 days, or until the cache is refreshed.

You can redefine the clustered topic object on a queue manager that is working correctly. If a new object is not defined within 27 days (inclusive) after the host queue manager failure, all members of the cluster will report that an expected object update has not been received.

Full repositories and topic hosts do not need to overlap or be separated. In publish/subscribe clusters that have just two highly available computers among many computers, it is good practice to define both the highly available computers as full repositories and cluster topic hosts.

In publish/subscribe clusters with many highly available computers it is good practice to define full repositories and cluster topic hosts on separate highly available computers, so that the operation and maintenance of one function can be managed without affecting the operation of other functions.

Overlapping cluster support and publish/subscribe

With WebSphere MQ clusters, a single queue manager can be a member of more than one cluster.

A reason for making a single queue manager a member of more than one cluster is to create a cluster gateway between two clusters, so that messages originating in one cluster can be routed to another cluster. Although a WebSphere MQ queue manager can be a member of more than one cluster and more than one publish/subscribe cluster, publications are not passed from one cluster to another

by means of overlapping clusters. The scope of proxy subscriptions is limited to the single cluster in which the clustered topic is defined.

In Figure 14, an application connected to queue manager QM3, subscribing on a topic that resolves to topic object T_B (which exists only in CLUSTER 1) results in proxy subscriptions being sent from queue manager QM3 to both queue managers QM1 and QM2. An application connected to queue manager QM3, subscribing on a topic that resolved to topic object T_C (which exists only in the CLUSTER 2) results in proxy subscriptions being sent from queue manager QM3 to both queue managers QM4 and QM5.

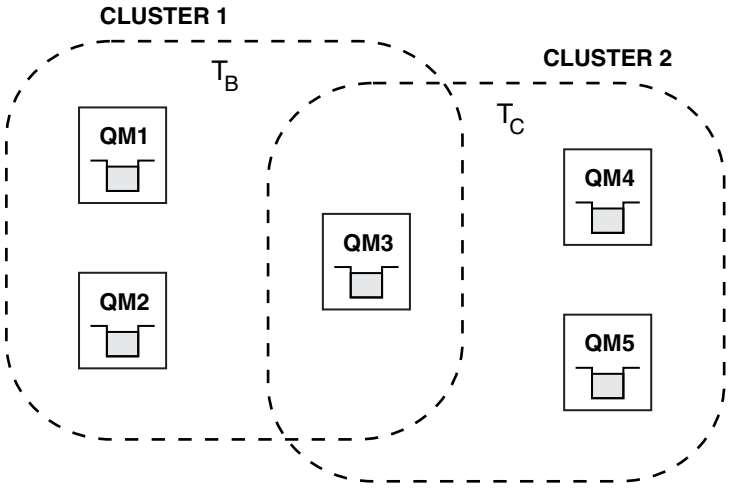


Figure 14. Overlapping clusters: two clusters each subscribing to different topics

In Figure 15 on page 34 messages are output to the log of queue manager QM3 informing users that the topic object T_A exists in two clusters. An application connected to queue manager QM3, subscribing on a topic that resolved to topic object T_A (which exists in both CLUSTER 1 and CLUSTER 2) results in proxy subscriptions being sent to one cluster only – so either to queue managers QM1 and QM2 or to queue managers QM4 and QM5. The cluster chosen depends on which cluster topic object was added last to the cluster cache in queue manager QM3.

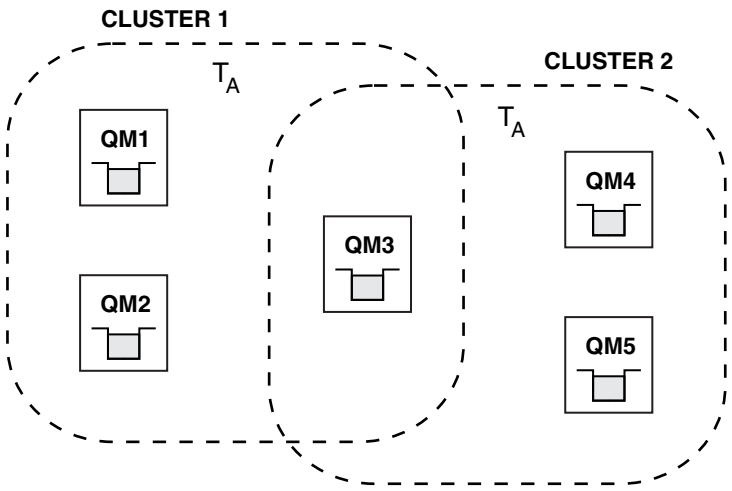


Figure 15. Overlapping clusters: two clusters each subscribing to the same topic

Publish/subscribe messages, for example, application publications and proxy subscriptions, are transmitted only over cluster channels that are part of the publish/subscribe cluster in which the cluster topic that the message relates to is defined.

For example, for the following topic definitions:

- Topic: T_A in CLUSTER 1 with TopicString: /football
- Topic: T_A in CLUSTER 2 with TopicString: /tennis

A subscription for T_A made on queue manager QM3 resolves to TopicString /tennis, assuming that this was the latest definition to be made, and causes receipt of publications on CLUSTER 2 for topic /tennis.

If any queue manager receives multiple definitions on the same topic string which differ in any detail, including cluster name, the behavior of publications and subscriptions on those topics or topic string is undefined. An informational message is issued to alert the administrator to the duplicate definition.

Subscription scope and publication scope in publish/subscribe clusters

The scope of publications and subscriptions is defined in the cluster topic object.

If a cluster topic object is defined with SUBSCOPE(QMGR), the definition is shared with the cluster, but the scope of subscriptions based on that topic is local only and publications are not shared with the cluster.

If a cluster topic object is defined with PUBSCOPE(QMGR), the definition is shared with the cluster, but the scope of publications based on that topic is local only and they are not shared with the cluster.

REFRESH CLUSTER considerations

As with point-to-point messaging, REFRESH CLUSTER can cause temporary disruption to publish/subscribe traffic.

The disruption can occur as follows:

- Up to 10 second pauses in message delivery.

- MQOPEN and MQPUT failures, for example, MQRC_NO_DESTINATIONS_AVAILABLE.

Publish/subscribe hierarchies

Queue managers can be grouped together in a hierarchy, where the hierarchy contains one or more queue managers that are directly connected. Queue managers are connected together using a connection-time parent and child relationship. When two queue managers are connected together for the first time, the child queue manager is connected to the parent queue manager.

When the parent and child queue managers are connected in a hierarchy there is no functional difference between them until you disconnect queue managers from the hierarchy.

Note: WebSphere MQ hierarchical connections require that the queue manager attribute PSMODE is set to ENABLED.

Connecting a queue manager to a hierarchy

You can connect a local queue manager to a parent queue manager to create a hierarchy.

Before you can add a queue manager to a hierarchy, channels in both directions must exist between the prospective parent queue manager and child queue manager. Queue managers use explicit addressing when sending messages to queues that reside on another queue manager. When the queue is opened by the queue manager, both the queue and queue manager names are specified. To facilitate multi-queue manager 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, you can use a queue manager alias definition to ensure that messages are placed on the appropriate transmission queue.

For example, to specify that messages sent to queue manager PARENT are placed on transmission queue PARENT.XMITQ, use the following MQSC command:

```
DEFINE QREMOTE (PARENT) RNAME('') RQMNAME(PARENT) XMITQ(PARENT.XMITQ)
```

In WebSphere MQ Version 6.0, when the appropriate channels and queues were defined, queue managers were connected to one another using the strmqbrk command. The strmqbrk command works differently in WebSphere MQ Version 7.0 and you can no longer use it to connect children to parents. Instead, you connect queue managers by using the following command:

```
ALTER QMGR PARENT(<parent qmgr>)
```

Where `<parent qmgr>` is the name of the parent queue manager to which the local queue manager is connected, using appropriate channels and queues. You can use a queue manager alias, see *WebSphere MQ Application Programming Guide* for details. A queue manager can have only one parent.

If a parent has already been defined, the ALTER QMGR PARENT command disconnects from the original parent and sends a connection flow to the new parent queue manager.

If the original parent is unavailable, the ALTER QMGR PARENT command does not fail, but a warning is written to the error log. The ALTER QMGR PARENT

command is not synchronous with the complete negotiation with the parent queue manager. Use the following command to monitor the status of the hierarchical connection to the parent queue manager:

```
DISPLAY PUBSUB TYPE(PARENT)
```

Disconnecting a queue manager from a hierarchy

You can disconnect a child queue manager from a parent queue manager in a hierarchy. You can also check that the disconnection was successful.

To disconnect queue managers, issue the following MQSC command at the child queue manager:

```
ALTER QMGR PARENT(' ')
```

To confirm that the parent has been disconnected, issue the command `DISPLAY PUBSUB TYPE(PARENT)` at the child queue manager and check that NONE is returned.

To confirm that the child queue manager has been disconnected, issue the command `DISPLAY PUBSUB TYPE(CHILD)` at the parent queue manager, and check that the name of the child queue manager is not returned. When the parent and child have been disconnected, you can manually delete the queues and channels on the child queue manager and the parent queue manager, as required.

Note: In WebSphere MQ Version 6.0, queue managers were disconnected from one another using the dltmqbrk command, and required that all child queue managers were disconnected first. The dltmqbrk command works differently in WebSphere MQ Version 7.0 and can no longer be used to disconnect children from parents.

Chapter 4. Migrating to WebSphere Version 7.0 publish/subscribe

strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0)

Purpose

Use the **strmqbrk** command to migrate WebSphere MQ Version 6.0 broker state to WebSphere MQ Version 7.0 publish/subscribe.

In WebSphere MQ Version 6.0, **strmqbrk** started a broker. The WebSphere MQ Version 7.0 publish/subscribe engine cannot be started in this manner. To enable publish/subscribe for a queue manager, use the ALTER QMGR command; for details, see ALTER QMGRin the *WebSphere MQ Script (MQSC) Command Reference*.

Publish/subscribe command messages

The WebSphere MQ Version 6.0 publish/subscribe command message interface is being deprecated. If you have applications that use this interface directly, you should migrate those applications to use the new Version 7.0 publish/subscribe functions.

The following sections explain how to replace existing command messages.

Identity

In WebSphere MQ Version 6 there were two ways of identifying a subscriber. These were referred to as the traditional identity and the subscription name.

The traditional identity was also used to identify a publisher. The traditional identity was a combination of queue name, queue manager name, and optional correlation identifier.

A publisher no longer has an explicit publisher identity, but can be identified in the same way as any only WebSphere MQ application, by means of its connection to the queue manager. Since there is no explicit registration of a publisher, or his identity over and above what can be obtained by displaying the connections to the queue manager, there is no longer a need for the anonymous option on Register Publisher. Your application must now use the SubName field in the MQSD to identify a subscriber.

The correlation identifier also had a secondary use which was to allow subscribers to MQGET by CorrelId to only get publications for a particular subscription, if there were multiple subscriptions all using the same queue. This is provided by using the SubCorrelId field returned in the MQSD at MQSUB time.

Stream Name

MQPS_STREAM_NAME is deprecated since stream names are part of the full topic name. Stream names can be mapped to administrative topic objects, and then the topic name used along with the stream name can be mapped to a topic string to be concatenated with the topic string from the topic object. For example, if the

application was previously using a stream queue name of SYSTEM.BROKER.RESULTS.STREAM and a topic of `Sport/Soccer/State/LatestScore/*`, then a topic object can be created whose name is SYSTEM.BROKER.RESULTS.STREAM which is defined to have a TOPICSTR of / and the new application will provide a two part topic name in the MQOD or MQSD using an ObjectName of SYSTEM.BROKER.RESULTS.STREAM and an ObjectString of `Sport/Soccer/State/LatestScore/*`.

If an administrative topic object that does not exist is used in place of a stream name, the error (effectively mapping to MQRCCF_STREAM_ERROR) which is given is MQRC_UNKNOWN_OBJECT_NAME. supported.

Application migration details

When migrating to use the MQ API to do publish/subscribe, the code within any one application program must be consistent.

The application program must not contain a mixture of these deprecated APIs and the new MQ API options. An entire application suite, such as the combination of a subscribing application program and a publishing application program, does not all need to be migrated at the same time. Interaction between a publishing application program using the deprecated APIs and a subscribing application using the new MQ API is supported.

Delete Publication - Version 7 replacement

The Delete Publication command message contains a number of parameters. This should be replaced by using the PCF ClearTopic command. This section details the equivalent options or fields in the PCF command message to show how an application would migrate from using the Delete Publication command message to using the PCF ClearTopic command message.

Required parameters

MQPS_COMMAND with value MQPS_DELETE_PUBLICATION is implied when you use the ClearTopic command.

MQPS_TOPIC is provided in a field in the ClearTopic command message. If your application provided more than one MQPS_TOPIC in a single Delete Publication command message, it must now issue a separate ClearTopic call for each separate topic string.

Optional parameters

MQPS_DELETE_OPTIONS is replaced with an attribute of the ClearTopic command message.

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_UNKNOWN_OBJECT_NAME
MQRCCF_TOPIC_ERROR	MQRC_OBJECT_STRING_ERROR
MQRCCF_INCORRECT_STREAM	See Note 1

Notes:

1. No equivalent since there is no need to provide the stream name twice, once in the command and once by putting it to the stream queue, so you cannot have a mismatch.

Deregister publisher - Version 7 replacement

The Deregister Publisher command message contains a number of parameters. You should replace it with the MQCLOSE verb. This section details the equivalent options or fields in the MQ API to show how to migrate an application from the Deregister Publisher command message to MQCLOSE.

A difference in behaviour will be seen because a Register Publisher command could leave an application registered even when it was not connected, whereas the equivalent MQOPEN will only show a publisher's intent when the application is connected and keeps the handle from MQOPEN available. Even without issuing MQCLOSE, an application will be deregistered when the queue manager detects that the application's connection is lost.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_PUBLISHER is implied when closing a handle to a topic previously opened using MQOPEN with the MQOO_OUTPUT option.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) these attributes are now implied by the provision of the handle obtained when opening the topic.

MQPS_REGISTRATION_OPTIONS is replaced with options on the MQCLOSE call. See MQCLOSE for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_CORREL_ID_AS_IDENTITY	MQREGO_CORREL_ID_AS_IDENTITY	See Chapter 7, "Publish/subscribe deprecated function," on page 81
MQPS_DEREGISTER_ALL	MQREGO_DEREGISTER_ALL	See Note 1

String constant	Integer constant	MQCLOSE Options field constant
Notes: 1. Since only one topic can be opened by the MQOPEN call, closing the handle closes that one topic. There is no need for an equivalent option. If many topics are opened, simply issuing MQDISC will close them all, saving the need to MQCLOSE each handle.		

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81, although in this case, the stream name is implied by the provision of the handle obtained when opening the topic. MQPS_TOPIC is implied by the provision of the handle obtained when opening the topic.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_TOPIC_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_NOT_REGISTERED	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_MGR_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_DUPLICATE_IDENTITY	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_UNKNOWN_STREAM	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_REG_OPTIONS_ERROR	MQRC_OPTIONS_ERROR
Notes: 1. This error code implies the same type of problem, but since all of these fields are now implied by the provision of the handle obtained when opening the topic, this is the only equivalent error.	

Deregister subscriber - Version 7 replacement

The Deregister Subscriber command message contains a number of parameters. This should be replaced by using the MQCLOSE verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Deregister Subscriber command message to using MQCLOSE. If the Deregister Subscriber command message was used in a different program from that of the Register Subscriber command message, the application must now first use the MQSUB call with the MQSO_RESUME option to get a handle to the subscription, in order to deregister it.

Required parameters

MQPS_COMMAND with value MQPS_DEREGISTER_SUBSCRIBER is replaced by the use of the MQCLOSE verb with the option MQCO_REMOVE_SUB.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) these attributes are now implied by the provision of the handle obtained when subscribing to the topic.

MQPS_REGISTRATION_OPTIONS is replaced with Options on the MQCLOSE call. See MQCLOSE for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	**Integer constant**	**MQCLOSE Options field constant**
MQPS_CORREL_ID_AS_ IDENTITY	MQREGO_CORREL_ID_AS_ IDENTITY	See Note 1
MQPS_DEREGISTER_ALL	MQREGO_DEREGISTER_ ALL	See Note 2
MQPS_FULL_RESPONSE	MQREGO_FULL_RESPONSE	See Note 3
MQPS_LEAVE_ONLY	MQREGO_LEAVE_ONLY	See Note 4
MQPS_VARIABLE_USER_ID	MQREGO_VARIABLE_ USER_ID	See Note 1

Notes:

1. This option is implied by the provision of the handle obtained when subscribing to the topic.
2. Since only one topic (separate topic string that is – of course wildcards can still be used within one topic string) can be subscribed to by the MQSUB call, closing the handle closes that one topic. There is no need for an equivalent option. If many topics are opened, simply issuing MQDISC will close them all, saving the need to MQCLOSE each handle.
3. Use of this option is implied in the use of the MQSUB verb. The fields returned in the response message are now populated in the MQSD structure. See MQSUB for more details. Because an MQSUB call must be made in order to obtain the handle to pass to the MQCLOSE call, this option is deprecated.
4. Use of these options are deprecated and moved to spiCONN for the one environment where they are needed.

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81, although in this case, the stream name is implied by the provision of the handle obtained when subscribing to the topic. MQPS_SUBSCRIPTION_IDENTITY is replaced by a field in spiCONN for the one environment where it is needed. MQPS_SUBSCRIPTION_NAME is replaced by the field in the MQSD called SubName and is therefore implied by the provision of the handle obtained when subscribing to the topic. MQPS_TOPIC is provided in a field in the MQSD called ObjectString, and is therefore implied by the provision of the handle obtained when subscribing to the topic.

See MQSD for more details

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_TOPIC_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_NOT_REGISTERED	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_MGR_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_DUPLICATE_IDENTITY	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_UNKNOWN_STREAM	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_REG_OPTIONS_ERROR	MQRC_OPTIONS_ERROR

Notes:

1. This error code implies the same type of problem, but because all of these fields are now implied by the provision of the handle obtained when opening the topic, this is the only equivalent error.

Publish - Version 7 replacement

The Publish command message contains a number of parameters. This should be replaced by using the MQPUT/MQPUT1 verbs. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Publish command message to using MQPUT/MQPUT1.

Required parameters

MQPS_COMMAND with value MQPS_PUBLISH is implied when putting a message to an object handle opening a topic for MQOO_OUTPUT. If your application did not use Register Publisher, see the details in Publish for remaining unregistered.

MQPS_TOPIC is provided in a field in the MQOD called ObjectString. See MQOD - Object Descriptor for more details. If your application provided more than one MQPS_TOPIC in a single Register Publisher command message, it must now issue a separate MQOPEN call for each separate topic string.

Optional parameters

MQPS_INTEGER_DATA can be replaced with a message property.

MQPS_PUBLICATION_OPTIONS is replaced with the Options field in the MQPMO structure. See MQPMO for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_CORREL_ID_AS_IDENTITY	MQREGO_CORREL_ID_AS_IDENTITY	See Chapter 7, “Publish/subscribe deprecated function,” on page 81 for more information

String constant	Integer constant	MQCLOSE Options field constant
MQPS_IS_RETAINED_ PUBLICATION	MQREGO_IS_RETAINED_ PUBLICATION	See Note 1
MQPS_NO_REGISTRATION	MQREGO_NO_ REGISTRATION	See Note 2
MQPS_OTHER_ SUBSCRIBERS_ONLY	MQREGO_OTHER_ SUBSCRIBERS_ONLY	See Note 3
MQPS_RETAIN_ PUBLICATION	MQREGO_RETAIN_ PUBLICATION	MQPMO_RETAIN

Notes:

1. A message property will contain this information
2. This option is deprecated because publishers are no longer registered
3. This option is deprecated. If an application does not want to receive its own publications it should subscribe using the option MQSO_NO_LOCAL on the MQSUB call.

MQPS_Q_MGR_NAME is replaced by the ReplyToQMgr in the MQMD of the publication. If the publisher specifies MQPMO_NO_DIRECT_REQUEST the ReplyToQMgr will not contain the publishers queue manager name, otherwise it will.

MQPS_Q_NAME is replaced by the ReplyToQ in the MQMD of the publication. If the publisher does not set this, it is not available.

MQPS_REGISTRATION_OPTIONS is replaced with Options in the MQPMO. See MQPMO for more details. These are exactly the same as those in the section on Register Publisher below.

MQPS_SEQUENCE_NUMBER is replaced with a message property.

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81. MQPS_STRING_DATA is replaced with a message property.

Register publisher - Version 7 replacement

The Register Publisher command message contains a number of parameters. This should be replaced by using the MQOPEN verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Register Publisher command message to using MQOPEN. A difference in behaviour will be seen because a Register Publisher command could leave an application registered even when it was not connected, whereas MQOPEN will only show a publishers intent when the application is connected and keeps the handle from MQOPEN available.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_PUBLISHER is implied when opening a topic for MQOO_OUTPUT. If your application did not use Register Publisher, see the details in Publish for remaining unregistered.

MQPS_TOPIC is provided in a field in the MQOD called ObjectString. See MQOD - Object Descriptor for more details. If your application provided more than one

MQPS_TOPIC in a single Register Publisher command message, it must now issue a separate MQOPEN call for each separate topic string.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) in order for subscribing applications to be able to directly contact the publisher, then your application must now provide these details on each published message.

MQPS_REGISTRATION_OPTIONS is replaced with Options in the MQPMO. See MQPMO for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_ANONYMOUS	MQREGO_ANONYMOUS	See Chapter 7, "Publish/subscribe deprecated function," on page 81
MQPS_CORREL_ID_ AS_IDENTITY	MQREGO_CORREL_ID_ AS_IDENTITY	See Chapter 7, "Publish/subscribe deprecated function," on page 81
MQPS_DIRECT_REQUEST	MQREGO_DIRECT_ REQUEST	See Note 1
MQPS_LOCAL	MQREGO_LOCAL	MQPMO_SCOPE_QMGR

Notes:

1. Use of this option is implied if the ReplyToQ and ReplyToQMgr fields are provided in the MQMD of the message put. If these fields are not provided, the queue manager will still fill in the ReplyToQMgr as the queue manager local to the publisher. To remain completely anonymous and not even provide this information to subscribers, your application should use the MQPMO_NO_DIRECT_REQUEST option.

For MQPS_STREAM_NAME see Chapter 7, "Publish/subscribe deprecated function," on page 81.

Register subscriber - Version 7 replacement

The Register Subscriber command message contains a number of parameters. This should be replaced by using the MQSUB verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Register Subscriber command message to using MQSUB.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_SUBSCRIBER is replaced by the use of the MQSUB verb. If your application did not use Register Subscriber then the use of the MQSUB verb is not required for equivalent behaviour.

MQPS_TOPIC is provided in a field in the MQSD called ObjectString. See MQSD for more details. If your application provided more than one MQPS_TOPIC in a

single Register Subscriber command message, it must now issue a separate MQSUB call for each separate topic string.

Optional parameters

If your application provided a non-local queue name and/or a queue manager name other than the one connected to (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) then your application must now provide an object handle, which has been returned by a MQOPEN call for that queue, in the Hobj parameter of the MQSUB verb.

If your application provided the name of a queue local to the queue manager it connected to, it now has the option to request that the queue manager manage where the publications are sent. This can be done by using the MQSO_MANAGED option in the field in the MQSD called Options.

MQPS_REGISTRATION_OPTIONS is replaced with a field in the MQSD called Options. See MQSD for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_ADD_NAME	MQREGO_ADD_NAME	See Note 1
MQPS_ANONYMOUS	MQREGO_ANONYMOUS	See Identity
MQPS_CORREL_ID_AS_ IDENTITY	MQREGO_CORREL_ID_ AS_IDENTITY	See Identity (also see Note 7)
MQPS_DUPLICATES_OK	MQREGO_DUPLICATES_OK	See Note 2
MQPS_FULL_RESPONSE	MQREGO_FULL_RESPONSE	See Note 3
MQPS_INCLUDE_STREAM_ NAME	MQREGO_INCLUDE_ STREAM_NAME	See Note 4
MQPS_INFORM_IF_ RETAINED	MQREGO_INFORM_IF_ RETAINED	See Note 5
MQPS_JOIN_EXCLUSIVE	MQREGO_JOIN_ EXCLUSIVE	See Note 6
MQPS_JOIN_SHARED	MQREGO_JOIN_SHARED	See Note 6
MQPS_LOCAL	MQREGO_LOCAL	MQSO_SCOPE_QMGR
MQPS_LOCKED	MQREGO_LOCKED	See Note 6
MQPS_NEW_ PUBLICATIONS_ONLY	MQREGO_NEW_PUB LICATIONS_ONLY	MQSO_NEW_ PUBLICATIONS_ONLY
MQPS_NO_ALTERATION	MQREGO_NO_ ALTERATION	MQSO_RESUME
MQPS_NON_PERSISTENT	MQREGO_NON_ PERSISTENT	MQSO_NON_PERSISTENT
MQPS_PERSISTENT	MQREGO_PERSISTENT	MQSO_PERSISTENT
MQPS_PERSISTENT_AS_ PUBLISH	MQREGO_PERSISTENT_ AS_PUBLISH	MQSO_PERSISTENT_AS_ PUBLISH
MQPS_PERSISTENT_AS_Q	MQREFO_PERSISTENT_ AS_Q	MQSO_PERSISTENT_AS_ QUEUE_DEF

String constant	Integer constant	MQCLOSE Options field constant
MQPS_PUBLISH_ON_ REQUEST_ONLY	MQREGO_PUBLISH_ON_ REQUEST_ONLY	MQSO_PUBLICATIONS_ ON_REQUEST
MQPS_VARIABLE_USER_ID	MQREGO_VARIABLE_ USER_ID	MQSO_ANY_USERID, (also see Note 7)

Notes:

1. Use of this option is deprecated since the only identity of a subscription is the SubName. See Deprecation.
2. Use of this option is deprecated since the queued interface has been removed.
3. Use of this option is implied in the use of the MQSUB verb. The fields returned in the response message are now populated in the MQSD structure. See MQSD for more details.
4. Use of this option is deprecated since stream names are part of the full topic name.
5. Use of this option is deprecated since the information about whether a publication is a retained publication or not is a message property that is always present.
6. Use of these options are deprecated and moved to spiCONN for the one environment where they are needed.
7. A tick in this column indicates this option is also relevant for Request Update.

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81, although in this case, the stream name is implied by the provision of the handle obtained when subscribing to the topic.

MQPS_SUBSCRIPTION_IDENTITY is replaced by a field in spiCONN for the one environment where it is needed.

MQPS_SUBSCRIPTION_NAME is replaced by the field in the MQSD called SubName. See MQSD for more details.

MQPS_SUBSCRIPTION_USER_DATA is replaced by the field in the MQSD called SubUserData. See MQSD for more details.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	
MQRCCF_TOPIC_ERROR	
MQRCCF_Q_MGR_NAME_ERROR	
MQRCCF_Q_NAME_ERROR	
MQRCCF_DUPLICATE_IDENTITY	MQRC_IDENTITY_MISMATCH
MQRCCF_CORREL_ID_ERROR	
MQRCCF_NOT_AUTHORIZED	
MQRCCF_UNKNOWN_STREAM	
MQRCCF_REG_OPTIONS_ERROR	
MQRCCF_DUPLICATE_SUBSCRIPTION	

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRC_SUB_ALREADY_EXISTS	
MQRCCF_SUB_NAME_ERROR	
MQRCCF_SUB_IDENTITY_ERROR	See note 1
MQRCCF_SUBSCRIPTION_IN_USE	
MQRC_SUBSCRIPTION_IN_USE MQRCCF_SUBSCRIPTION_LOCKED	See note 1
MQRCCF_ALREADY_JOINED	See note 1
Notes: 1. No equivalent since the use of SubIdentity is deprecated since the only identity of a subscription is the SubName. See Deprecation.	

Request Update - Version 7 replacement

The Request Update command message contains a number of parameters. This should be replaced by using the MQSUBRQ verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Request Update command message to using MQSUBRQ.

Required parameters

MQPS_COMMAND with value MQPS_REQUEST_UPDATE is replaced by the use of the MQSUBRQ verb.

MQPS_TOPIC is implied by the use of the Hsub handle returned from the MQSUB call which is used as a parameter on the MQSUBRQ call.

Optional parameters

QMgrName, QName and StreamName are used in exactly the same way in Request Update command messages as they are in Register Subscriber command messages.

See “Register subscriber - Version 7 replacement” on page 44 for details of how to migrate the use of these fields.

See “Register subscriber - Version 7 replacement” on page 44 for details of how to migrate your application’s use of MQPS_REGISTRATION_OPTIONS in this command message.

For MQPS_STREAM_NAME see Chapter 7, “Publish/subscribe deprecated function,” on page 81.

MQPS_SUBSCRIPTION_NAME is implied by the use of the Hsub handle returned from the MQSUB call which is used as a parameter on the MQSUBRQ call.

WebSphere MQ publish/subscribe topology migration

This section contains topics that describe various scenarios for migration to WebSphere MQ Version 7.0 publish/subscribe.

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a WebSphere MQ Version 7.0 publish/subscribe cluster

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a Version 7.0 publish/subscribe cluster - all queue managers simultaneously

How to migrate an entire existing Websphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, to a Websphere Version 7.0 publish/subscribe cluster, migrating all queue managers at the same time.

To migrate the hierarchy, perform the following steps:

1. Install WebSphere MQ Version 7.0 on all of the computers that contain queue managers in the hierarchy, to upgrade all queue managers in the hierarchy to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** control command on each queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
3. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*.
4. Ensure that each queue manager is in the cluster by using the MQSC command DISPLAY CLUSQMGR(*), described in *WebSphere MQ Script (MQSC) Command Reference*. If a queue manager that should be in the cluster is not, then add it. For more information, refer to *WebSphere MQ Queue Manager Clusters*
5. To remove the hierarchical relationship on each child queue manager within the hierarchy, execute the following MQSC command: `ALTER QMGR PARENT(' ')`
6. Before proceeding to the next step, to confirm that all the hierarchical relationships have been cancelled, use the MQSC command `DISPLAY PUBSUB TYPE(ALL)` on each queue manager.
7. On one of the queue managers within the cluster, define one cluster topic by executing the following MQSC command: `ALTER TOPIC(<topic name>) PUBSCOPE(ALL) SUBSCOPE(ALL) CLUSTER(<cluster>)` Use a high-level topic, but not the root. For information about cluster topic naming, see Cluster topics.

Alternative procedure for i5/OS:

The following steps show an alternative procedure for WebSphere MQ for i5/OS, using CL commands and panels in place of MQSC commands.

1. Install WebSphere MQ Version 7.0 on all of the computers that contain queue managers in the hierarchy, to upgrade all queue managers in the hierarchy to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on each queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
3. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*

4. Ensure that each queue manager is in the cluster. If a queue manager that should be in the cluster is not, then add it.
5. Execute `WRKMQMPS PUBSUBNAME(<parent_queue_manager>)` to display the hierarchy.
6. On each child queue manager within the hierarchy, use **option 4=Remove** to detach from the parent, followed by **option 34=Work with Pub/Sub** to move down the sub-hierarchy. Repeat options 4 and 34 until no child queue managers are displayed.
7. Repeat step 6 for each child queue manager belonging to PUBSUBNAME(<parent_queue_manager>) until no child queue managers are displayed.
8. On one of the queue managers within the cluster, define at least one cluster topic by executing the following command: `CHGMQMTOP TOPNAME(<topic name>) PUBSCOPE(*ALL) SUBSCOPE(*ALL) CLUSTER(<cluster>) MQMNAME(<queue manager name>)` Use a high-level topic, but not the root. For information about cluster topic naming, see .

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a Version 7.0 publish/subscribe cluster - queue manager by queue manager

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, to a WebSphere Version 7.0 publish/subscribe cluster, one queue manager at a time.

To migrate the hierarchy, perform the following steps:

1. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*
2. Select the first queue manager to migrate into the publish/subscribe cluster. To cause the least disruption, select a queue manager that is a leaf node.
3. Install WebSphere MQ Version 7.0 on the computer that contains the selected queue manager in the hierarchy, to migrate the queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on this queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
5. Join this queue manager into the cluster.
6. If this queue manager has a hierarchical relationship to an existing member of the cluster, use the MQSC command `ALTER QMGR PARENT(' ')` at the child queue manager to cancel the relationship. Before proceeding to the next step, to confirm that the hierarchical relationship has been cancelled, use the MQSC command `DISPLAY PUBSUB TYPE(PARENT)` at the child queue manager.
7. If this is the first queue manager to be migrated into the publish/subscribe cluster, define at least one cluster topic by executing the MQSC command `ALTER TOPIC(<topic name="">) PUBSCOPE(ALL) SUBSCOPE(ALL) CLUSTER(<cluster>)`

   Note: Use a high-level topic, but not the root. For information about cluster topic naming, see Cluster topics in *WebSphere MQ Publish/Subscribe User's Guide*.
8. To migrate the remainder of the hierarchy without introducing loops, repeat recursively from step 3 for each child queue manager that is not already in the cluster, and repeat recursively from step 3 for each parent queue manager that is not already in the cluster.

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a WebSphere MQ Version 7.0 hierarchy

Migrating a WebSphere MQ Version 6.0 two queue manager publish/subscribe hierarchy to a Version 7.0 hierarchy - parent first

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, into a WebSphere Version 7.0 hierarchy, migrating the parent queue manager first.

To migrate the hierarchy, perform the following steps:

1. Install WebSphere MQ Version 7.0 on the computer that contains the parent queue manager in the hierarchy, to migrate the parent queue manager to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on the parent queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0. At this point you can either run as a mixed WebSphere MQ Version 6.0 and Version 7.0 hierarchy or continue with the migration by upgrading the child in the next step.
3. Install WebSphere MQ Version 7.0 on the computer that contains the child queue manager in the hierarchy, to migrate the child queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on the child queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0, and check the migration log to verify that the migration was successful. The migration log is in the queue manager directory: for example, /var/mqm/<QMgrName> on Linux or C:\Program Files\IBM\WebSphereMQ\qmgrs\<QMgrName> on Windows, unless you have specified otherwise on the command line.

Migrating a WebSphere MQ Version 6.0 two publish/subscribe queue manager hierarchy to a Version 7.0 hierarchy - child first

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, into a WebSphere Version 7.0 hierarchy, migrating the child queue manager first.

To migrate the hierarchy, perform the following steps:

1. Install WebSphere MQ Version 7.0 on the computer that contains the child queue manager in the hierarchy, to migrate the child queue manager to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on the child queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0. At this point you can either run as a mixed WebSphere MQ Version 6.0 and Version 7.0 hierarchy or continue with the migration by upgrading the parent in the next step.
3. Install WebSphere MQ Version 7.0 on the computer that contains the parent queue manager in the hierarchy, to migrate the parent queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on the parent queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0, and check the migration log to verify that the migration was successful. The migration log is in the queue manager directory: for example,

/var/mqm/<QMgrName> on Linux or C:\Program Files\IBM\WebSphereMQ\qmgrs\<QMgrName> on Windows, unless you have specified otherwise on the command line.

Chapter 5. Writing publish/subscribe applications

Message ordering

For a given topic, messages are published by the queue manager in the same order as they are received from publishing applications (subject to reordering based on message priority). This normally means that each subscriber receives messages from a particular queue manager, on a particular topic, 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 queue manager 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.

Intercepting publications

It may be useful to be able to intercept a publication before it reaches a subscriber so that you can:

- attach additional information to the message
- block messages
- transform messages

You can perform the same operation on each message or vary the operation depending on something in the message or message header.

Once intercepted, the message can be passed on to an application capable or doing message transformation.

To intercept a publication use the subscription level subscription attribute. The interceptor is simply another subscriber to the topic the messages it wishes to intercept are being published on. The interceptor then republishes the message so that is can be received by other subscribers.

To ensure the interceptor receives the messages before any other subscribers, make sure it has the highest subscription level of all subscribers by using the *SubLevel* field in the MQSD. By default, subscribers have a *SubLevel* of 1.

- If you have one intercepting subscriber it should be configured to subscribe at a *SubLevel* of 9.
- If more than one intercepting application is required to receive the publication, set the sublevel of each iterceptor's subscription appropriately to determine the order in which these intercepting applications receive the publication.

The interceptor with the highest subscription level will receive the publication first, after which it will be republished and will be received by the subscription with the next highest subscription level and so on. When configuring multiple intercepting applications, there should be no more than one at each *SubLevel* value which republishes the message; otherwise duplicate publications will be sent to the final set of subscribing applications because more than one interceptor will republish the message to the next *SubLevel* down.

By default, applications will publish to a topic using a *PubLevel* of 9. *PubLevel* is a field in the MQPMO. If there are any subscriptions with a *SubLevel* of 9, only those subscriptions will be given a copy of the publication. If there are no subscriptions with a *SubLevel* of 9, the publications are given to all those subscriptions on this topic which have the highest *SubLevel*.

An intercepting application should make its subscription using the options described in Table 5.

Table 5. Intercepting subscriber options

Subscription option	Notes
MQSO_SET_CORREL_ID and *SubCorrelId* set to MQCI_NONE	This ensures that the *CorrelId* in the publication message when it is re-published by the interceptor, is the one set by the original publisher, in case any subscriptions at a lower *SubLevel* has requested that.
PubPriority set to MQPRI_PRIORITY_AS_PUBLISHED	This ensures that the *Priority* in the publication message when it is re-published by the interceptor, is the one set by the original publisher, in case any subscriptions at a lower *SubLevel* has requested that.

An intercepting application should process the publication message (for example, transform or encrypt it) and then re-publish it to the same topic at a publication level one lower than the subscription level which intercepted it. For example, a subscription with a *SubLevel* of 9 should republish the message with a *PubLevel* of 8. In order to republish the message correctly, several pieces of information are required as shown in Table 6the table below, and the intercepting application should use the same MQMD as in the original message and use MQPMO_PASS_ALL_CONTEXT to ensure all information in that MQMD is preserved and passed on to the next application (ordinary subscriber or interceptor).

Table 6. MQMD values for republished messages

Republish message using MQPUT	Information in publication message
MQOD.ObjectString	Message property MQTopicString
MQPMO.Options should OR with the information in the message	Message property MQPubOptions

An intercepting subscriber is a normal subscriber and as such can use any of the normal publish/subscribe or WebSphere MQ functions.

A maximum of 8 intercepting applications can be implemented (with sub levels from 9 down to 2 inclusive). In this case the final recipient of the message will have a subscription level of 1.

You can have a subscriber with sub level 0 that serves as a catch all if no one else is interested in the message. This configuration can be useful because you can monitor the messages this subscriber receives and check why no other subscribers received it and whether it is correct that it not be received by anyone else.

Retained publications

If the publication is put by the original application with Put-message option MQPMO_RETAIN, it will only be retained if it is received by a subscriber whose sub level is 1 or 0. In order to ensure that the instruction to retain this publication is preserved as the publication passes through an intercepting application, the MQPMO options are carried with the publication as a message property and must be used on the republishing MQPUT call by the intercepting application.

Publishing options

Several options are available that control the way messages are published.

Withholding reply-to information from subscribers

If you do not want subscribers to be able to reply to publications they receive, it is possible to withhold information in the ReplyToQ and ReplyToQgr fields of the MQMD by using the MQPMO_SUPPRESS_REPLYTO put-message option. If this option is used, the queue manager removes that information from the MQMD when it receives the publication before forwarding it to any subscribers.

This option cannot be used in combination with a report option that needs a ReplyToQ, if this is attempted the call with fail with MQRC_MISSING_REPLY_TO_Q.

Publication level

Using publication levels is a way of controlling which subscribers receive the publication. The publication level denotes the level of subscription targeted by the publication. Only subscriptions with the highest subscription level less than or equal to the publication's publication level, will receive the publication. This value must be in the range zero to nine; zero is the lowest publication level. The initial value of this field is 9. One of the uses of publication and subscription levels is to intercept publications.

Subscription options

Subscriptions and message persistence

Queue managers maintain the persistence of the publications they forward to subscribers as set by the publisher, unless changed by options specified when the subscription is registered. These options are:

- Nonpersistent
- Persistent
- Persistence as queue
- Persistence as publisher (the default)

The system administrator can determine which users are allowed to have publications sent persistently.

Subscriptions and retained publications

To control when retained publications are received, subscribers can use two subscription options.

Publish on request only, MQSO_PUBLICATIONS_ON_REQUEST

If you want a subscriber to have control of when it receives publications you can use the MQSO_PUBLICATIONS_ON_REQUEST subscription option. A subscriber can then control when it receives publications by using the MQSUBRQ call (specifying the Hsub handle that was returned from the original MQSUB call) to request that it is sent a topic's retained publication. Subscribers using the MQSO_PUBLICATIONS_ON_REQUEST subscription option, do not receive any non-retained publications.

If a subscriber uses the MQSUBRQ call and uses wildcards in the subscription's topic, the subscription might match multiple topics or nodes on a topic tree, all of whose retained messages (if any exist) will be sent to the subscriber.

This option can be particularly helpful when used with durable subscriptions because a queue manager will continue to send publications to a subscriber if it subscribed durably even if that subscriber application is not running. This could lead to a buildup of messages on the subscriber queue. This build up can be avoided if the subscriber registers using the MQSO_PUBLICATIONS_ON_REQUEST option. Alternatively, you can use non-durable subscriptions if appropriate to your application to avoid a build up of unwanted messages.

If a subscription is durable and a publisher uses retained publications the subscriber application can use the MQSUBRQ call to refresh its state information after a restart. The subscriber must then refresh its state periodically using the MQSUBRQ call.

No publications will be sent as a result of the MQSUB call using this option. A durable subscription that has been resumed following disconnection will use the MQSO_PUBLICATIONS_ON_REQUEST option if the original subscription was configured to use this option.

New publications only, MQSO_NEW_PUBLICATIONS_ONLY

If a retained publication exists on a topic, any subscribers that make a subscription after the publication was made will receive a copy of that publication. If a subscriber does not want to receive any publications that were made prior to the subscription being made, the subscriber can use the MQSO_NEW_PUBLICATIONS_ONLY subscription option.

Grouping subscriptions

If multiple subscriptions with identical subscription levels, use the same subscriber queue and correlation ID and subscribe to overlapping topic strings, you can group them using the MQSO_GROUP_SUB option.

If you group subscriptions, only one copy of a publication is sent to the subscriber queue. If this option is not used, then each unique subscription (identified by SubName) that has registered an interest in a topic are provided with a copy of the publication which will can result in more than one copy of the publication being placed on the subscriber queue shared by a number of subscriptions.

In a subscription group, the subscription with the most significant subscription is provided with the publication. The most significant subscription is based on the full topic name up to the point where a wild card is found. If a mixture of wildcard schemes is used within the group, only the position of the wild card is important. You are advised not to combine different wild card schemes within a group of subscriptions that share the same queue.

If a subscribing application has the most significant subscription in group and uses the MQSO_NOT_OWN_PUBS option, and this same application publishes a message that matches the subscription, no publication is delivered to the subscriber queue.

When creating a new grouped subscription it must still have a unique SubName, however if the SubName matches the full topic name of an existing subscription in the group, the call will fail with MQRC_DUPLICATE_GROUP_SUB.

When altering a grouped subscription, the fields that imply the grouping, Hobj on the MQSUB call (representing the queue and queue manager name), and the SubCorrelId cannot be changed. Attempting to alter these fields will cause the call to fail with MQRC_GROUPING_NOT_ALTERABLE.

The group subscription option must be combined with MQSO_SET_CORREL_ID with either a user provided SubCorrelId or MQCI_NONE.

The group subscription option cannot be combined with managed destinations (MQSO_MANAGED).

Publish/subscribe message properties

Put your short description here; used for first paragraph and abstract.

PubAccountingToken

This is the value that will be in the AccountingToken field of the Message Descriptor (MQMD) of all publication messages matching this subscription. AccountingToken is part of the identity context of the message. For more information about message context, see the *WebSphere MQ Application Programming Guide*. For more information about the AccountingToken field in the MQMD, see the *WebSphere Application Programming Reference*.

PubApplIdentityData

This is the value that will be in the ApplIdentityData field of the Message Descriptor (MQMD) of all publication messages matching this subscription. ApplIdentityData is part of the identity context of the message. For more information about message context, see the *WebSphere MQ Application Programming Guide*. For more information about the ApplIdentityData field in the MQMD, see the *WebSphere MQ Application Programming Reference*.

If the option MQSO_SET_IDENTITY_CONTEXT is not specified, the ApplIdentityData which will be set in each message published for this subscription is blanks, as default context information.

If the option MQSO_SET_IDENTITY_CONTEXT is specified, the PubApplIdentityData is being generated by the user and this field is an input field

which contains the ApplIdentityData to be set in each publication for this subscription.

PubPriority

This is the value that will be in the Priority field of the Message Descriptor (MQMD) of all publication messages matching this subscription. For more information about the Priority field in the MQMD, see the *WebSphere MQ Application Programming Reference*.

The value must be greater than or equal to zero; zero is the lowest priority. The following special values can also be used:

- MQPRI_PRIORITY_AS_Q_DEF - When a subscription queue is provided in the Hobj field in the MQSUB call, and is not a managed handle, then the priority for the message is taken from the DefProirity attribute of this queue. If the queue so identified is a cluster queue or there is more than one definition in the queue-name resolution path then the priority is determined when the publication message is put to the queue as described for Priority in the MQMD in the *WebSphere MQ Application Programming Reference*. If the MQSUB call uses a managed handle, the priority for the message is taken from the DefPriority attribute of the model queue associated with the topic subscribed to.
- MQPRI_PRIORITY_AS_PUBLISHED - The priority for the message is the priority of the original publication. This is the initial value of this field.

SelectionString

This variable length field will be returned on output from an MQSUB call using the MQSO_RESUME option, if a big enough buffer is provided. If the buffer provided on the call is not big enough (by the value in VSBufSize) only the length of the selection string will be returned in the VSLength field of the MQCHARV and the contents of the buffer will not be altered. The MQSUB call with the MQSO_RESUME option can then be issued again providing a buffer long enough to fit VSLength bytes.

SubCorrelId

All publications sent to match this subscription will contain this correlation identifier in the message descriptor. If multiple subscriptions use the same queue to get their publications from, using MQGET by correlation id allows only publications for a specific subscription to be obtained. This correlation identifier can either be generated by the queue manager or by the user.

If the option MQSO_SET_CORREL_ID is not specified, the correlation identifier is generated by the queue manager and this field is an output field which contains the correlation identifier which will be set in each message published for this subscription.

If the option MQSO_SET_CORREL_ID is specified, the correlation identifier is being generated by the user and this field is an input field which contains the correlation identifier to be set in each publication for this subscription. In this case, if the field contains MQCI_NONE, the correlation identifier which will be set in each message published for this subscription will be the correlation identifier created by the original put of the message.

If the option MQSO_GROUP_SUB is specified and the correlation identifier specified is the same as an existing grouped subscription using the same queue and an overlapping topic string, only the most significant subscription in the group is provided with a copy of the publication.

SubUserData

This is the subscription user data. The data provided on the subscription in this field will be included as the MQSubUserData message property of every publication sent to this subscription.

Chapter 6. WebSphere MQ publish/subscribe security

Example publish/subscribe security setup

This section describes a scenario that has access control setup on topics in a way that allows the security control to be applied as required.

Grant access to a user to subscribe to a topic

This topic is the first one in a list of tasks that tells you how to grant access to topics by more than one user.

This task assumes that no administrative topic objects exist, nor have any profiles been defined for subscription or publication. The applications are creating new subscriptions, rather than resuming existing ones, and are doing so using the topic string only.

An application can make a subscription by providing a topic object, or a topic string, or a combination of both. Whichever way the application selects, the effect is to make a subscription at a certain point in the topic tree. If this point in the topic tree is represented by an administrative topic object, a security profile is checked based on the name of that topic object.

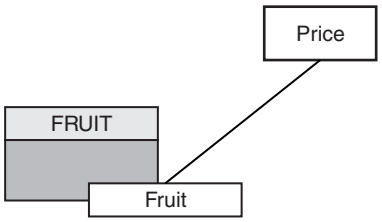


Figure 16. Topic object access example

Table 7. Example topic object access

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Fruit	USER1	FRUIT

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(FRUIT) TOPICSTR('Price/Fruit')`.
2. Grant access as follows:
   a. z/OS. Grant access to `USER1` to subscribe to topic `"Price/Fruit"` by granting the user access to the `hlq.SUBSCRIBE.FRUIT` profile. Do this, using the following RACF® commands:

      ```
      RDEFINE MXTOPIC hlq.SUBSCRIBE.FRUIT UACC(NONE)
      PERMIT hlq.SUBSCRIBE.FRUIT CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      ```

   b. Other platforms. Grant access to `USER1` to subscribe to topic `"Price/Fruit"` by granting the user access to the `FRUIT` profile. Do this, using the following setmqaut command:

      ```
      setmqaut -t topic -n FRUIT -p USER1 +sub
      ```

When USER1 attempts to subscribe to topic "Price/Fruit" the result is success.

When USER2 attempts to subscribe to topic "Price/Fruit" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit"
```

Note that this is an illustration of what you see; not all the fields.

Grant access to a user to subscribe to a topic deeper within the tree

This topic is the second in a list of tasks that tells you how to grant access to topics by more than one user.

This topic uses the setup described in "Grant access to a user to subscribe to a topic" on page 61.

If the point in the topic tree where the application makes the subscription is not represented by an administrative topic object, move up the tree until the closest parent administrative topic object is located. The security profile is checked, based on the name of that topic object.

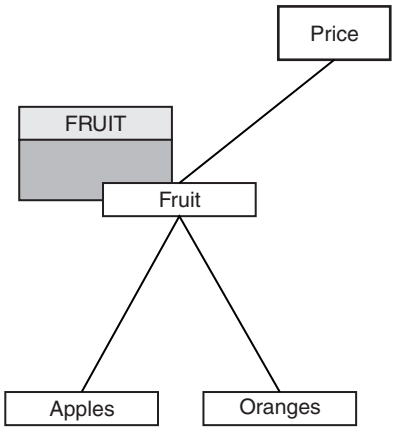


Figure 17. Example of granting access to a topic within a topic tree

Table 8. Access requirements for example topics and topic objects

Topic	Subscribe access required	Topic object
Price	No user	None

Table 8. Access requirements for example topics and topic objects (continued)

Topic	Subscribe access required	Topic object
Price/Fruit	USER1	FRUIT
Price/Fruit/Apples	USER1	
Price/Fruit/Oranges	USER1	

In the previous task USER1 was granted access to subscribe to topic "Price/Fruit/Apples" by granting it access to the hlq.SUBSCRIBE.FRUIT profile on z/OS and subscribe access to the FRUIT profile on other platforms. This single profile also grants USER1 access to subscribe to "Price/Fruit/Oranges" and "Price/Fruit/#".

When USER1 attempts to subscribe to topic "Price/Fruit/Apples" the result is success.

When USER2 attempts to subscribe to topic "Price/Fruit/Apples" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Apples"
```

Note the following:

- The messages you receive on z/OS are identical to those received in the previous task as the same topic objects and profiles are controlling the access.
- The event message you receive on other platforms is similar to the one received in the previous task, but the actual topic string is different.

Grant another user access to subscribe to only the topic deeper within the tree

This topic is the third in a list of tasks that tells you how to grant access to subscribe to topics by more than one user.

This topic uses the setup described in “Grant access to a user to subscribe to a topic deeper within the tree” on page 62.

In the previous task USER2 was refused access to topic "Price/Fruit/Apples". This topic tells you how to grant access to that topic, but not to any other topics.

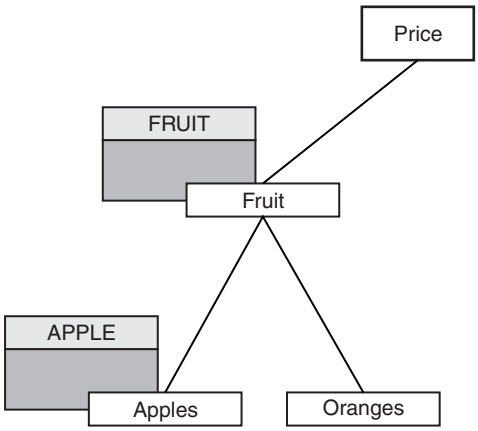


Figure 18. Granting access to specific topics within a topic tree

Table 9. Access requirements for example topics and topic objects

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Fruit	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2	APPLE
Price/Fruit/Oranges	USER1	

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(APPLE) TOPICSTR('Price/Fruit/Apples')`.
2. Grant access as follows:
   a. z/OS.

      In the previous task `USER1` was granted access to subscribe to topic `"Price/Fruit/Apples"` by granting the user access to the `hlq.SUBSCRIBE.FRUIT` profile.

      This single profile also granted `USER1` access to subscribe to `"Price/Fruit/Oranges"` `"Price/Fruit/#"` and this access remains even with the addition of the new topic object and the profiles associated with it.

      Grant access to `USER2` to subscribe to topic `"Price/Fruit/Apples"` by granting the user access to the `hlq.SUBSCRIBE.APPLE` profile. Do this, using the following RACF commands:

      ```
      RDEFINE MXTOPIC hlq.SUBSCRIBE.APPLE UACC(NONE)
      PERMIT hlq.SUBSCRIBE.FRUIT APPLE(MXTOPIC) ID(USER2) ACCESS(ALTER)
      ```

   b. Other platforms.

      In the previous task `USER1` was granted access to subscribe to topic `"Price/Fruit/Apples"` by granting the user subscribe access to the `FRUIT` profile.

      This single profile also granted `USER1` access to subscribe to `"Price/Fruit/Oranges"` and `"Price/Fruit/#"`, and this access remains even with the addition of the new topic object and the profiles associated with it.

      Grant access to `USER2` to subscribe to topic `"Price/Fruit/Apples"` by granting the user subscribe access to the `APPLE` profile. Do this, using the following setmqaut command:

      ```
      setmqaut –t topic –n APPLE –p USER2 +sub
      ```

On z/OS, when USER1 attempts to subscribe to topic "Price/Fruit/Apples" the first security check on the hlq.SUBSCRIBE.APPLE profile fails, but on moving up the tree the hlq.SUBSCRIBE.FRUIT profile allows USER1 to subscribe, so the subscription succeeds and no return code is sent to the MQSUB call. However, a RACF ICH message is generated for the first check:

```
ICH408I USER(USER1   ) ...
  hlq.SUBSCRIBE.APPLE ...
```

When USER2 attempts to subscribe to topic "Price/Fruit/Apples" the result is success because the security check passes on the first profile.

When USER2 attempts to subscribe to topic "Price/Fruit/Oranges" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Oranges"
```

The disadvantage of this setup is that, on z/OS, you receive additional ICH messages on the console. You can avoid this if you secure the topic tree in a different manner.

Change access control to avoid additional messages

This topic is the fourth in a list of tasks that tells you how to grant access to subscribe to topics by more than one user and to avoid additional RACF ICH408I messages on z/OS.

This topic enhances the setup described in “Grant another user access to subscribe to only the topic deeper within the tree” on page 63 so that you avoid additional error messages.

This topic tells you how to grant access to topics deeper in the tree, and how to remove access to the topic lower down the tree when no user requires it.

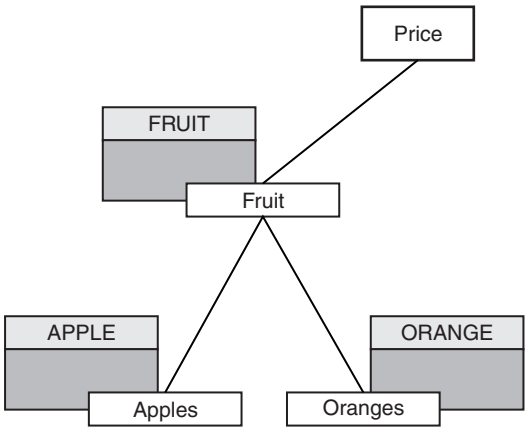


Figure 19. Example of granting access control to avoid additional messages.

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(ORANGE) TOPICSTR('Price/Fruit/Oranges')`.
2. Grant access as follows:
   a. z/OS.

      Define a new profile and add access to that profile, and the existing profiles. Do this, using the following RACF commands:

      ```
      RDEFINE MXTOPIC hlq.SUBSCRIBE.ORANGE UACC(NONE)
      PERMIT hlq.SUBSCRIBE.ORANGE CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      PERMIT hlq.SUBSCRIBE.APPLE CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      ```

   b. Other platforms.

      Setup the equivalent access by using the following setmqaut commands:

      ```
      setmqaut –t topic –n ORANGE –p USER1 +sub
      setmqaut -t topic -n APPLE -p USER1 +sub
      ```

On z/OS, when `USER1` attempts to subscribe to topic `"Price/Fruit/Apples"` the first security check on the `hlq.SUBSCRIBE.APPLE` profile succeeds.

Similarly, when `USER2` attempts to subscribe to topic `"Price/Fruit/Apples"` the result is success because the security check passes on the first profile.

When `USER2` attempts to subscribe to topic `"Price/Fruit/Oranges"` the result is failure with an `MQRC_NOT_AUTHORIZED` message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

  ```
  ICH408I USER(USER2   ) ...
    hlq.SUBSCRIBE.ORANGE ...

  ICH408I USER(USER2   ) ...
    hlq.SUBSCRIBE.FRUIT ...

  ICH408I USER(USER2   ) ...
    hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
  ```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   ORANGE, FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Oranges"
```

Grant access to a user to publish to a topic

This topic is the first one in a list of tasks that tells you how to grant access to publish topics by more than one user.

This task assumes that no administrative topic objects exist on the right hand side of the topic tree, nor have any profiles been defined for publication. The assumption used is that publishers are using the topic string only.

An application can publish to a topic by providing a topic object, or a topic string, or a combination of both. Whichever way the application selects, the effect is to publish at a certain point in the topic tree. If this point in the topic tree is represented by an administrative topic object, a security profile is checked based on the name of that topic object. For example:

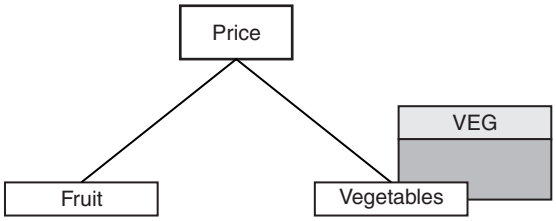


Figure 20. Granting publish access to a topic

Table 10. Example publish access requirements

Topic	Publish access required	Topic object
Price	No user	None
Price/Vegetables	USER1	VEG

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(VEG) TOPICSTR('Price/Vegetables')`.
2. Grant access as follows:
   a. z/OS. Grant access to `USER1` to publish to topic `"Price/Vegetables"` by granting the user access to the `hlq.PUBLISH.VEG` profile. Do this, using the following RACF commands:

      ```
      RDEFINE MXTOPIC hlq.PUBLISH.VEG UACC(NONE)
      PERMIT hlq.PUBLISH.VEG CLASS(MXTOPIC) ID(USER1) ACCESS(UPDATE)
      ```

   b. Other platforms. Grant access to `USER1` to publish to topic `"Price/Vegetables"` by granting the user access to the `VEG` profile. Do this, using the following setmqaut command:

      ```
      setmqaut –t topic –n VEG –p USER1 +pub
      ```

When `USER1` attempts to publish to topic `"Price/Vegetables"` the result is success; that is, the MQOPEN call succeeds.

When `USER2` attempts to publish to topic `"Price/Vegetables"` the MQOPEN call fails with an `MQRC_NOT_AUTHORIZED` message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.PUBLISH.VEG ...

ICH408I USER(USER2   ) ...
  hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   VEG, SYSTEM.BASE.TOPIC
TopicString     "Price/Vegetables"
```

Note that this is an illustration of what you see; not all the fields.

Grant access to a user to publish to a topic deeper within the tree

This topic is the second in a list of tasks that tells you how to grant access to publish to topics by more than one user.

This topic uses the setup described in "Grant access to a user to publish to a topic" on page 67.

If the point in the topic tree where the application publishes is not represented by an administrative topic object, move up the tree until the closest parent administrative topic object is located. The security profile is checked, based on the name of that topic object.

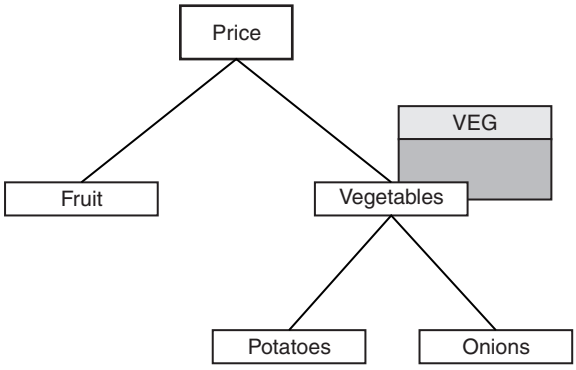


Figure 21. Granting publish access to a topic within a topic tree

Table 11. Example publish access requirements

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Vegetables	USER1	VEG
Price/Vegetables/Potatoes	USER1	
Price/Vegetables/Onions	USER1	

In the previous task USER1 was granted access to publish topic "Price/Vegetables/Potatoes" by granting it access to the hlq.PUBLISH.VEG profile on z/OS or publish access to the VEG profile on other platforms. This single profile also grants USER1 access to publish at"Price/Vegetables/Onions" and at "Price/Fruit/#".

When USER1 attempts to publish at topic "Price/Vegetables/Potatoes" the result is success; that is the MQOPEN call succeeds.

When USER2 attempts to subscribe to topic "Price/Vegetables/Potatoes" the result is failure; that is, the MQOPEN call fails with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.PUBLISH.VEG ...

ICH408I USER(USER2   ) ...
  hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   VEG, SYSTEM.BASE.TOPIC
TopicString     "Price/Vegetables/Potatoes"
```

Note the following:

- The messages you receive on z/OS are identical to those received in the previous task as the same topic objects and profiles are controlling the access.
- The event message you receive on other platforms is similar to the one received in the previous task, but the actual topic string is different.

Grant access for publish and subscribe

This topic is the last in a list of tasks that tells you how to grant access to publish and subscribe to topics by more than one user.

This topic uses the setup described in “Grant access to a user to publish to a topic deeper within the tree” on page 68.

In a previous task USER1 was given access to subscribe to the topic "Price/Fruit". This topic tells you how to grant access to that user to publish to that topic.

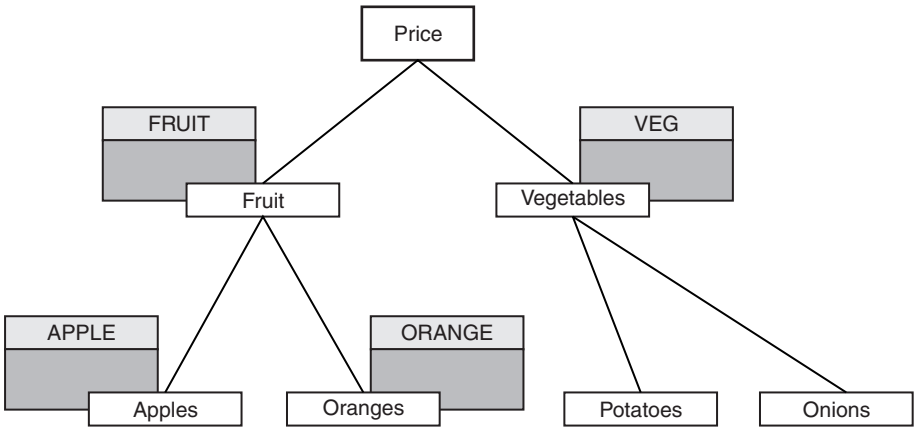


Figure 22. Granting access for publishing and subscribing

Table 12. Example publishing and subscribing access requirements

Topic	Subscribe access required	Publish access required	Topic object
Price	No user	No user	None
Price/Fruit	USER1	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2		APPLE
Price/Fruit/Oranges	USER1		ORANGE

Grant access as follows:

1. z/OS.

   In an earlier task USER1 was granted access to subscribe to topic "Price/Fruit" by granting the user access to the hlq.SUBSCRIBE.FRUIT profile.

   In order to publish to the "Price/Fruit" topic, grant access to USER1 to the hlq.PUBLISH.FRUIT profile. Do this, using the following RACF commands:

   ```
   RDEFINE MXTOPIC hlq.PUBLISH.FRUIT UACC(NONE)
   PERMIT hlq.PUBLISH.FRUIT CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
   ```

2. Other platforms.

   Grant access to USER1 to publish to topic "Price/Fruit" by granting the user publish access to the FRUIT profile. Do this, using the following setmqaut command:

   ```
   setmqaut –t topic –n FRUIT –p USER1 +pub
   ```

On z/OS, when USER1 attempts to publish to topic "Price/Fruit" the security check on the MQOPEN call passes.

When USER2 attempts to publish at topic "Price/Fruit" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

  ```
  ICH408I USER(USER2   ) ...
    hlq.PUBLISH.FRUIT ...

  ICH408I USER(USER2   ) ...
    hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
  ```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit"
```

Following the complete set of these tasks, gives USER1 and USER2 the following access authorities for publish and subscribe to the topics listed:

Table 13. Complete list of access authorities resulting from security examples

Topic	Subscribe access required	Publish access required	Topic object
Price	No user	No user	None
Price/Fruit	USER1	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2		APPLE
Price/Fruit/Oranges	USER1		ORANGE
Price/Vegetables		USER1	VEG
Price/Vegetables/ Potatoes			
Price/Vegetables/ Onions			

Where you have different requirements for security access at different levels within the topic tree, careful planning ensures that you do not receive extraneous security warnings on the z/OS console log. Setting up security at the correct level within the tree avoids misleading security messages.

Topic objects

This section gives an overview of the WebSphere MQ topic object type and explains the concepts used for authorization when using this type of object.

A topic has a 48 character name used for administration, and a hierarchically organized string of unlimited length. The topic string is of the form:

```
<topic name A>/<topic name B> ...
```

The topic string of each topic object is used to create a tree, with each node in the tree representing a topic name segment of the full topic string.

When an application needs to subscribe to a topic, the user ID associated with the application must be authorized to perform a subscribe operation on the specified topic. The topic can be specified in the form of a topic string, or as the MQCHAR48 name of a topic object.

It is possible that, when an application subscribes by specifying a topic string, that no corresponding node in the topic tree exists at the time of subscription. Note that the topic string specified by the subscribing application can contain wildcard characters.

At the time of subscription, an authority check is made to ensure that the same user ID is authorized to put to the specified destination queue. This is because when messages are published to that destination, they are put using the user context associated with the application. If the destination is a managed queue, no

security checks are performed against the managed destination. It is assumed that if you are allowed to subscribe to that topic you can use managed destinations.

If the application is re-subscribing to an existing subscription, the security checks made are the same as those that would occur if the application was creating a new subscription. For example, a subscribe security check and a destination queue security check take place.

To publish on a topic, an application first performs an MQOPEN command of the topic on which they want to publish. Messages are then published using the MQPUT command.

The authority check to publish is performed when the application performs the open command. The application must be authorized to publish to the specified topic, which can be specified in the form of a topic string, or as the MQCHAR48 name of a topic object. As with a subscribe operation, it is possible that no corresponding node in the topic tree exists at the time of the MQOPEN call.

Security model

Only defined topic objects, that are specified as administration nodes, have associated security attributes. These attributes specify whether a specified user ID, or security group, is permitted to perform a subscribe or a publish operation on each topic object.

The security attributes are associated with the appropriate administration node in the topic tree. When an authority check is made for a particular user ID during a subscribe or publish operation, the authority granted is based on a set of rules dependent on the security attributes associated with the appropriate topic tree nodes.

You can represent the security attributes as an access control list, thereby indicating what authority a particular operating system user ID, or security group, has to the topic object.

Consider the following example where the topic objects have been defined with the security attributes, or authorities shown:

Table 14. Example topic object authorities

Topic name	Topic string	Authorities - not z/OS	z/OS authorities
SECROOT	SEC	none	NONE
SECGOOD	SEC/GOOD	usr1+subscribe	ALTER Hlq.SUBSCRIBE.SECGOOD
SECBAD	SEC/BAD	none	NONE Hlq.SUBSCRIBE.SECBAD
SECCOMB	SEC/COMB	none	NONE Hlq.SUBSCRIBE.SECCOMB
SECCOMBB	SEC/COMB/ GOOD/BAD	none	NONE Hlq.SUBSCRIBE.SECCOMBB

Table 14. Example topic object authorities (continued)

Topic name	Topic string	Authorities - not z/OS	z/OS authorities
SECCOMBG	SEC/COMB/GOOD	usr2+subscribe	NONE Hlq.SUBSCRIBE.SECCOMBG
SECCOMBN	SEC/COMB/BAD	none	NONE Hlq.SUBSCRIBE.SECCOMBN

The examples listed give the following authorizations:

- At the root of the tree `/SEC` no user has authority at that node.
- `usr1` has been granted subscribe authority to the object `/SEC/GOOD`
- `usr2` has been granted subscribe authority to the object `/SEC/COMB/GOOD`

The topic tree with the associated security attributes at each node can be represented as:

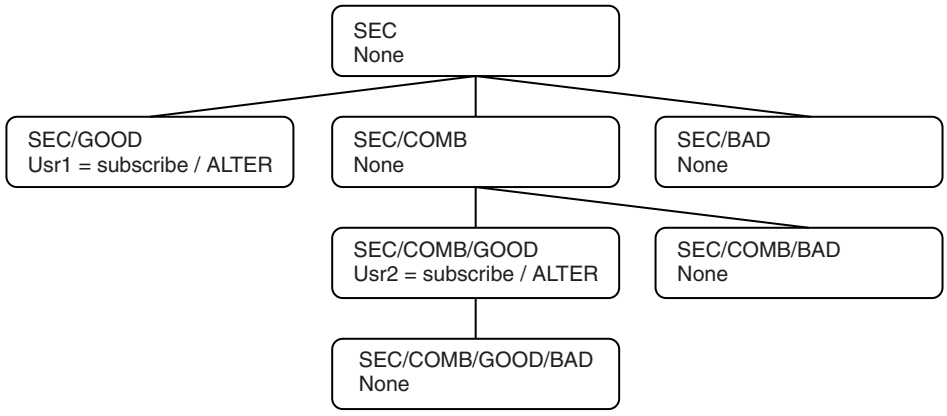


Figure 23. Example topic tree security attributes

Subscribing using the topic object name

When subscribing to a topic object by specifying the MQCHAR48 name, the corresponding node in the topic tree is located. If the security attributes associated with the topic node indicate that the user has authority to subscribe, then access is granted.

If not, the parent node in the tree is considered to determine if the user has authority to subscribe to that node. If so, then access is also granted. If not, then the parent of that node is considered, and so on, until a node is located that grants subscribe authority to the user, or until the root node is considered without authority having been granted. In the latter case, access is denied.

This means that if any node in the path grants authority to subscribe to that user or application, the subscriber is allowed to subscribe at that node, or anywhere below that node in the topic tree.

The root node in the above example is `SEC`; note, it is possible that the root node will always be a topic object.

The security attributes indicate that a particular user ID has subscribe authority, if the access control list indicates that the user ID itself has authority, or that an operating system security group of which the user ID is a member has authority.

So, for example:

- If `usr1` tries to subscribe, using a topic string of `SEC/GOOD`, the subscription would be allowed as the user ID has access to the node associated with that topic. However, if `usr1` tried to subscribe using topic string `SEC/COMB/GOOD` the subscription would not be allowed as the user ID does not have access to the node associated with it.
- If `usr2` tries to subscribe, using a topic string of `SEC/COMB/GOOD` the subscription would be allowed to as the user ID has access to the node associated with the topic. However, if `usr2` tried to subscribe to `SEC/GOOD` the subscription would not be allowed as the user ID does not have access to the node associated with it.
- If `usr2` tries to subscribe using a topic string of `SEC/COMB/GOOD/BAD` the subscription would be allowed to because the user ID has access to the parent node `SEC/COMB/GOOD`.
- If `usr1` or `usr2` tries to subscribe using a topic string of `/SEC/COMB/BAD`, neither would be allowed as they do not have access to the topic node associated with it, or the parent nodes of that topic.

A subscribe operation specifying a topic object name in the case where the topic object does not exist results in an MQRC_UNKNOWN_OBJECT_NAME error.

Subscribing using a topic string where the topic node exists

The behavior is the same as when specifying the topic by the MQCHAR48 object name.

Subscribing using a topic string where the topic node does not exist

When an application subscribes specifying a topic string representing a topic node that does not currently exist in the topic tree, the authority check is performed as outlined in the previous section, starting with the parent node of that which is represented by the topic string. If the authority is granted, a new node representing the topic string is created in the topic tree.

For example, if `usr1` tries to subscribe to a topic `SEC/GOOD/NEW`, this would be allowed as `usr1` has access to the parent node `SEC/GOOD` and a new topic node is created in the tree as the following diagram shows. As this is not a topic object it does not have any security attributes associated with it directly; the attributes are inherited from its parent.

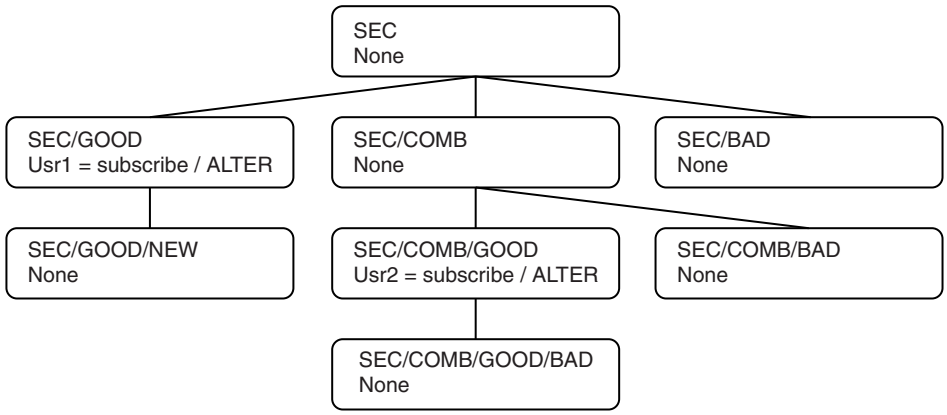


Figure 24. Example topic tree security attributes

Subscribing using a topic string that contains wildcard characters

When an application attempts to connect by specifying a topic string that contains a wildcard character, the authority check is made against the node in the topic tree that matches the fully qualified part of the topic string.

So, if an application needs to subscribe to `SEC/COMB/GOOD/*`, an authority check is carried out as outlined in the previous two sections on the node `SEC/COMB/GOOD` in the topic tree.

Similarly, if an application needs to subscribe to `SEC/COMB/*/GOOD`, an authority check is carried out on the node `SEC/COMB`.

Authority to destination queues

When subscribing to a topic, one of the parameters on the call is the handle `hobj`, this is:

- The `hobj` of a queue that has been opened for output to receive the publications.
- `Blank` and:
  - The MQSO_MANAGED option has been specified, and
  - The subscription does not exist, and
  - Create is specified,

  in which case a managed queue is created.
- `Blank`, and you are altering or resuming an existing subscription, in which the destination queue can be either a managed or non managed queue.

In each case the application or user making the MQSUB request, has to have the authority to put messages to that destination queue it has provided; in effect authority to have published messages put on that queue, in order for the subscribe request to continue. This follows the existing rules for queue security checking.

This includes Alternate user ID and Context security checks where required. In order to be able to set any of the Identity context fields you have to specify the MQSO_SET_IDENTITY_CONTEXT option as well as the MQSO_CREATE or MQSO_ALTER option. You are not allowed to set any of the Identity context fields on an MQSO_RESUME request.

If the destination is a managed queue, no security checks are performed against the managed destination. If you are allowed to subscribe to that topic the assumption is that you can use managed destinations.

Publish using the topic name or topic string where the topic node exists

The security model for the publish operation is the same as that for the subscribe operation, except that there is no wildcard character in the topic string case to consider.

The authorities to publish and subscribe are distinct; a user or group can have one authority without necessarily having the other.

When publishing to a topic object by specifying either the MQCHAR48 name or the topic string, the corresponding node in the topic tree is located. If the security attributes associated with the topic node indicates that the user has authority to publish, then access is granted.

If not, then the parent node in the tree is considered to determine if the user has authority to publish to that node. If so, then access is also granted. If not, then the parent of that node is considered, and so on until a node is located which grants publish authority to the user, or until the root node is considered without authority having been granted. In the latter case, access is denied.

This means that if any node in the path grants authority to publish to that user or application, the publisher is allowed to publish at that node or anywhere below that node in the topic tree.

Publish using the topic name or topic string where the topic node does not exist

As with the subscribe operation, when an application publishes, specifying a topic string representing a topic node that does not currently exist in the topic tree, the authority check is performed starting with the parent node of that which is represented by the topic string. If the authority is granted, a new node representing the topic string is created in the topic tree.

Publish using an alias queue that resolves to a topic object

If you publish, using an alias queue that resolves to a topic object then security checking occurs on both the open of the alias queue that you specify and the underlying topic to which it resolves.

The security check on the alias queue looks to see that the user has authority to put messages to that alias queue and the security check on the topic looks to see that the user can Publish to that topic. This is different behavior from that which takes place when an alias queue resolves to other queues.

Closing a subscription

There is additional security checking if you close a subscription using the MQCO_ SUB_ REMOVE option and you did not create the subscription under this handle.

A security check is performed to ensure that you have the correct authority to do this as the action results in the removal of the subscription.

A similar process to that used to determine if a user has the correct level of authority to subscribe to a topic is followed to determine if the user has the correct level of authority required to perform the close remove request. If the security attributes associated with the topic node indicate that the user has authority , then access is granted. If not, then the parent node in the tree is considered to determine if the user has authority to close remove that subscription and so on until either authority is granted or the root node is reached.

Note that the security check takes place during close processing.

Defining, altering, and deleting a subscription

When a subscription is created administratively, rather that through an MQSUB API request, no subscribe security checks take place to see if the subscription can be created or altered, as the administrator has already been given this authority through the command, and command resource security associated with the command.

Security checks are performed to ensure that publications can be put to the destination queue associated with the subscription in the same way they are performed for a MQSUB request.

The user ID that is used for these security checks depends upon the command being issued and the contents of the SUBUSER parameter on the command if it is specified, as follows:

Table 15. User IDs used for security checks for commands

Command	SUBUSER specified and blank	SUBUSER specified and completed	SUBUSER not specified
DEFINE	Use the administrator ID	Use the User ID specified in SUBUSER	Use the administrator's ID
ALTER	Use the administrator ID	Use the User ID specified in SUBUSER	Use the User ID from the existing subscription

The only security check performed when deleting subscriptions using the DELETE SUB command is the command security check.

Subscription security

MQSO_ALTERNATE_USER_AUTHORITY

The AlternateUserId field contains a user identifier to use to validate this MQSUB call. The call can succeed only if this AlternateUserId is authorized to subscribe to the topic with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so.

MQSO_SET_IDENTITY_CONTEXT

The subscription is to use the accounting token and application identity data supplied in the PubAccountingToken and PubApplIdentityData fields.

If this option is specified, the same authorization check is carried out as if the destination queue was accessed using an MQOPEN call with MQOO_SET_IDENTITY_CONTEXT, except in the case where the MQSO_MANAGED option is also used in which case there is no authorization check on the destination queue.

If this option is not specified, the publications sent to this subscriber will have default context information associated with them as follows:

Table 16. Default publication context information

Field in MQMD	Value used
UserIdentifier	The user id associated with the subscription (see SUBUSER field on DISPLAY SBSTATUS) at the time the publication is made.
AccountingToken	Determined from the environment if possible; set to MQACT_NONE otherwise.
ApplIdentityData	Set to blanks.

This option is only valid with MQSO_CREATE and MQSO_ALTER. If used with MQSO_RESUME, the PubAccountingToken and PubApplIdentityData fields are ignored, so this option has no effect.

If a subscription is altered without using this option where previously the subscription had supplied identity context information, default context information will be generated for the altered subscription.

If a subscription allowing different user ids to use it with option MQSO_ANY_USERID, is resumed by a different user ID, default identity context will be generated for the new user ID now owning the subscription and any subsequent publications will be delivered containing the new identity context.

AlternateSecurityId

This is a security identifier that is passed with the AlternateUserId to the authorization service to allow appropriate authorization checks to be performed. AlternateSecurityId is used only if MQSO_ALTERNATE_USER_AUTHORITY is specified, and the AlternateUserId field is not entirely blank up to the first null character or the end of the field.

MQSO_ANY_USERID subscription option

When MQSO_ANY_USERID is specified, the identity of the subscriber is not restricted to a single userid. This allows any user to alter or resume the subscription when they have suitable authority. Only a single user may have the subscription at any one time. An attempt to resume use of a subscription currently in use by another application will cause the call to fail with MQRC_SUBSCRIPTION_IN_USE.

To add this option to an existing subscription the MQSUB call (using MQSO_ALTER) must come from the same userid as the original subscription.

If an MQSUB call refers to an existing subscription with MQSO_ANY_USERID set, and the userid differs from the original subscription, the call succeeds only if the

new userid has authority to subscribe to the topic. After successful completion, future publications to this subscriber are put to the subscriber's queue with the new userid set in the publication.

MQSO_FIXED_USERID

When MQSO_FIXED_USERID is specified, the subscription can only be altered or resumed by a single owning userid. This userid is the last userid to alter the subscription that set this option, thereby removing the MQSO_ANY_USERID option, or if no alters have taken place, it is the userid that created the subscription.

If an MQSUB verb refers to an existing subscription with MQSO_ANY_USERID set and alters the subscription (using MQSO_ALTER) to use option MQSO_FIXED_USERID, the userid of the subscription is now fixed at this new user id. The call succeeds only if the new userid has authority to subscribe to the topic.

If a userid other than the one recorded as owning a subscription trys to resume or alter an MQSO_FIXED_USERID subscription, the call will fail with MQRC_IDENTITY_MISMATCH. The owning user id of a subscription can be viewed using the DISPLAY SBSTATUS command.

If neither MQSO_ANY_USERID or MQSO_FIXED_USERID is specified, the default is MQSO_FIXED_USERID.

Chapter 7. Publish/subscribe deprecated function

Although WebSphere MQ Version 7.0 provides publish/subscribe function that is integrated into the queue manager, previous publish/subscribe engines and their applications, for example WebSphere MQ Version 6, WebSphere Message Broker Version 6 and WebSphere Event Broker Version 6 can coexist with WebSphere MQ Version 7.0. Coexistence is controlled with a new queue manager attribute, PSMODE. Using this attribute you can continue to use existing publish/subscribe applications without having to modify the WebSphere MQ Version 6.0 function that is deprecated. This collection of topics details WebSphere MQ deprecated Version 6.0 publish/subscribe function.

The PSMODE queue manager attribute controls whether WebSphere MQ Version 7.0 publish/subscribe and the queued publish/subscribe interface that enables previous publish/subscribe engines to coexist with WebSphere MQ Version 7.0 is running, and hence enables applications to publish/subscribe using the MQI and the queues being monitored by the queued publish/subscribe interface respectively. The three PSMODE settings are as follows:

- DISABLED - Neither WebSphere MQ Version 7.0 publish/subscribe nor the queued publish/subscribe interface are running, therefore it is not possible to publish or subscribe using the MQI and any publish/subscribe messages put to the queues monitored by the queued publish/subscribe interface will not be acted upon by WebSphere MQ.
- COMPAT - WebSphere MQ Version 7.0 publish/subscribe is running, therefore it is possible to WebSphere MQ Version 7.0 publish/subscribe using the MQI. The queued publish/subscribe interface is not running, therefore any messages put to the queues monitored by the queued publish/subscribe interface will not be acted upon by WebSphere MQ. This is the setting that should be used for compatibility with WebSphere Message Broker Version 6 (or earlier), since WebSphere Message Broker will read the same queues the queued publish/subscribe interface reads.
- ENABLED - WebSphere MQ Version 7.0 publish/subscribe and the queued publish/subscribe interface are both running, therefore it is possible to publish and subscribe using the MQI and the queues being monitored by the queued publish/subscribe interface respectively. This is the queue manager's initial default value.

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 99).
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 "Writing publisher applications" on page 99.

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.
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 "Writing subscriber applications" on page 102.

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 25 on page 83.

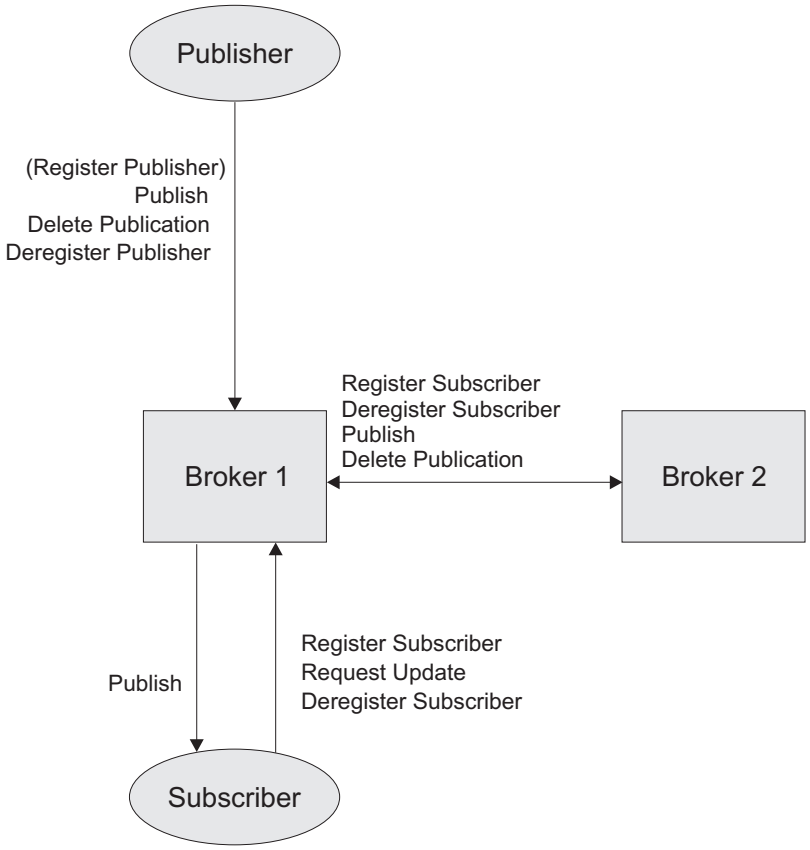


Figure 25. Communication between publishers, subscribers, and brokers

Streams

The use of streams is deprecated in WebSphere MQ Version 7.0.

WebSphere MQ Version 7.0 produces topic objects and topic strings by combining WebSphere MQ Version 6.0 StreamName and Topic parameters. For example, if the WebSphere MQ Version 6.0 StreamName is `UK.SPORTS.FEED` and the Topic is `Sport/Soccer/LatestScore`, the WebSphere MQ Version 7.0 publish/subscribe engine creates a topic called `/UK/SPORTS/FEED/Sport/Soccer/LatestScore`.

Where no StreamName is explicitly supplied, the queued publish/subscribe interface sets the topic object to the WebSphere MQ Version 6.0 default stream SYSTEM.BROKER.DEFAULT.STREAM.

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 146.

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 2 on page 5, 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.

  Wildcard characters are not used for stream names, and 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" on page 85). For example, if broker 1 in Figure 26 on page 86 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 allow different queue attributes (such as maximum message length) to be assigned for publications on different streams.

Version 6 wild card schema

If subscriptions are made using this schema, wild cards operate on all characters within the topic string. Using this schema, wild cards are interpreted in the same way they were in WebSphere MQ Version 6 and WebSphere Message Broker Version 6 when using MQRFH1 formatted message for publish/subscribe. Do not use this schema for new subscriber applications

To use this schema specify the MQSO_WILDCARD_CHAR option when subscribing.

If you use this wild card schema, the wild card characters recognized by WebSphere MQ publish/subscribe are:

***** Zero or more characters

? One character

In the example shown in Figure 2 on page 5, 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:

***** 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" on page 83).

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.

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 26 on page 86.

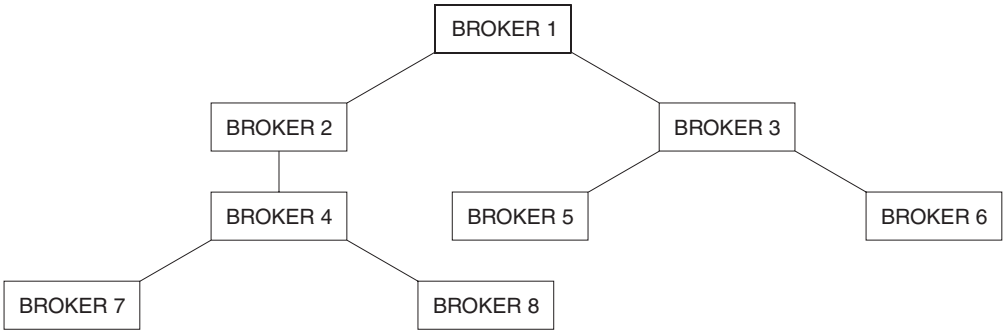


Figure 26. Simple broker hierarchy. Broker 1 is the root broker and brokers 2 and 3 are its children. Broker 4 is the child of broker 2 and the parent of brokers 7 and 8.

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.

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 .

Passing subscription information between brokers

Subscriptions flow to all nodes in the network that support the stream in question. This is shown in Figure 27 on page 87.

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 28 on page 87.

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 29 on page 88.

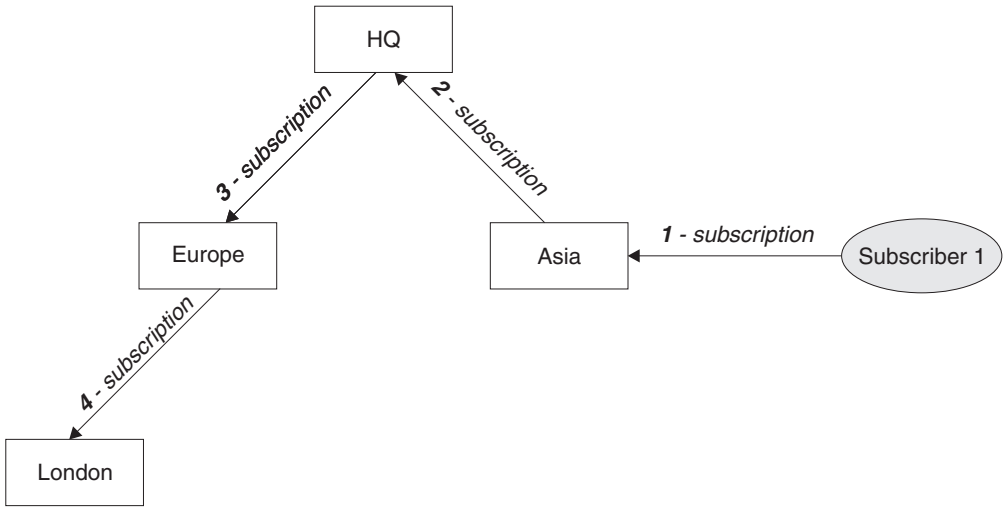


Figure 27. Propagation of subscriptions through a broker network. Subscriber 1 registers a subscription for a particular topic and stream on the Asia broker (1). The subscription for this topic is forwarded to all other brokers in the network that support the stream (2,3,4).

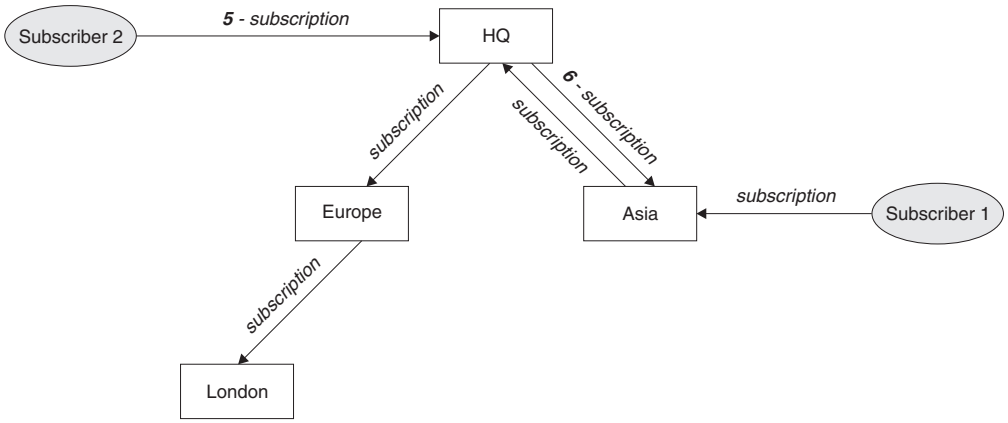


Figure 28. Multiple subscriptions. Subscriber 2 registers a subscription, with the same topic and stream as in Figure 27, on the HQ broker (5). The subscription for this topic is forwarded to the Asia broker, so that it is aware that subscriptions exist elsewhere on the network (6). The subscription does not have to be forwarded to the Europe broker, because a subscription for this topic has already been registered (step 3 in Figure 27).

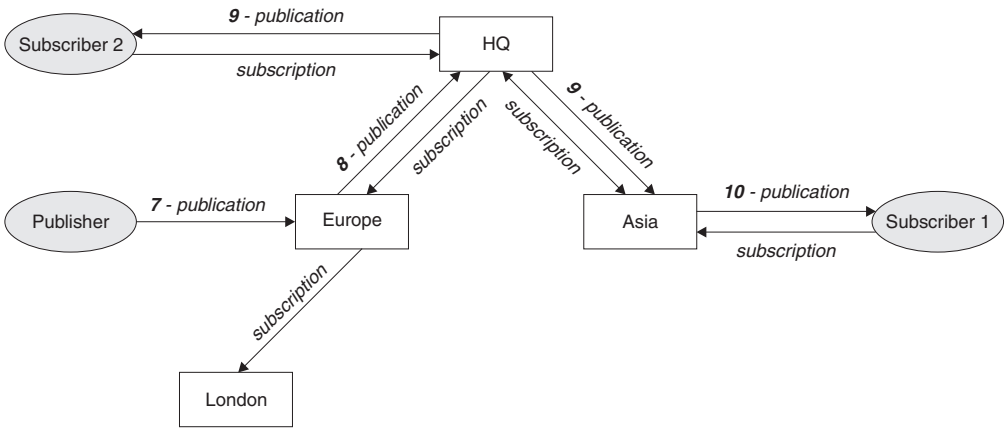


Figure 29. Propagation of publications through a broker network. A publisher sends a publication, on the same topic and stream as in Figure 28 on page 87, to the Europe broker (7). A subscription for this topic exists from HQ to Europe, so the publication is forwarded to the HQ broker (8). However, no 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 "Managing the broker" on page 146.

Writing publish/subscribe applications

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 "Format of command messages" on page 106. 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.

Message flows

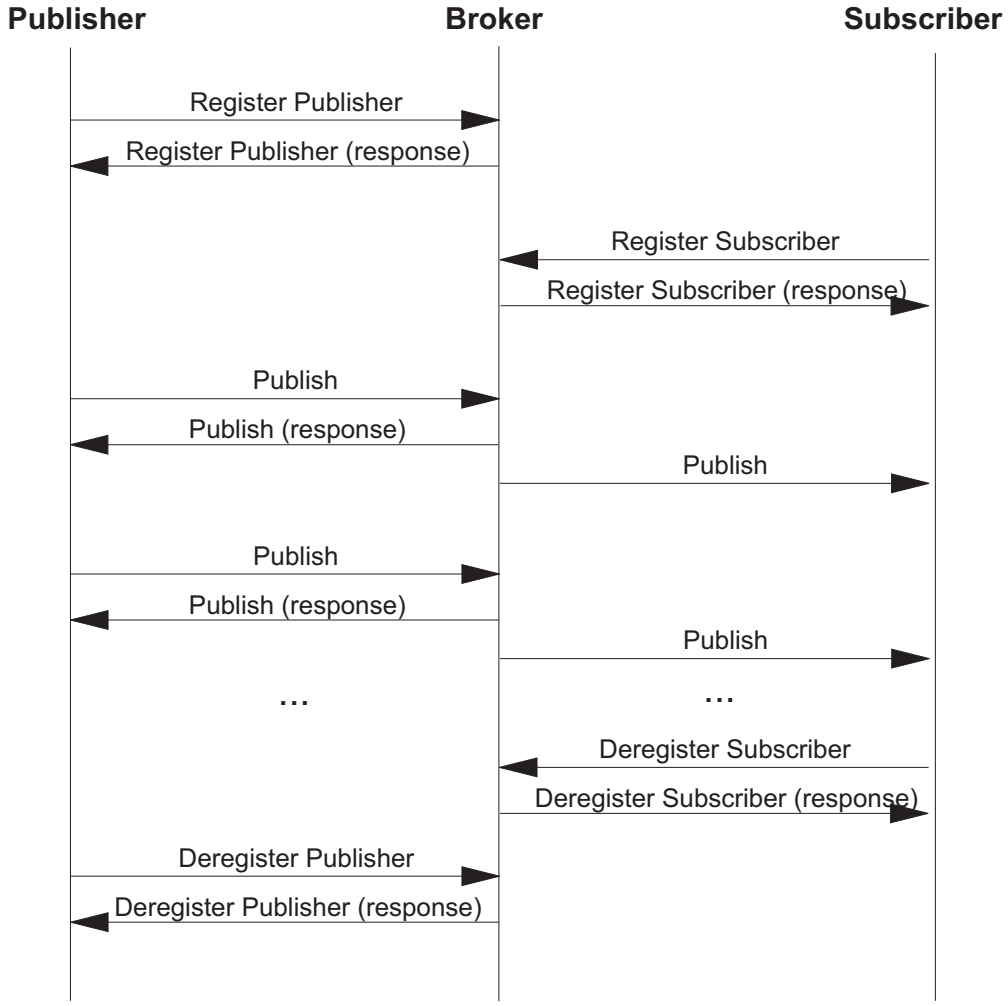


Figure 30. Basic flow of messages

Figure 30 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 31 on page 90.

Simplified message flow:

Figure 31 is a simplified version of Figure 30 on page 89 with the optional

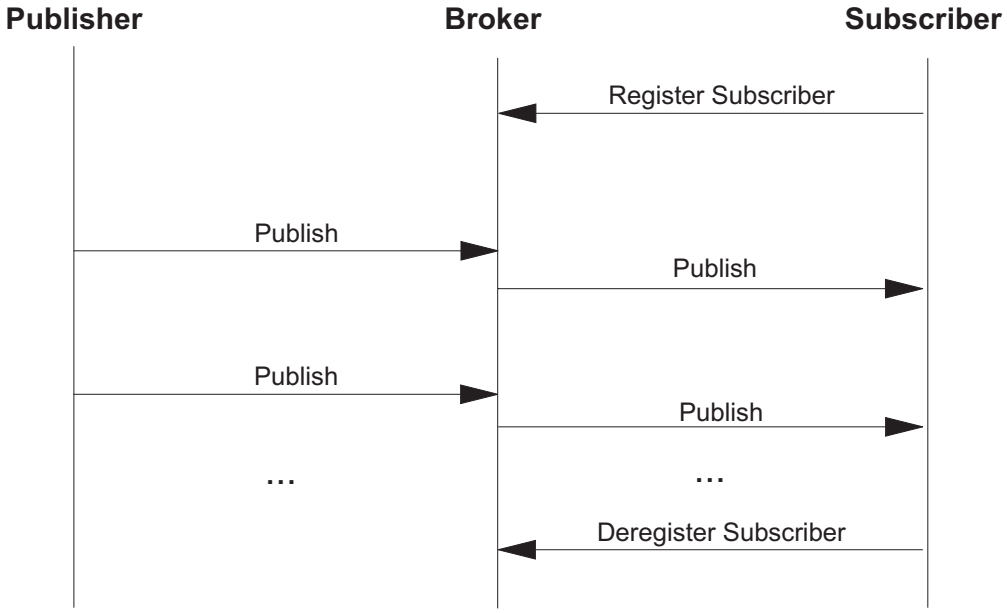


Figure 31. Simplified flow of messages

messages and responses omitted.

Figure 32 on page 91 shows how publish and subscribe messages flow between the publisher, the subscriber, and the broker queues. In Figure 33 on page 91 this is extended to a two-broker system.

The flow of messages when retained publications are used is shown in Figure 34 on page 92. In this case, the subscriber receives the current retained publication as soon as it registers a subscription. In Figure 35 on page 92, 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).

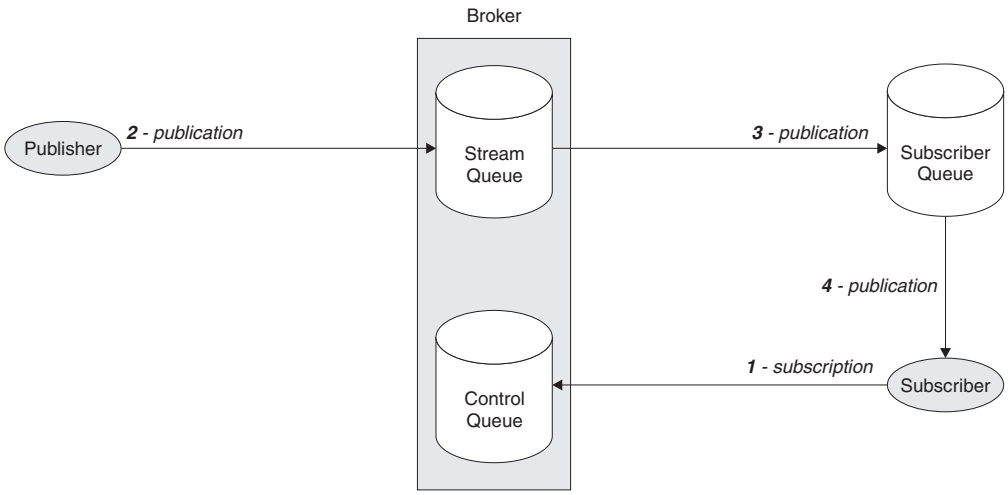


Figure 32. Flow of messages in a single-broker system. The subscriber registers a subscription by putting a message on the broker's control queue (1). Subsequently, a publisher puts a publication message, for the same topic, on the corresponding stream queue in the broker (2). The broker forwards the publication by putting the same message on the subscriber queue (3), from where the subscriber application can get it (4).

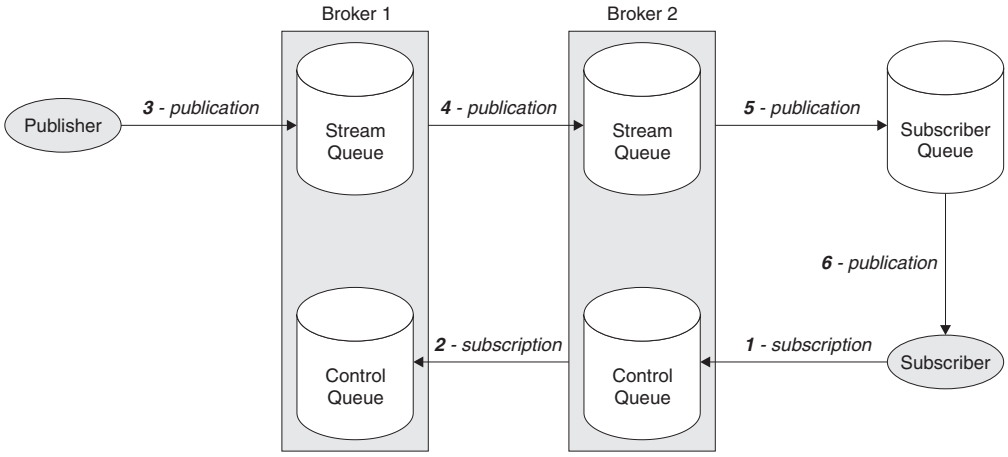


Figure 33. Flow of messages in a multi-broker system. The subscriber registers a subscription as in Figure 32(1). Broker 2 forwards the subscription by putting a message on the control queue of Broker 1 (2). Subsequently, a publisher puts a publication message, for the same topic, on the corresponding 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).

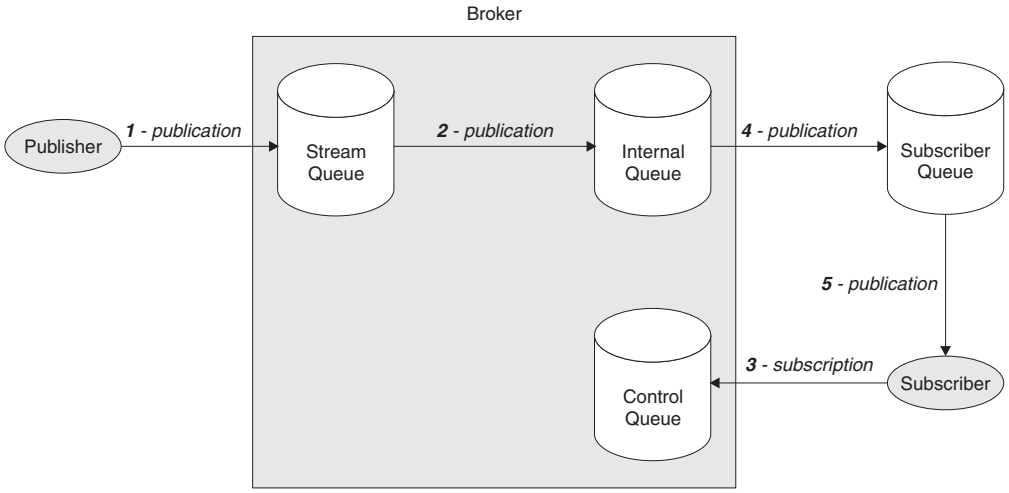


Figure 34. Flow of messages using retained publications. A publisher sends a retained publication by putting a message on the appropriate stream queue in the broker (1). The broker stores the publication on an internal queue (2). Subsequently, a subscriber registers a subscription, to the same topic 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).

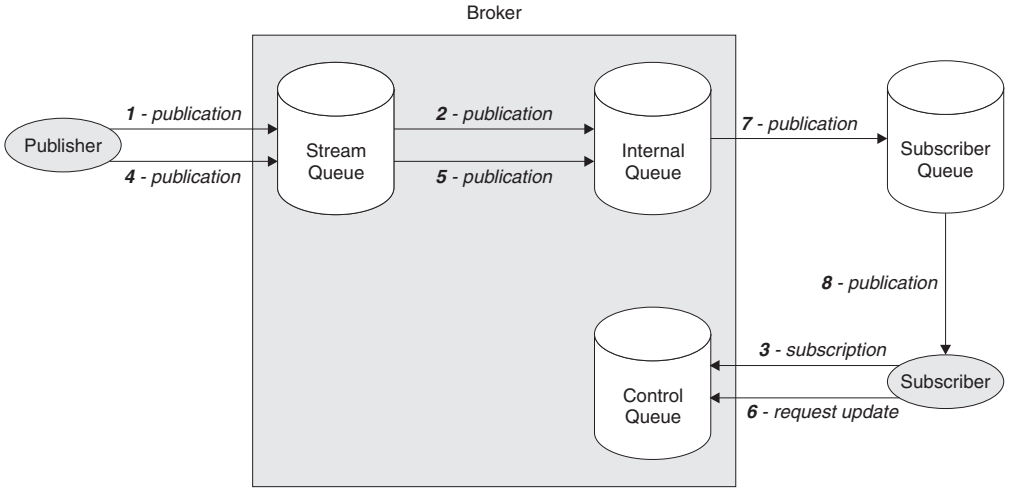


Figure 35. Flow of messages using publish on request only. A publisher sends a retained publication to a stream queue in the broker (1). The broker stores it on an internal queue (2). A subscriber registers a subscription, to the same topic and stream, by putting a message on the broker's control queue (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).

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 94.

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. 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 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 139.

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 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 149.

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 122).

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 "Publish/Subscribe command messages" on page 115.

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 Programmable Command Formats and Administration Interface book for details of how to use the functions mentioned above (including the name/value element helper functions).

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 100
- "Deleting information" on page 101
- "Deregistering with the broker" on page 102

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.

Registering with the broker

There is no **Register Publisher** command in WebSphere MQ Version 7.0. A publisher implicitly registers with the queued publish/subscribe interface by using any queue in SYSTEM.QPUBSUB.QUEUE.NAMELIST.

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 122.

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.

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 112. 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.

  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.

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 6 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.

To prevent further subscriptions to a retained publication, it can be removed from the topic and stream by clearing it. A retained publication can be cleared with the clear topicstring command

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 116.

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 117.

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).

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 105
- "Deregistering as a subscriber" on page 106

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 129.

Your application should send this message to a broker's control queue (see "Broker queues" on page 146). 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. 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. 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:

The use of streams is deprecated in WebSphere MQ Version 7.0.

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.. 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 described in "Request Update" on page 136. 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 5). 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.

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 119.

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*).

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 110
- "Publication data" on page 112

The name/value pairs that define the parameters needed for the command messages are detailed in "Publish/Subscribe command messages" on page 115.

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 topic. The AMI constructs and interprets the RF Header and its name/value pairs.

MQRFH – Rules and formatting header

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.

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:

- 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" on page 110.

Table 17. Initial values of fields in MQRFH

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

Table 17. Initial values of fields in MQRFH (continued)

Field name	Name of constant	Value of constant
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 break down the entire *NameValueString* into its constituent parts. 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 36 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 "Publish/Subscribe command messages" on page 115.

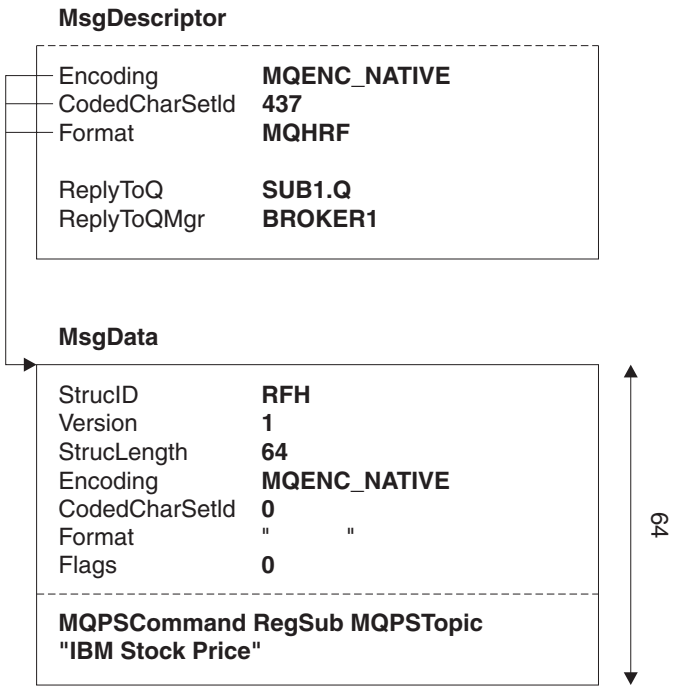


Figure 36. Message descriptor and RFH structure. The message descriptor indicates that the subscriber has nominated its subscriber queue to be the same as its reply queue. It also defines the encoding and CCSID of the RFH structure, which follows as the message data. The encoding and CCSID fields in the RFH structure are not set, because there is no data following the RFH structure (compare with Figure 37 on page 113). 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 122.

Figure 37 on page 113 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 38 on page 114 the publication data is included within the *NameValueString*, and in Figure 39 on page 114, 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).

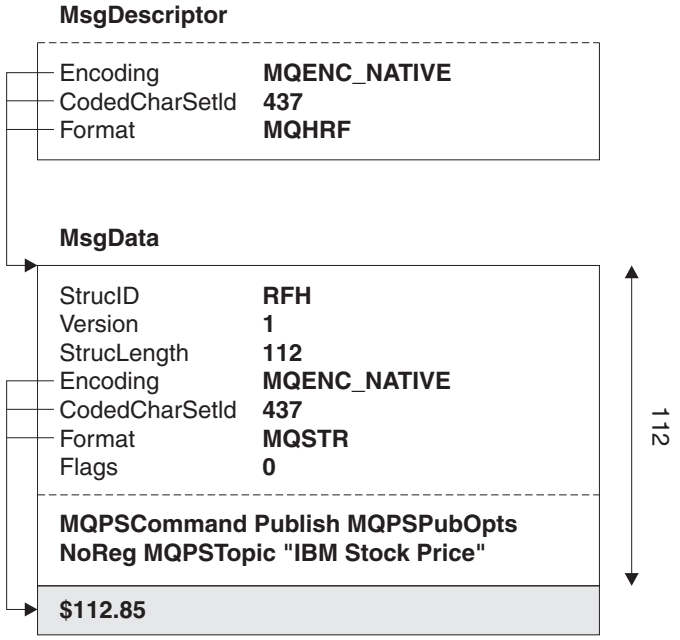


Figure 37. Publication data after the RFH structure. In this example, the publication data ($112.85) that is being published as string data in MQSTR format, is appended to the message after the NameValueString. Note that the RFH StrucLength includes the NameValueString, but not the publication data. 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.

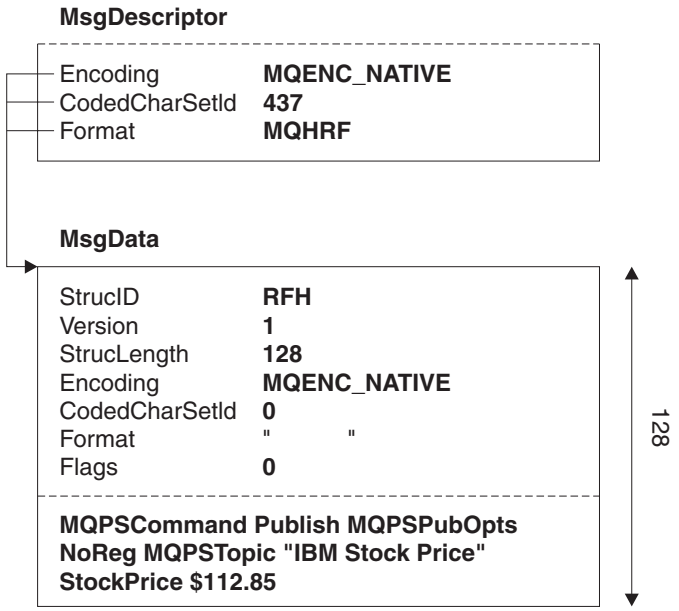
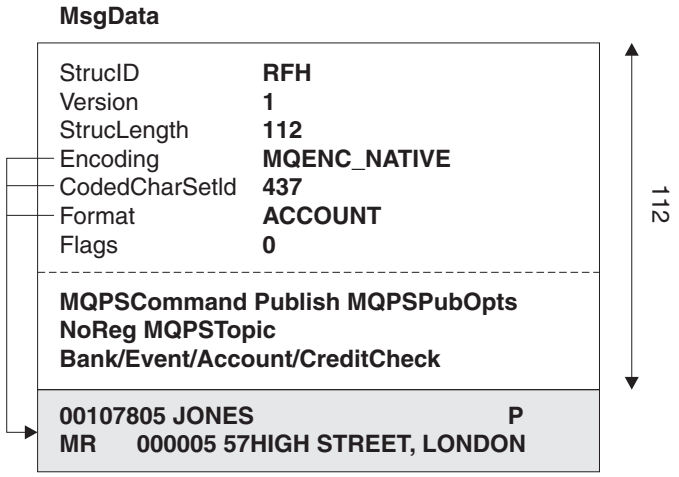


Figure 38. Publishing data within the NameValueString. Publication data can be included within the NameValueString, by means of one or more user-defined name/value pairs, as shown in this example. The encoding and CCSID fields in the RFH structure are not set, because there is no following data. The receiving application must parse the RFH structure to extract the publication data.




```
struct {  MQLOMG      AccountNo;
          MQCHAR      Customer[32];
          MQLONG      CreditRating;
          MQCHAR      Address[24] }
```

Figure 39. 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 40 shows how the CCSID for the RF header and the publication data can be inherited from the message descriptor.

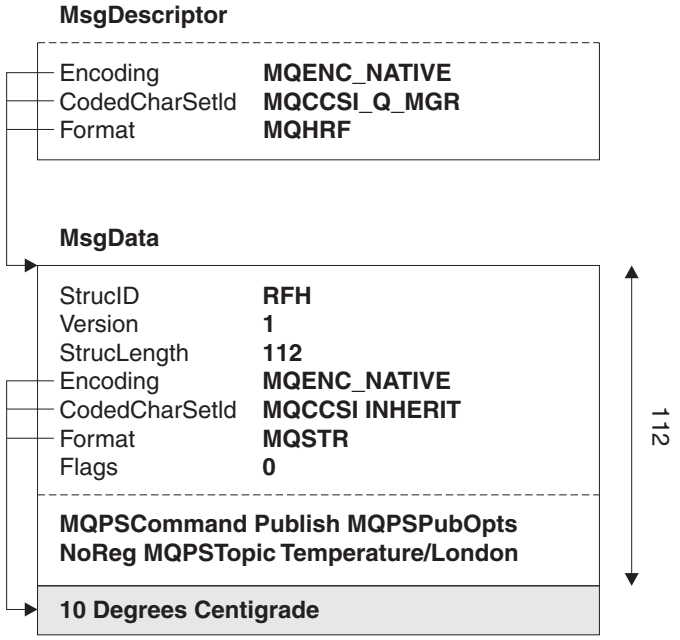


Figure 40. Inheriting the CCSID. The message descriptor uses the special value MQCCSI_Q_MGR to indicate that data within the RFH structure is in the same CCSID as the queue manager. The value of MQCCSI_INHERIT in the RFH structure indicates that the same CCSID is used for the publication data.

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 116
- "Deregister Publisher" on page 117
- "Deregister Subscriber" on page 119
- "Publish" on page 122
- "Register Publisher" on page 127
- "Register Subscriber" on page 129
- "Request Update" on page 136

"Format of command messages" on page 106 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. 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 122 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 in topic "Error codes applicable to all commands" on page 145.

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.

If a response message is required, this command builds a successful response message and puts it to the appropriate queue as specified by the MQMD ReplyToQ and ReplyToQMgr.

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 in topic "Error codes applicable to all commands" on page 145.

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.

Reason	Reason text	Explanation
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 can, optionally, be supplied.

"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: "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: 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.

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 can, optionally, be supplied. Otherwise, this parameter 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
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 in topic "Error codes applicable to all commands" on page 145.

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.

Reason	Reason text	Explanation
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 a publisher (or another broker) to a broker's stream queue or 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: `"MQPSCommand"` (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: "MQPSPubTime" (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).

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:

```
MQPSCommand     Publish
MQPSPubOpts     NoReg
MQPSStreamName  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.

```
MQPSCommand     Publish
MQPSPubOpts     RetainPub
MQPSPubOpts     NoReg
MQPSStreamName  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 in topic "Error codes applicable to all commands" on page 145.

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.

```
MQPSCommand   RegPub
MQPSRegOpts   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 in topic "Error codes applicable to all commands" on page 145.

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.

Reason	Reason text	Explanation
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>
MQPSQName       <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.

"InclStreamName"
(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 `"MQPSPubOpts IsRetainedPub"` to the *NameValueString* of the message (after the *StreamName* if that has been added in accordance with the `InclStreamName` 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.

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.

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 in topic "Error codes applicable to all commands" on page 145.

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 in topic "Error codes applicable to all commands" on page 145.

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.

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

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

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) 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.

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 "Setting up a broker" on page 146.

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.

Note:

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 MQMI_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.

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 in topic "Error codes applicable to all commands" on page 145, 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 in topic "Error codes applicable to all commands" on page 145, 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: "MQPSPubOpts" (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

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

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. Set the *MsgType* and *Report* options in the message descriptor of the command message so that the broker sends a response message.

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 `amqspsd` 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.
- 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.
  - 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 `amqspsd` sample to check that the stream is supported by all necessary brokers.

Managing the broker

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 "Controlling the broker" on page 150.

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`. 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 153).

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 83).

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 41 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:
   DiscardNonPersistentInputMsg=no
   DLQNonPersistentResponse=yes
   DiscardNonPersistentResponse=no
   GroupId=nobody
```

Figure 41. 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:

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 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.

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

i5/OS

- *GroupId*=*PUBLIC

Windows

- *GroupId*=Users

Note: For WebSphere MQ for Windows, the *GroupId* is set to 'Users' or the national language equivalent.

Controlling the broker

This chapter describes the following broker operations:

Starting a broker

mqconv7: Potential task?

Using triggering to start the broker:

mqconv7: Potential task?

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

Displaying the status of a broker

To display the publish/subscribe status of a queue manager, use DISPLAY QMGR or DISPLAY QMSTATUS. See DISPLAY QMGR or DISPLAY QMSTATUS for details.

Adding a stream

The use of streams is deprecated in WebSphere MQ Version 7.0.

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:

mqconv7: Potential task?

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.

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:

mqconv7: Potential task?

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

The use of streams is deprecated in WebSphere MQ Version 7.0.

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:

mqconv7: Potential task?

To delete a stream on a broker that is not part of a broker network:

1. Delete the queue.

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.

Deleting a stream on a broker that is part of a network:

mqconv7: Potential task?

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

mqconv7: Potential task?

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 i5/OS:

```
CRTMQMQ QNAME(PARENT) QTYPE(*RMT) RMTMQMNAME(PARENT) TMQMNAME(PARENT.XMITQ)
```

Deleting a broker from the network

mqconv7: Potential task?

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 154 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. Quiesce all applications that use the broker.
2. 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.
3. Use the **dltmqbrk** command (**DLTMQMBRK** on i5/OS) to delete the broker. This command is described in "dltmqbrk (Delete broker)" on page 155.

The broker performs the actions listed in "dltmqbrk (Delete broker)" on page 155 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.

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 are 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.

Sequence of commands for adding and deleting brokers

mqconv7: Potential super task?

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)
```

Use the following sequence for i5/OS:

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.

```
STOP CHANNEL (A.to.B)
START CHANNEL (A.to.B)
dltmqbrk -m B

STOP CHANNEL (B.to.A)
START CHANNEL (B.to.A)
dltmqbrk -m A
```

Use the following sequence for i5/OS:

```
ENDMQMCHL CHLNAME(A.to.B)
STRMQMCHL CHLNAME(A.to.B)
DLTMQMBRK MQMNAME(B)

ENDMQMCHL CHLNAME(B.to.A)
STRMQMCHL CHLNAME(B.to.A)
DLTMQMBRK MQMNAME(A)
```

Control commands

This topic describes the commands that you can use to manage your brokers. "Controlling the broker" on page 150 discusses the circumstances under which you would use these commands. The commands are:

- "dltmqbrk (Delete broker)" on page 155
- "endmqbrk (End broker function)" on page 156
- "strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0)" on page 37

You can now use the following commands on i5/OS and issue them using the command line. See chapter 2 ″Managing WebSphere MQ for i5/OS using CL commands″ in the WebSphere MQ for i5/OS System Administration Guide for further information about using the i5/OS command line.

- **DLTMQMBRK** - "dltmqbrk (Delete broker)" on page 155

- **ENDMQMBRK** - "endmqbrk (End broker function)" on page 156
- **STRMQMBRK** - "strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0)" on page 37

dltmqbrk (Delete broker)

Purpose

Use the **dltmqbrk** command to delete the broker. On i5/OS, the command name is **DLTMQMBRK**. The broker must be stopped when this command is issued, and the queue manager running. To delete more than one broker in the hierarchy, it is essential that you stop 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.

The broker performs the following actions:

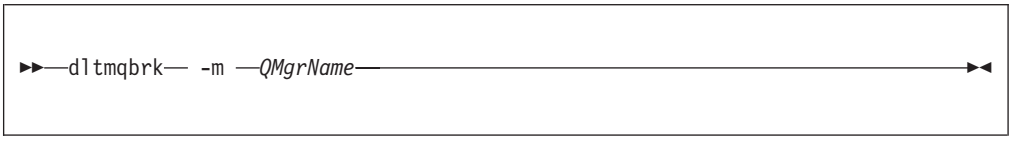
1. Put-inhibits its input queues (SYSTEM.BROKER.CONTROL.QUEUE and all stream queues).
2. Deregisters all its subscribers and publishers.
3. Sends Delete Publication commands to its parent for its metatopics.
4. Deregisters all its subscriptions with the parent.
5. Processes any messages on its input 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 (but does not remove the stream queues).
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 **dltmqbrk** again when the queue manager has been restarted.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows



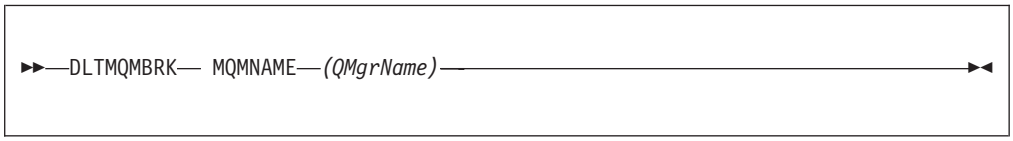
Required parameters

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

i5/OS



Required parameters

i5/OS

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

`dltmqbrk -m exampleQM`	Deletes the broker on exampleQM.

endmqbrk (End broker function)

The **endmqbrk** command is accepted for compatibility with earlier releases, but has no effect. To disable publish/subscribe on a queue manager, use the ALTER QMGR command. See ALTER QMGR for details.

strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0)

Purpose

Use the **strmqbrk** command to migrate WebSphere MQ Version 6.0 broker state to WebSphere MQ Version 7.0 publish/subscribe.

In WebSphere MQ Version 6.0, **strmqbrk** started a broker. The WebSphere MQ Version 7.0 publish/subscribe engine cannot be started in this manner. To enable publish/subscribe for a queue manager, use the ALTER QMGR command; for details, see ALTER QMGRin the *WebSphere MQ Script (MQSC) Command Reference*.

System programming

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 159

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 162.

Format of broker administration messages

The broker sends administration messages as **Publish** messages in PCF format. The following parameters are always present:

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 164.

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 *dltmqbrk* 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."

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.

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*'
- '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") 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.

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.

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 18. 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
StreamName	Y	Y	Y	Y	Y
Topic	Y	Y	Y	Y	Y
PublishTimestamp	Y	Y	Y	Y	Y
BrokerCount	Y	Y	Y	Y	Y
ApplCount	Y	Y	Y	Y	Y
AnonymousCount	Y	Y	Y	Y	Y
RegistrationTopic	Y [2]	N	N [3]	N	N [3]
RegistrationQMgrName	N	N	N	Y	Y
RegistrationQName	N	N	N	Y	Y
RegistrationCorrelId	N	N	N	Y	Y
RegistrationUserIdentifier	N	N	N	Y	Y
RegistrationRegistrationOptions	N	N	N	N	Y
RegistrationTime	N	N	N	N	Y
RegistrationSubName	N	N	N	N	Y
RegistrationSubUserData	N	N	N	N	Y
RegistrationSubIdentity	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. *Topic* parameter contains the registered topic.
4. Repeated.

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-k,u
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.

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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

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

AIX	DB2 Universal Database	i5/OS
IBM	IBMLink	MQSeries
OS/2	RACF	SupportPac
Tivoli	WebSphere	z/OS

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

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Other company, product, or service names may be trademarks or service marks of others.

Index

R

S

T

U

V

W

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.

SC34-6950-00

Spine information:

WebSphere MQ Publish/Subscribe User's Guide Version 7.0