MQSeries Integrator

Using the Control Center

Version 2.0.1

SC34-5602-01

MQSeries Integrator

Using the Control Center

Version 2.0.1

SC34-5602-01

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix F, “Notices” on page 401.

Second edition (August 2000)

This edition applies to IBM® MQSeries Integrator Version 2 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this book

This book describes how to use the MQSeries Integrator Version 2 Release 0 Modification Level 1 *Control Center*.

Who this book is for

This book is intended for anyone who needs to use the Control Center to perform these tasks:

- Defining messages and message sets
- Defining message flows
- Defining and managing the broker topology
- Setting up publish/subscribe access control

What you need to know to understand this book

You need to have read and understood the general introduction to all aspects of MQSeries Integrator in the *MQSeries Integrator Introduction and Planning* book.

Terms used in this book

Term	**Meaning**
click	Point to an object or action specified in the instructions, then press and release the left mouse button.
right click	Point to an object or action specified in the instructions, then press and release the right mouse button.
double-click	Point to an object or action specified in the instructions then press and release the left mouse button twice in rapid succession.
drag	Point to an object specified in the instructions, then press and hold the left mouse button and move the mouse pointer to the desired location. Release the left mouse button.

Where to find more information

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

MQSeries Integrator publications

The following books make up the MQSeries Integrator V2.0.1 library:

- *MQSeries Integrator Introduction and Planning*, GC34-5599
- *MQSeries Integrator for AIX® Installation Guide*, GC34-5841
- *MQSeries Integrator for Sun Solaris Installation Guide*, GC34-5864
- *MQSeries Integrator for Windows NT® Installation Guide*, GC34-5600
- *MQSeries Integrator Messages*, GC34-5601
- *MQSeries Integrator Using the Control Center*, GC34-5602 (this book)
- *MQSeries Integrator Programming Guide*, SC34-5603
- *MQSeries Integrator Administration Guide*, SC34-5792

MQSeries information available on the Internet

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

```
http://www.ibm.com/software/ts/mqseries
```

By following links from this web site you can:

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

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

  ```
  http://www.ibm.com/software/ts/mqseries/txppacs/
  ```

Summary of changes

This section describes changes to this edition of *MQSeries Integrator Using the Control Center*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

Changes for this edition (SC34-5602-01)

Changes for this edition include:

- Additional information to cover the following product changes:
  - New products MQSeries Integrator for AIX Version 2.0.1 and MQSeries Integrator for Sun Solaris Version 2.0.1.
  - New IBM primitive nodes (FlowOrder, Label, and RouteToLabel)
- Extended scenario showing additional function
- Extended examples of ESQL usage
- Extended information about supported messages
- Minor technical and editorial improvements throughout the book

Part 1. Introducing the Control Center

Chapter 1. Control Center concepts

This chapter introduces the Control Center by describing its role in an MQSeries Integrator broker domain, and defining those concepts that you need to understand as a Control Center user. For a comprehensive description of MQSeries Integrator concepts, see the *MQSeries Integrator Introduction and Planning* book.

The Control Center has two main functions in a broker domain. These are:

- The creation, manipulation, and deployment of configuration data for a broker domain
- The monitoring and management of the operational state of the same broker domain

These functions are described in the remainder of this chapter.

Working with configuration data

When a broker is created using the **mqsicreatebroker** command, and started for the first time using the **mqsistart** command, it has no configuration to run. A broker can perform useful functions only when it has been given a configuration to run by the Control Center user.

Configuration data is of three types:

Assignments data
Is the assignment of: execution groups to brokers; message flows to execution groups; and message sets to brokers.

Topology data
Is the relationship between brokers and collectives in a publish/subscribe network in the broker domain.

Topics data
Is topics and associated Access Control List (ACL) entries used in a publish/subscribe network in the broker domain.

Configuration and message repositories

Configuration data of all three types is created by Control Center users, and is managed by the *Configuration Manager* in two repositories called the *configuration repository* and the *message repository*.

- The message repository contains definitions of message sets.
- The configuration repository contains all other configuration data.

There is only ever one Configuration Manager in a broker domain, but there can be any number of instances of the Control Center.

Shared and deployed configurations

The Configuration Manager manages two versions of the configuration data. These are the *shared configuration* and the *deployed configuration*.

Shared configuration
Consists of configuration data as created by one or more Control Center users and made visible to other Control Center users in the broker domain.

Deployed configuration
Is the configuration data that is operational in (that is, that is having an effect in) the broker domain.

Configuration data in the shared configuration is sent to brokers by the Configuration Manager under the direction of Control Center users, by means of an operation called *deploy*. If deployment is successful, the Configuration Manager updates its deployed configuration accordingly.

The workspace

The concept of the *workspace* is key to the operation of the Control Center. It is the term given to the “snapshot” of that part of the shared configuration data that you, as a Control Center user, want to work with. The shared configuration can consist of many brokers, collectives, execution groups, message flows, message sets, and topics, many of which are of no interest to you. The workspace allows you to work with a subset of this overall set of configuration data.

All brokers, collectives, execution groups, and topics in the shared configuration always appear in your workspace. However, you can choose which message flows and message sets you want to appear, to make your view of the shared configuration more manageable. For example, if there are 500 message flows defined in the shared configuration, you can choose to see only the 10 that are owned by you. You do this using an operation called *add*. Similarly you can *remove* any configuration resource from your workspace.

In summary, the workspace is a collection of references to specific objects in the configuration.

Managing the contents of the workspace

If you want to create new configuration resources, you use an operation called *create*. This creates a new object within the Control Center and adds a reference to it to your workspace.

At this point, your new object does not exist in the shared configuration. To make your object visible in the shared configuration, you use an operation called *check in*. Once an object has been checked in it becomes visible to all other Control Center users.

If you want to modify an object in the shared configuration, you use an operation called *check out*. This locks the object in the shared configuration, preventing other Control Center users from modifying it, and makes a copy in your Control Center.

When you have made your modifications, you save them back to the shared configuration using *check in*, which also unlocks the object so that others can modify it.

If you want to destroy an object, you use an operation called *delete*. The object is deleted from wherever it exists, which could be the Control Center (if the object is newly created), or the shared configuration. If the object that you delete is in the shared configuration, and another user has a copy of this object in a local workspace, that user is not notified that you have deleted the object unless a refresh is requested.

Saving the workspace

You can save your workspace, so that it is preserved from one invocation of the Control Center to another, and to prevent work being lost. The workspace is saved as an XML (eXtensible Markup Language) file to the local file system. Any objects that you have created or have checked out are also saved to the local file system, to a directory known as your *local repository*. Therefore, the references in your workspace are to objects that exist either in the shared configuration or in your local configuration.

You can have as many workspaces as you like, but you have only one local configuration per shared configuration.[1] When an object is checked in, it is removed from the local configuration, if it existed there.

Monitoring the broker domain

When a deploy operation has taken place successfully, the target brokers automatically start to run the message flows, or to provide the publish/subscribe capability, associated with the deployment request. Using the Control Center, you can monitor the status of the brokers and the message flows they are running, and can perform a limited number of actions to control the operation of the brokers. For example, you can start and stop message flows.

Figure 1 on page 6 summarizes the concepts that have been introduced in this section. It shows:

- How the Control Center relates to other components of MQSeries Integrator
- The different containers of configuration data (the workspace, the local configuration, and the shared and deployed configurations)
- The main Control Center operations on the configuration data (save, check in, check out, deploy, and various operational actions)

[1] It is possible to switch between shared configurations on different Configuration Managers (for example, between a test and production system) using the **File —> Connection** dialog, as described in Chapter 2, “Getting started with the Control Center” on page 7. For each such shared configuration, there is one local configuration.

Control Center concepts

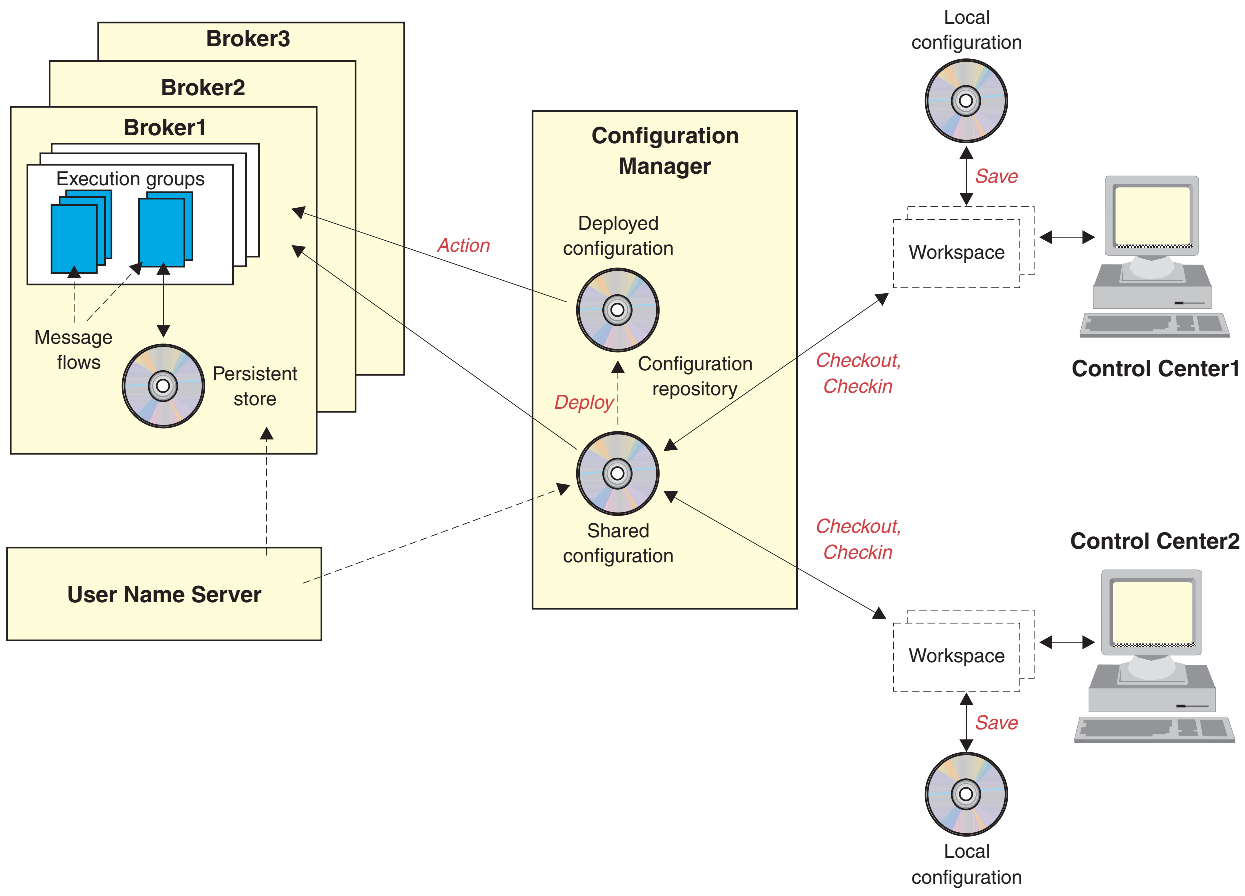


Figure 1. The role of the Control Center in the broker domain. In this figure, the configuration repository includes the message repository for simplicity. Terms in italics are Control Center operations.

Chapter 2, “Getting started with the Control Center” on page 7 provides instructions for performing many of the tasks introduced in this chapter.

Chapter 2. Getting started with the Control Center

This chapter describes how to get started with the Control Center. It introduces general considerations, such as tailoring your view of the Control Center, managing access to Control Center tasks, and working with the workspace.

Before you start

You are assumed to have read the *MQSeries Integrator Introduction and Planning* book. Before you can start to use the Control Center, you must have successfully completed the following tasks:

- MQSeries Integrator must have been installed.
- Users and groups must have been added to MQSeries Integrator security groups.
- All databases required by MQSeries Integrator must have been created, and users and groups authorized to use them.
- A Configuration Manager must have been created. A broker must also have been created if you will be deploying data.
- If you are using ACLs on publish/subscribe topics, a user name server must have been created and started.
- The MQSeries resources required to connect the queue managers hosting MQSeries Integrator components must have been defined.
- Listeners for the queue managers must have been started.
- The Configuration Manager must have been started.

For more information about these tasks, see the *MQSeries Integrator Installation Guide* for your operating system.

Release to release migration

If you are already using MQSeries Integrator Version 2.0, and are upgrading to the new release 2.0.1, you must be aware of the following:

- The Control Center at Version 2.0.1 level can only interact with a Configuration Manager that is also at Version 2.0.1 level. If the Configuration Manager is at Version 2.0 level, the connection fails with message BIP1360.
- The Control Center at Version 2.0 level can interact with a Configuration Manager that is at Version 2.0.1 level. However, you are recommended to upgrade all Control Center installations to Version 2.0.1 when you install a broker or the Configuration Manager at Version 2.0.1. This will ensure product and component consistency.
- You must delete and recreate the Configuration Manager after installing Version 2.0.1. This enables access to the new message processing nodes and message sets supplied with the product. For details of this task, see the *MQSeries Integrator Administration Guide*.
- Before you upgrade your Control Center to Version 2.0.1, you are recommended to check in message flows that you have checked out, or that

are new (and have never been checked in). After installation, all message flows will reflect the changes made to the IBM supplied nodes.

- If you have exported a workspace from a Version 2.0 Control Center, and import that workspace into a Control Center at Version 2.0.1 level, you can add the three new primitive nodes (FlowOrder, Label, and RouteToLabel) using the **Message Flow —> Add to workspace** option.

The following usability improvements have been made to the Control Center:

- **File —> Update from Shared** has been replaced by **View —> Refresh from Shared** and a Refresh icon on the toolbar to provide consistent refresh actions across the views.
- **File —> Local —> Revert to Shared** has been replaced by **File —> Revert to Shared**.
- **File —> Check In List** has been replaced by **File —> Check In —> List**.
- **File —> Local —> Save to Shared** has been replaced by **File —> Check In —> All (Save to Shared)**.
- A new action, **File —> Check In —> All in Current Workspace** saves to the shared configuration only those resources that are new or checked out in the current workspace. It includes a check that informs you if dependent resources from other workspaces have been checked in as a result of this action.
- The Deploy Topology action is only accessible from the Topology document context menu.
- The Deploy All Types action is now only available from the File menu (not from the Topology document context menu).
- The Log view now appears by default. The **File —> Log** action has been removed: the Log view selection is now on the User Preferences dialog.
- The detailed action of Clear Log and Refresh Log has changed to ensure that messages are preserved across restarts of the Control Center:
  - Refresh Log now only displays messages in the Log view.
  - Clear Log now clears the Log view and deletes messages from the Configuration Manager's deployed configuration.

Starting the Control Center

To start the Control Center, you can:

- Double click the **Control Center** icon in the MQSeries Integrator product folder on your desktop.

or

- From the **Start** menu, click
  Programs —> IBM MQSeries Integrator Version 2.0 —> Control Center.

When you start the Control Center, the **Configuration Manager Connection** dialog is displayed. To complete the dialog:

- In the Host Name field, type the network host name of the system on which the Configuration Manager has been created.
- In the Port field, type the port number on which the queue manager hosting the Configuration Manager is listening. The default port number is 1414. (You can find out the port number to enter here from **MQSeries Services**. Right click the listener associated with the queue manager, select **Properties** and click the **Parameters** tab to display the port number.) No other queue manager must be listening on this port.
- In the Queue Manager Name field, type the name of the queue manager hosting the configuration manager.
- Click **OK**.

After you have started the Control Center and connected to the Configuration Manager, you can update these connection details. To do this, click **Connection** from the Control Center **File** menu. The **Configuration Manager Connection** dialog is displayed. You can alter the values displayed in this dialog, and click **OK** to apply the new values. If you do this from an unsaved workspace, you are given the opportunity to save the workspace before changing the connection information.

If you change your mind about the values you have typed in the dialog and have not clicked **OK** to apply them, click **Reset** to restore the values with which you connected to the Configuration Manager.

Note: You must not alternate between alias names for the Host Name value. If you connect using a different alias for the same host, you get a different local configuration that is unique to the alias name. You will no longer be able to access resources you created in your original local configuration, nor will you be able to check in any resources checked out to the original local configuration.

When you start the Control Center subsequently, the fields in the **Configuration Manager Connection** dialog display the values you supplied when you last connected to the Configuration Manager.

The Control Center interface presented to you initially looks something like this:

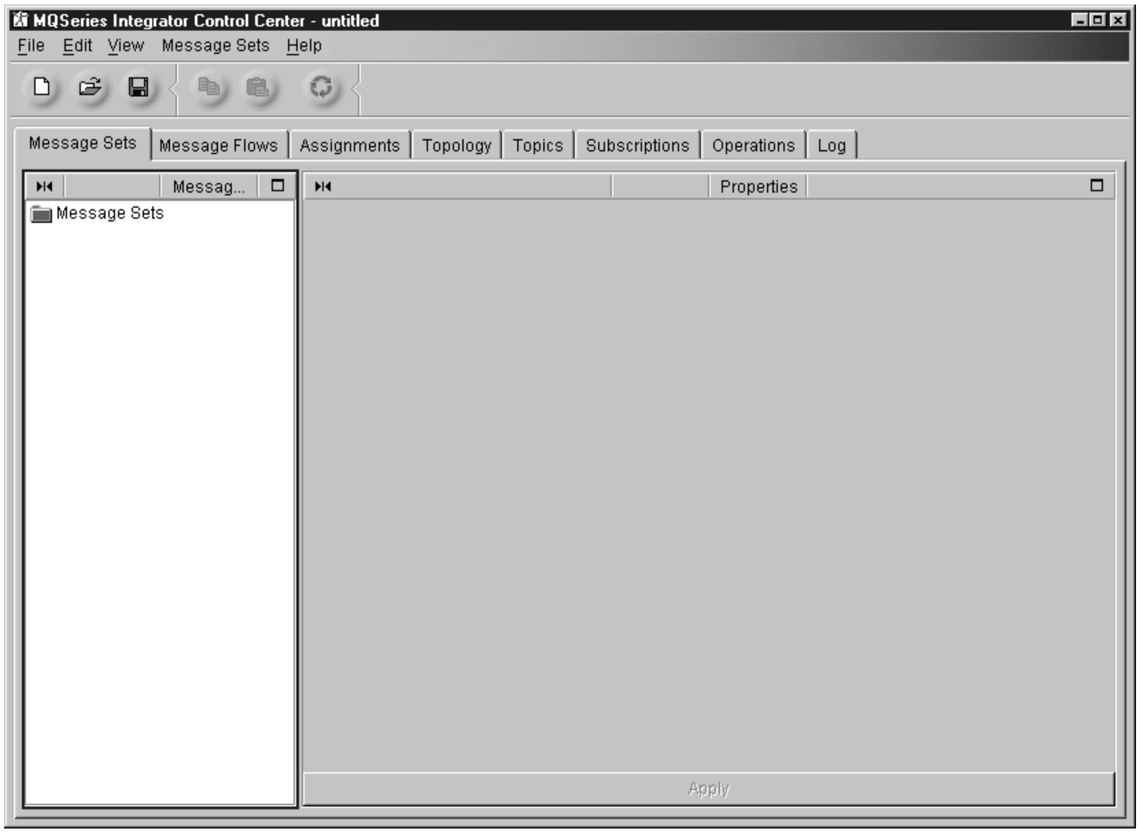


Figure 2. The Control Center

You see all the tabs shown here if your user role is **All roles**. User roles are described in "Managing permissions to Control Center tasks" on page 12.

Exiting the Control Center

To exit the Control Center, click **Exit** from the **File** menu in the taskbar. You are prompted to save any unsaved work before exiting.

The Control Center Tour

The Control Center provides an online tour, accessible from the Help menu. The tour gives you an overview of the MQSeries Integrator product: the components, the tasks you have to do to install, configure and run it, and the main interface you'll use (the Control Center).

The Tour introduces the product in three ways:

- It provides introductory information that you can read, with links to MQSeries information on the Web.
- It provides animated sequences of actions in the Control Center. For example, you can see how a message flow and message set are created using the Control Center.

- It creates sample objects, in your own Control Center workspace. These sample objects cannot be deployed but you can examine them to see how the business scenario can be implemented.

The Tour is based on a simple example scenario, in which MQSeries Integrator is used to integrate the processes of an international company.

To access any section of the Tour, click on the title of the section in the left-hand panel. When you run the animated sequences, there are four buttons at the bottom of the panel. These buttons let you play, pause, stop, and view a particular segment of the movie you are playing.

If you use the Tour to create definitions in your workspace, note the following:

- You have to create definitions in the Tour in the order in which they are presented. You cannot use the Tour to create a message set, for example, then close it, reopen and go directly to assigning that message set to a broker in the Control Center.
- You can save the sample objects to the shared repository if you are the only user of the Tour on a particular Configuration Manager. If another user has already performed this task, you will not be able to update the shared repository with the sample objects created in your workspace.

To use the Tour:

- Your logon user ID must be a member of one of the MQSeries Integrator groups (mqbrkrs, mqbrasgn, mqbrdevt, mqbrops, or mqbrtpic).
- You must have one of the following Control Center roles:
  - Message flow and message set developer
  - Message flow and message set assigner
  - Operational domain controller
  - All roles

If you want to use the Tour to create definitions in the workspace, you must have the appropriate authority and have set the appropriate Control Center role. See “Managing permissions to Control Center tasks” on page 12 for further details about user roles and authorizations.

To run the animated sequences supplied as part of the Tour you need to have installed two plug-ins:

- Java Plugin 1.1.2_001
- Lotus® Media 2.2

If you did not install these plug-ins when you installed the Control Center, you can install them now from the TourXtra directory on the product CD.

Managing permissions to Control Center tasks

The Control Center supports many different tasks that you perform when working with configuration data or monitoring operational brokers. These tasks are grouped by *user role*, as follows:

Message flow and message set developer
Can create message flows and message sets.

Message flow and message set assigner
Can create execution groups within brokers, assign message flows to execution groups, and assign message sets to brokers. Can also deploy this data.

Operational domain controller
Can create brokers and collectives, and define the relationships between them (the topology). Can also deploy all types of data, and monitor the operational broker domain.

Topic security administrator
Can create topics and associated ACLs. Can also deploy this data.

How to select the role you want to adopt is described in “Setting user roles” on page 13.

According to the role you select, the Control Center displays only those views or tabs that are relevant to that role, as follows:

- The **Message flow and message set developer** sees the *Message Sets* view and the *Message Flows* view.
- The **Message flow and message set assigner** sees the *Assignments* view only.
- The **Topic security administrator** sees the *Topics* view and the *Topology* view.
- The **Operational domain controller** sees the *Topology* view, the *Assignments* view, the *Topics* view, the *Operations* view, and the *Subscriptions* view.

If you want to perform all tasks, you should select All roles, which allows you to see all available views.

The role you select for yourself only configures what you *see* on the Control Center. It does not control the type of objects you can view or modify. For security purposes, this aspect is controlled by the MQSeries Integrator security groups of which you are a member.

The MQSeries Integrator security groups, and the Control Center tasks that membership of those groups allows, are:

mqbrdevt
Members of this group can design message sets and message flows.

mqbrasgn
Members of this group can manage execution groups within brokers; view message sets and message flows; assign message flows to execution groups; and assign message sets to brokers.

mqbrops
Members of this group can create and delete brokers; deploy, start, and stop message flows; start and stop trace activity on message flows; manage and deploy the broker domain topology, including collectives; view the whole deployed system, including message sets, message flows, and subscriptions; deploy topics; and view logs that report on the deployment activity.

mqbrtpic
Members of this group can manage topics, and the access control lists for the topic tree; deploy topics; view the logs that report on that deployment activity.

The Configuration Manager performs a security check based on the above whenever a Control Center user attempts to view or modify an object in the configuration and message repositories.

Adding users and groups to the MQSeries Integrator groups

You must use the Windows NT User Manager to add users and groups to the MQSeries Integrator security groups, as follows:

1. Invoke the Windows NT User Manager by selecting **Start —> Programs —> Administrative Tools (Common) —> User Manager**.
2. Double click the MQSeries Integrator group you want to update.
3. Select **Add**. From the list of available user IDs, select the user ID to be added to the group.
4. Click **Add**. Click **OK** to the close the **Add Users and Groups** dialog.
5. Click **OK** to close the **Local Group Properties** dialog.
6. Close the User Manager.

The authorization is effective after a delay of approximately five minutes, as the Configuration Manager caches this information.

For more detailed information about the MQSeries Integrator groups, and about security in general, see the *MQSeries Integrator Introduction and Planning* book.

Setting user roles

At any time during a Control Center session, you can change your user role. To set your Control Center user role:

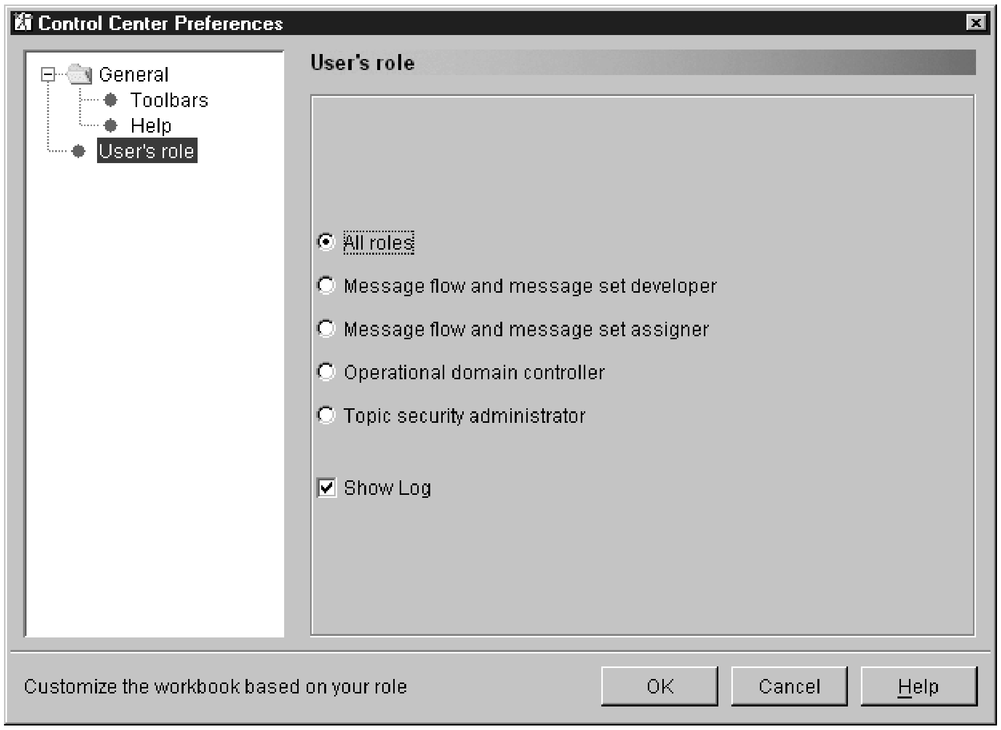
1. From the **File** menu in the Control Center taskbar, click **Preferences**.

   The **Control Center Preferences** dialog is displayed.
2. In the left-hand pane of the dialog, click **User's role**.

   In the User's role pane, as shown in Figure 3 on page 14, select the role you want to adopt, and click **OK**.

You can select only one role at a time.

User preferences that govern the general appearance of the Control Center can also be set by selecting **File —> Preferences...**. For more information, see the Control Center online help.



User's role

(•) All roles
( ) Message flow and message set developer
( ) Message flow and message set assigner
( ) Operational domain controller
( ) Topic security administrator

[✓] Show Log

Customize the workbook based on your role

Figure 3. Setting the user role

You can also control the presence of the Log view from the Control Center Preferences dialog. The default setting is that the Log view is always visible, regardless of the user role currently in force. You can change this setting if you choose, to suppress the Log view, by deselecting the check box.

The setting of this option is remembered from session to session.

Performing workspace tasks

This section describes the various ways in which you can manipulate the contents of the entire workspace.

Creating a new workspace

To create a new workspace, click **File —> New Workspace**.

A new workspace is created. This workspace is untitled, and displays the default contents. You specify a title for the workspace when you save it.

You are prompted to save any changes made in a previous workspace: if you do not do so, they are discarded.

Opening an existing workspace

When you open an existing workspace, your workspace is populated with the resources from the chosen file, and these replace the contents of any previous workspace.

You are prompted to save any changes made in a previous workspace: if you do not do so, they are discarded.

To open an existing workspace:

1. Click **File —> Open Workspace**.

   The **Open** dialog is displayed.

2. Select the file (which must be a file of any name and extension that contains valid XML: typically the file will have an extension of `.xml`). from the list presented, or specify the name of the file in the File name field. Click **Open** to open the selected workspace.

Alternatively, if you have recently used the workspace, click the **File** menu in the Control Center taskbar, which displays the names of the most recently used workspaces. There can be up to four names in this list. If the workspace you want is listed in the **File** menu, click its name to open it.

Saving the workspace

When you save a workspace, both the workspace and any resources created or modified in that workspace are saved to the local configuration.

To save a workspace, click **File —> Save Workspace**. The workspace contents are saved to an XML file. If the workspace is untitled, you are prompted for a name. The name you give it is displayed on the Control Center title bar whenever that workspace is your current workspace.

To save a named workspace under a different name, click **File —> Save Workspace As**. The workspace contents are saved to an XML file of the specified name. This effectively takes a copy of the workspace contents.

Updating the workspace

Before you can make significant updates to any resource in your workspace, you must check it out of the repository in which it is maintained. Message sets and all their components are maintained in the message repository, all other resources are maintained in the configuration repository. Resources that are currently checked out have the **Key** icon against their entries wherever they are displayed. They are locked to your user ID, thus preventing other users making changes to them.

You can also work with new resources, that you create within your workspace. New resources that have never been checked in have the **New** icon against their entries wherever they are displayed.

The two icons are shown in Figure 4.

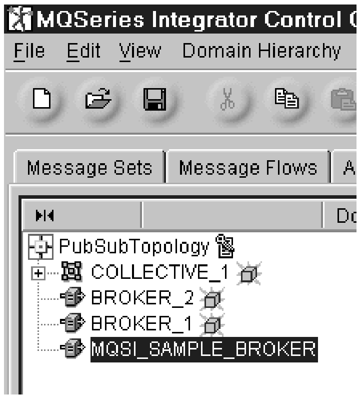


Figure 4. The Key icon and the New icon. The Topology is checked out, and brokers BROKER_1 and BROKER_2 and the collective have never been checked in. Broker MQSI_SAMPLE_BROKER is already checked in (and has no icon against it).

You can also take the following actions for the workspace itself:

- Revert to the shared version of the workspace
- Save the workspace to the shared configuration

Reverting your workspace to the shared repository

When you revert your workspace to the shared repository, any changes you made to it since opening the local version are lost, and any resources you had checked out are unlocked. The latest versions of the workspace objects in the shared repository are opened.

To revert to the shared version of the workspace, you can use either of the following two methods:

- **File —> Revert to Shared**

  This action unlocks all resources you have checked out, in all your workspaces. Changes you have made are lost. All resources revert back to the state they were in when you checked them out.

  For example, if you have checked out a message flow and have changed a property value for a node within it, and then select **Revert to Shared**, the message flow is unlocked and its content reloaded from the Configuration Manager. The change you have made to the node is lost.

  This action does not restore deleted resources. Any resource that you have deleted remains deleted.

- **View —> Refresh from Shared**

  This action does not affect the state of the resources you have checked out. The content of all other resources is updated to be consistent with the information held by the Configuration Manager. This allows you to update to and work with the latest changes that other users have made, without losing any changes you are making.

Saving the workspace to the shared repository

When you save the workspace to the shared repository, configuration changes you have made are saved, and objects that were checked out are checked in.

You can check in any individual resource at any time by selecting it, right-clicking to bring up the resource menu, and selecting **Check In**.

When you click **Check In**, other resources that are dependent on this resource are also checked in.

You can find out which configuration resources are checked out to you in your current workspace by selecting **File —> Check In —> List**. The **Check In List** dialog, shown in Figure 5, is displayed.

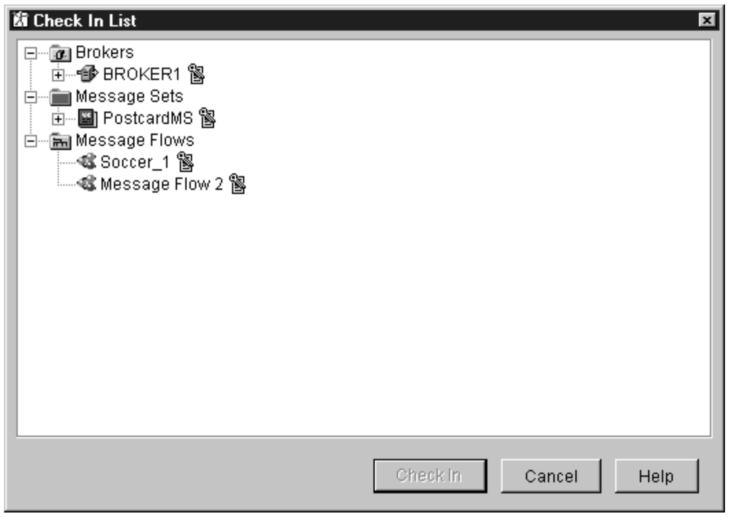


Figure 5. The Check In list dialog

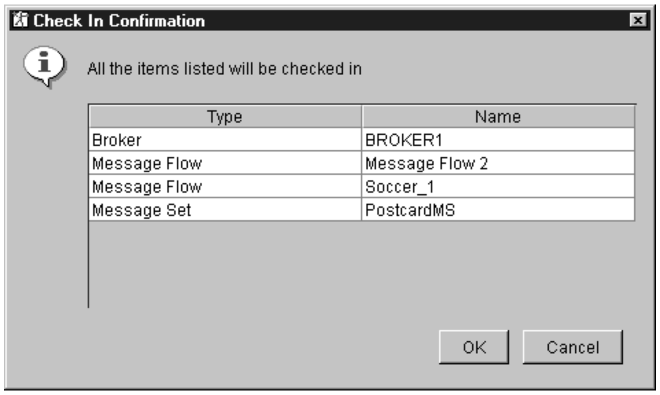
When you select one or more items from this list, other resources that are dependent on this resource are displayed in a dialog, and you can choose to continue with the check in action, or cancel it.

You can check in all the objects that are new or checked out in your current workspace. Your current workspace is identified in the title bar of the Control Center. Select **File —> Check In —> All in Current Workspace** to select this option.

The Control Center locates the new and locked objects that you have in this current workspace, and it checks for other new or locked objects that are referenced directly (in this same workspace) or indirectly (in another workspace) by these objects.

Therefore this action can check in more than you have in your current workspace. This extra action is required to ensure that the data in the message and configuration repositories retains its integrity and consistency.

You are presented with a dialog box, shown in Figure 6 on page 18, that lists all the resources that will be checked in by this action, and you can continue or cancel the action.


Check In Confirmation

All the items listed will be checked in

Type	Name
Broker	BROKER1
Message Flow	Message Flow 2
Message Flow	Soccer_1
Message Set	PostcardMS

Figure 6. The Check In All in Current Workspace dialog

You can also check in all the objects that are new or checked out in all your local workspaces. Select **File —> Check In —> All (Save to Shared)**. Information from all your workspaces (your current open workspace and all others on your local system) are reviewed by the Control Center to identify new or locked objects. All items are checked in together.

You can check in a resource from the **Check In List** dialog by highlighting the resource and clicking **Check In**.

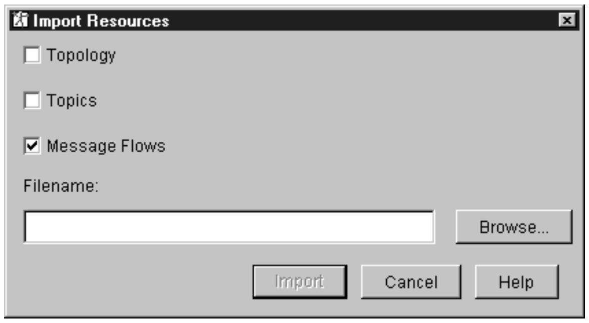
You can find further details on checking resources in and out in the online help for the Control Center.

Importing resources

You can import resources from an XML export file into the local repository. To import resources:

1. Click **File —> Import**.

   The **Import** dialog is displayed.


Import Resources

Topology [ ]
Topics [ ]
Message Flows [✓]

Filename:

Figure 7. The Import dialog

2. Select a file from the list in the dialog, or specify a name in the File name field. You must specify the fully-qualified path for the file.

   The specified file is interrogated, and you are given a choice of types of resources that you want to import from the file you have selected. The valid types are:

   - Topology
   - Topics
   - Message flows

Importing and exporting message sets

You cannot import message sets using this method. You must use the import and export command (**mqsimrmimpexp**) to import message sets. See the *MQSeries Integrator Administration Guide* for details.

When you have selected the type or types of resources you want to be imported, the relevant contents of the file are imported. The file contents replace the current workspace contents. Your current workspace is being replaced with the workspace in the export file to help you locate the imported resources.

You cannot import individual resources: the import action imports all the message flows, all the topics, all the topology data, depending on the type of resource you have selected.

When the import action has completed, a report is displayed indicating how many resources have been imported.

For the import action to succeed:

- You must check out objects that you are importing if they already exist in the shared configuration. Each existing checked out object that is successfully imported is included in the report displayed, and remains in your workspace and is marked with the key icon.
- If you do not check out an existing object, and it is in the import file, it is not imported but it is added to your workspace. You will therefore see the object appear in the appropriate view, but it is not checked out and will therefore not be marked by the key icon.
- Resources that do not already exist are imported into your workspace and are marked with the new icon. You must check these in if you want to save them in the shared configuration. The imported workspace might refer to resources that cannot be found in the Configuration Manager you are currently connected to. You can safely remove these resources if you do not want them created in this Configuration Manager.
- You must check out the Topology root if you are importing Topology information.
- You must check out the Topic root, and any other topic that will be changed by the import, if you are importing Topic information.

If the current workspace contents were unsaved, you are prompted to save them before the new resources are imported.

Please see the Control Center online help for more information about the effects of the import operation.

Exporting the workspace

When you export a workspace, all resources currently displayed in your current workspace (including topology, message flows, and topics but excluding message sets), and all resources that they depend on, are exported to an XML file, along with the workspace itself. The export file can then be imported by other Control Center users.

Export does not permit selection of objects, as import does. You can only export a complete workspace.

Importing and exporting message sets

You cannot export message sets using this method. You must use the import and export command (**mqsimrmimpexp**) to import message sets. See the *MQSeries Integrator Administration Guide* for details.

To export a workspace:

1. Click **File —> Export**.

   The **Export** dialog is displayed.

2. Select a file from the list in the dialog (if you want to export the workspace to an existing file) or specify a name in the File name field.

   The workspace is exported to the specified file. Its contents are in addition to any resources already in the file.

Note that information being exported might contain sensitive information pertaining to the users and groups who are defined on the User Name Server. If you are a member of MQSeries Integrator group **mqbrtpic** or **mqbrops**, the topic hierarchy and associated ACL are also exported. If you want to avoid this, you should sign on as a user who is not a member of either group before you run the export.

Naming Control Center resources

There are some rules you must follow when providing names for the resources you create using the Control Center:

- You can use the characters:
  - Uppercase A — Z
  - Lowercase a — z
  - Numerics 0 — 9
  - The special characters $ % ‘ ’ – _ @ ˜ ! ( ) { } [ ] ^ # & + , ; =
- You can also use the space character, and any Unicode character with an ASCII value greater than 127 (X'7F').

More specific guidelines exist on some operating systems. For example, names on UNIX systems (AIX and Sun Solaris) are case sensitive, but on Windows NT they are not. Therefore you must ensure that names specified in the Control Center, for example broker names, exactly match the names used on the system on which they are created. For more information about these guidelines, see the *MQSeries Integrator Administration Guide*.

Problem determination

If an error occurs while you are performing a Control Center operation, the Control Center displays a dialog box containing an MQSeries Integrator V2.0.1 message. The message can originate from either the Control Center itself or from the Configuration Manager. The message should explain any corrective action you can take.

Any errors that occur:

- During the second phase of a deploy operation

or

- From starting or stopping message flows

or

- From starting or stopping user tracing

or

- From deleting subscriptions

are displayed as MQSeries Integrator messages in the **Log** view. These messages originate from the broker.

You might also find it helpful to refer to additional information provided in the MQSeries Integrator V2.0 SupportPac MHI1. This SupportPac provides latest problem determination information in a useful question-and-answer format. You can find this SupportPac at:

```
http://www.ibm.com/software/ts/mqseries/txppacs/
```

Controlling service traces

The Control Center can be traced by invoking it with a special command, **mqsilcc**, which is described in the *MQSeries Integrator Administration Guide*. You are recommended to use service traces only when you receive an error message that instructs you to start service trace, or when directed to do so by your IBM Support Center.

Part 2. Using the Control Center

Chapter 3. Defining messages

MQSeries Integrator Version 2 provides a message brokering function that can transform messages from one format to another. The brokers that manage these transformations need to interpret the structure and content of the messages they receive to perform the full range of transformation functions available with MQSeries Integrator.

This chapter introduces the messages supported by MQSeries Integrator, and how those messages are handled. It contains the following sections:

Basic message concepts

A message consists of a one dimensional array of bits organized into bytes. The applications that send and receive messages, and the broker that provides additional message processing between sender and receiver, place a particular interpretation on the bytes of each message, and their order.

When a broker receives a message, its first task is to pass the message to a message parser. This reads the string of bits and converts them to a tree format. The tree format is easier to understand and manipulate, but contains identical content to the bits from which it is formed. When a broker delivers a message to a recipient, the message is converted back into a bit-stream.

A message tree

A message tree is made up of a number of *elements*. At the top of the tree is the *root*: this has no *parent* and no *siblings*. The root is parent to a number of *child* elements. Each child must have a parent, it can have zero or more siblings (with which it shares its parent), and it can have zero or more children.

The tree structure of a message is shown in Figure 8 on page 30. The message root has two children, ElementA1 and ElementB1 (which are therefore siblings sharing a single parent). The child ElementA1 has three children (ElementA2, ElementB2, and ElementC2) and ElementB2 has a further child ElementC1.

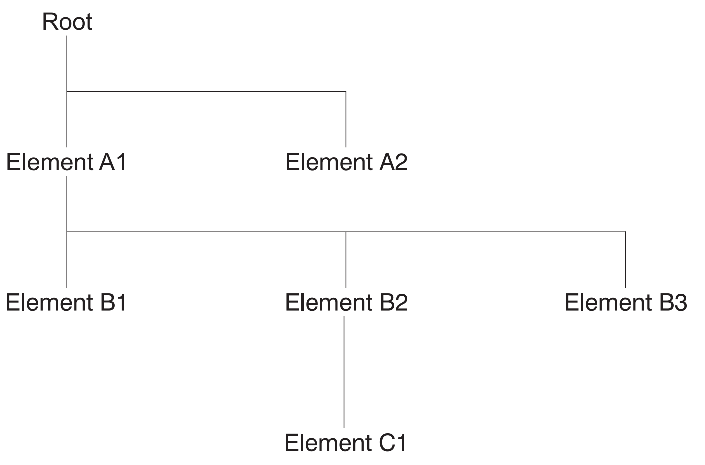


Figure 8. A message tree structure

This tree structure is explored further in the *MQSeries Integrator Programming Guide*.

Message domains

The messages supported by MQSeries Integrator are of three broad types, that are identified by a property of the message called the message domain:

1. A message can be *unstructured*: its message domain must be set to BLOB.
2. A message can be *self-defining*: its message domain must be set to XML.

   Two additional domains are included in this category to support JMS messages: the domain JMSMap can be used for jms_map messages and the domain JMSStream can be used for jms_stream messages.
3. A message can be *predefined*: its message domain must be set to one of:

   a. MRM
   b. NEON

   A predefined message has a *logical structure* and a *physical structure*:

   - The logical structure of a predefined message is a tree structure that demonstrates the hierarchical relationships between the components of a message (illustrated in Figure 8).
   - The physical structure of a message, which is also referred to as its *wire format*, is just a string of bits and bytes. Without the logical structure, the physical structure (the bit-stream) has no intrinsic meaning.

How a message is interpreted

When the message arrives in a broker, it is removed from the input queue by the MQInput node defined in the message flow that processes messages from this queue. It must be processed by an appropriate parser to decode the physical structure and create the logical structure.

The MQInput node determines what to do with each message:

- If the message has an MQRFH or MQRFH2 header following the MQMD header, the domain identified in the MQRFH2 header is used to decide which root message parser is invoked.
- If the message does not have an MQRFH or MQRFH2 header, but the properties of the MQInput node indicate the domain of the message, the parser specified by the node property is invoked.
- If the message has a valid MQMD, but the message body cannot be recognized, the message cannot be interpreted or parsed, and it is handled as a binary object (BLOB). See "Working with unstructured messages in the BLOB domain" on page 34 for more information about these messages.

Each message received must have an MQMD header, and can have zero or more additional headers. MQSeries Integrator provides a parser for each of the following MQSeries headers:

- MQCFH
- MQCIH
- MQDLH
- MQIIH
- MQMD
- MQMDE
- MQRFH
- MQRFH2
- MQRMH
- MQSAPH
- MQWIH
- SMQ_BMH

Further details about the support for these parsers is given in Appendix D, "MQSeries message header parsers" on page 381.

MQSeries Integrator also supports the use of additional parsers. You can create a message parser using a defined programming interface. This interface and the techniques you must employ to create your own "plug-in" parsers are described in the *MQSeries Integrator Programming Guide*. If you use your own parser, you must set up your MQInput node properties to identify your parser.

Unstructured messages in the BLOB domain

An unstructured message must have a message domain of *BLOB*. It has no known (or defined structure). These messages can be processed and routed by MQSeries Integrator, but the manipulation that you can perform is very limited.

You can perform some simple manipulation at the message level, and take other actions on the whole message. See "Working with unstructured messages in the BLOB domain" on page 34 for further details.

Self-defining messages in the XML domain

A self-defining message must have a message domain of *XML*. It carries the information about its content and structure within the message. Its definition is not held anywhere else.

When a self-defining message is received by the broker, it is handled by the XML parser, and a tree is created according to the XML definitions contained within that message.

A self-defining message is also known as a *generic XML message*. It does not have a recorded format.

A self-defining message can be handled by every IBM-supplied message processing node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow.

The message can also be manipulated, constructed, and reformatted by nodes in the message flow, using a specialized form of standard database Structured Query Language (SQL). This specialized form is known as Extended SQL, or ESQL, and supports MQSeries Integrator processing of the message structure. This means that although you do not have to define the message structure to the Control Center, you do have to understand the definition to be able to construct valid ESQL for message manipulation.

See “Working with messages in the XML domain” on page 34 for further details.

Predefined messages in the MRM domain

A predefined message in the MRM message domain must have its message domain set to *MRM*. It must be defined to the *Message Repository Manager*, a component of the Configuration Manager. You can define messages to the MRM domain using the Control Center (Message Sets view). The MRM maintains these messages in the message repository.

You can also predefine a message to the MRM in the XML message domain (the domain is defined as a message set property, as described in “Message set properties” on page 51). If you define a message to the XML domain, you can use all the facilities available to MRM domain messages to manipulate and reference the message in the nodes within your message flows in the Control Center. However, you are not expected to assign these message sets to a broker, nor to deploy them. Because the domain is set to XML, the XML parser is invoked by the broker and does not reference any external message definition.

An MRM message can be handled by every IBM-supplied message processing node. The whole message, or parts of the message, can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow. The message can be manipulated using ESQL defined within all message processing nodes that support manipulation (for example, compute and filter).

You can also transform any message in the MRM domain into any other format defined to the MRM using ESQL (in most cases, just one line of ESQL). This includes code page and encoding conversion: this capability provides the significant

| benefit that data conversion exists in MQSeries are not required to provide this
| function.

| For a fuller discussion of data conversion considerations, see Chapter 9, Planning
| your MQSeries Integrator network, in *MQSeries Integrator Introduction and*
| *Planning*. For an example of conversion in a message processing node, see
| Appendix C, "ESQL reference" on page 293.

Messages with a message domain of MRM have three other characteristics for further classification:

1. Message format

   Three message formats are supported by the MRM:

   a. A message can have a message format of CWF (Custom Wire Format).

      These messages are MRM representations of legacy datastructures created in the C or COBOL programming language, and imported into the MRM using the Control Center facilities. See "Importing legacy formats" on page 54 for details of how to complete this task.

      You can also create new messages using this format.

   b. A message can have a message format of PDF.

      This is a specialized format used predominantly in the finance industry. It does not have any connection with the Portable Document Format defined by Adobe (also known as PDF).

      If you already use messages of this format, you can continue to use them and process them by specifying this format in the definitions.

   c. A message can have a message format of XML.

      These messages are represented as XML documents. They conform to an XML DTD (Document Type Definition) that can be generated by the Control Center for documentation purposes.

2. Message set

   This identifies the message set to which each message belongs. This is specified as the message set *identifier*, not the message set name. When you define a message in the MRM message domain, you must define a message set that contains it. A message set can contain one or more related messages.

3. Message type

   The message type identifies the message definition within the set. It is the unique identifier for each message of this particular content and format.

See "Working with messages in the MRM domain" on page 42 for further details.

Predefined messages in the NEON domain

A predefined message in the NEON message domain must have its message domain set to *NEON*. It must be defined using the MQSeries Integrator Version 1 graphical utilities that are supplied with MQSeries Integrator Version 2. You can create new messages and use existing messages defined to the NEON domain.

A NEON message can be handled by every IBM-supplied message processing node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow. The

NEONFormatter node can be used to transform a NEON message. No other node can manipulate the message contents.

For further information about working with these messages, refer to Appendix E, "NEON Rules and Formatter" on page 397 and the *MQSeries Integrator Version 1.1 User's Guide*.

Working with unstructured messages in the BLOB domain

The structure and format of an unstructured message are not recognized or understood. Therefore the broker cannot perform any validation on messages in this domain, and a message in the BLOB domain cannot be manipulated within a message flow, except at the message level. For example, you can work with a substring of the message (for example, the 10th to 20th characters) but you cannot work at the field or element level, as these structures are not known.

You can, however, store the message full in a database, you can route the message according to topic (derived from the header), and you can add or remove headers from the message.

Working with messages in the XML domain

Self-defining or generic XML messages are those whose content are documents that adhere to the XML specification. The following sections describe how these messages are represented in a tree of syntax elements.

The following topics are discussed:

- "XML Declaration"
- "Document Type Declaration" on page 35
- "The XML message body" on page 39

The name elements used in this description (for example, XmlDecl) are provided by MQSeries Integrator for symbolic use within the SQL that defines the processing of message content that is to be performed by the nodes within a message flow (for example, a filter node). They are not a part of the XML specification itself. You can find examples of SQL syntax that handles the definition of generic XML messages in "Examples for generic XML messages" on page 364.

The information provided here does not provide a full definition or description of XML terminology, concepts, and message constructs: it is a summary that highlights aspects that are important when you use XML messages with MQSeries Integrator. For further information about XML, see the IBM web site at:

`http://www.ibm.com/developer/xml`

XML Declaration

The beginning of an XML message must contain what is called an XML declaration.

An XML declaration might take the following form in the XML bit-stream:

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"  ?>
```

The XML declaration must be at the beginning of every XML message.

XmlDecl

This is a name element that corresponds to the XML declaration itself. The XmlDecl element must be a child of the root element, and is the element that is written to a bit-stream first. This element can have three children of the following types:

1. Version

   The version element is a value element and stores the data corresponding to the version string in the actual declaration. It is always a child of the XmlDecl element. For example, for the declaration shown above the version element would contain the string value "1.0".

2. Standalone

   The standalone element is a value element and stores the data corresponding to the value of the standalone string in the declaration. It is always a child of the XmlDecl element. The values for the standalone element must be the string "yes" or "no".

   | "no" is the default: this means that processing of the message depends on an
   | external (DTD) reference.

3. Encoding

   The encoding element is also a value element and is always a child of the XmlDecl element. The value of the encoding element is a string which corresponds to the value of the encoding string in the declaration. In the example shown above the encoding element would have a value of "UTF-8".

   | **Note:** MQSeries encodings cannot be specified in this element.

Document Type Declaration

The document type declaration (DTD) of an XML message is represented by a syntax element of type DocTypeDecl and its children and descendants. These comprise the DOCTYPE construct.

Only internal DTD subsets are represented in the syntax element tree. External DTD subsets (identified by the SystemID or PublicId elements described below) can be referenced in the message but those referenced are not resolved in the MQSeries Integrator run-time environment.

DocTypeDecl

The DocTypeDecl is a named element and must be a child of the root element. It is written to the bit-stream before the element that represents the body of the document during serialization. The following can be specified within this element:

1. IntSubset

   IntSubset is a named element that groups all of those elements that represent the DTD constructs contained in the internal subset of the message. Although the IntSubset element is a named element its name is not relevant.

2. SystemId

   SystemId is a value element and is used to represent a general system identifier construct found in an XML message. It can be a part of a DocTypeDecl or a NotationDecl element. The value of the SystemId is a URI, and is typically a URL or the name of a file on the current system. A system identifier of the form `SYSTEM "Note.dtd"` has a string value of "Note.dtd"

3. PublicId

   PublicId represents a general public identifier construct found in an XML message. It can be a part of a DocTypeDecl or a NotationDecl element. The value of the PublicId is typically a URL.

NotationDecl

The NotationDecl element represents a notation declaration in an XML message. It is a name element whose name corresponds to the name given with the notation declaration. It must have a SystemId as a child, and it can optionally have a child element of type PublicId.

Entities

Entities in the DTD are represented by one of five named element types described below.

1. ParameterEntityDecl

   The ParameterEntityDecl represents a parameter entity definition in the internal subset of the DTD. It is a named element and has a single child element that is of type EntityDeclValue. For parameter entities the name of the entity does not include the percent sign %. In XML a parameter entity declaration takes the form:

   ```
   <!ENTITY % inline "#PCDATA | emphasis | link">
   ```

2. ExternalParameterEntityDecl

   The ExternalParameterEntityDecl represents a parameter entity definition where the entity definition is contained externally to the current message. It is a named element and has a child of type SystemId. It can also have a child of type PublicId. The name of the entity does not include the percent sign %. In XML an external parameter entity declaration takes the form:

   ```
   <!ENTITY % bookDef SYSTEM "BOOKDEF.DTD">
   ```

   This is represented by an ExternalParameterEntityDecl element of name bookDef with a single child of type SystemId with a string value of “BOOKDEF.DTD”.

3. EntityDecl

   The EntityDecl element represents a general entity and is declared in the internal subset of the DTD. It is a named element and has a single child element which is of type EntityDeclValue.

   An entity declaration of the form:

   ```
   <!ENTITY bookTitle "User Guide">
   ```

   has an EntityDecl element of name "bookTitle", and a child element of type EntityDeclValue with a string value of “User Guide”.

4. ExternalEntityDecl

   The ExternalEntityDecl element represents a general entity where the entity definition is contained externally to the current message. It is a named element and has a child of type SystemId. It can also have a child of type PublicId.

   An external entity declaration of the form:

   ```
   <!ENTITY  bookAppendix SYSTEM "appendix.txt">
   ```

has an EntityDecl element of name "bookAppendix" and a child element of type SystemId with a string value of "appendix.txt".

5. UnparsedEntityDecl

   An unparsed entity is an external entity whose external reference is not parsed by an XML processor.

   The UnparsedEntityDecl is named element. It has a child of type SystemId and optionally a child of type PublicId. The presence of NDATA after the SystemId in the entity declaration indicates that this entity is not parsed by the XML processor. After NDATA is the name of a corresponding notation declaration. In XML an unparsed entity declaration takes the form:

   ```
       <!ENTITY pic SYSTEM "scheme.gif" NDATA gif>
   ```

   - NotationReference

     The NotationReference name element represents a reference to a notation declaration from within an UnparsedEntityDecl element. It is always a child of an UnparsedEntityDecl element.

EntityDeclValue: This value element represents the value of an EntityDecl, or a ParameterEntityDecl defined internally in the DOCTYPE construct. It is always a child of an element of one of those types, and is a value element. For the following entity:

```
    <!ENTITY  bookTitle "User Guide">
```

the EntityDeclValue element has the string value "User Guide".

ElementDef

The ElementDef name-value element represents the <!ELEMENT construct in a DTD. The name of the element that is defined corresponds to the name member of the syntax element. The value member corresponds to the element definition.

AttributeList

The AttributeList name element represents the <!ATTLIST construct in a DTD. The name of the AttributeList element corresponds to the name of the element for which the list of attributes is being defined.

AttributeDef

The AttributeDef name element describes the definition of an attribute within a <!ATTLIST construct. It is always a child of the AttributeList element. The name of the syntax element is the name of the attribute being defined. It can have three children:

1. AttributeDefValue

   For attributes of type CDATA (see AttributeDefType below) the AttributeDefValue gives the default value of the attribute.

2. AttributeDefDefaultType

   The AttributeDefDefaultType syntax element is a value element which represents the attribute default as defined under the attribute definition. The value can be one of the following strings:

   - #REQUIRED
   - #IMPLIED

- #FIXED

3. AttributeDefType

   The AttributeDefType syntax element is a name-value element whose name corresponds to the attribute type found in the attribute definition. Possible values for the name are:

   - CDATA
   - ID
   - IDREF
   - IDREFS
   - ENTITY
   - ENTITIES
   - NMTOKEN
   - NMTOKENS
   - NOTATION

If there is an enumeration present for the attribute definition the entire enumeration string is held as a string in the value member of the name-value syntax element. The value string starts with an open bracket "{" and ends with a close bracket "}". Each entry in the enumeration string will be separated by a '|' character. For an enumerated type that is not a NOTATION, the name member of the syntax element is empty.

DocTypePI and ProcessingInstruction

The DocTypePI element represents a processing instruction found within the DTD. The ProcessingInstruction element represents a processing instruction found in the XML message body.

Both of these elements are name-value elements. In both cases, the name of the element is used to store the processing instruction target name, and the value contains the character data of the processing instruction. The value of the element can be empty. The name cannot be the string "XML" or any uppercase or lowercase variation of "XML".

DocTypeWhiteSpace and WhiteSpace

The DocTypeWhiteSpace element represents whitespace found inside the DTD that is not represented by any other element. The WhiteSpace element represents any white space characters found in the message body that is not represented by any other element. Both are value elements.

For example, white space within the body of the message is reported as element content using the pcdata element type, but white space characters found between the XML declaration and the beginning of the message body are represented by the WhiteSpace element.

```
    <?xml version="1.0"?>       <BODY>....</BODY>
```

The characters between 1.0"?>" and <BODY> are represented by the WhiteSpace element. White space characters found within a DocType between two definitions are represented by the DocTypeWhiteSpace element.

```
  <!DOCTYPE Note SYSTEM "Note.DTD"[
  <!ENTITY % bookDef SYSTEM "BOOKDEF.DTD">  <!ENTITY bookTitle "User Guide"> ]>
```

The characters between DTD"> and <!ENTITY are represented by the DocTypeWhiteSpace element.

DocTypeComment and Comment

Comments in the XML message are represented by the Comment and DocTypeComment elements. The former is used within the message body, the latter within the DTD. Both element types are value elements where the value string contains the comment text.

The XML message body

Every XML message must have a body element. The body element is a top level XML element which encapsulates the whole of the body. XML elements are represented in the syntax element tree with a type of "tag".

- tag

  The tag syntax element is the default name element supported by the XML parser and is the most common element. This element can have many children of many different types. XML attributes that are attached to an XML element are represented by a series of "attr" elements that are children of the tag element. Similarly, sections of PCDATA which are content of the XML element are represented by syntax elements of type "pcdata". "tag" elements can also have other tag elements as children.

  – attr

    The attr element is the principal name-value element supported by the XML parser. It is used to represent attributes that are associated with elements in the XML message. The name and value of the syntax element correspond to the name and value of the attribute being represented. "attr" elements have no children and must always be children of a "tag" element.

  – pcdata

    Element content is represented by the pcdata value element. There can be more than one pcdata element child of a single tag element. In these cases they would be separated by any syntax elements that represent XML constructs allowed within element content, including "tags", "ProcessingInstruction", "Cdata", "EntityDecl".

The following XML illustrates an extract of message body:

```
<PERSON age="32" height="172cm">
  <FIRSTNAME>Cormac</FIRSTNAME>
  <SECONDNAME>Keogh</SECONDNAME>
</PERSON>
```

This is represented in the syntax element tree as:

- One **"tag"** element with a name of "PERSON". This tag has seven children.
  1. Two **attr** (name-value) with names "age" and "height" and string values "32" and "172cm" respectively.
  2. One **pcdata** (value) element with string value containing the white space character data found between "172cm"> and <FIRST .
  3. One **tag** (name) with a name "FIRSTNAME". This tag has one child:
     – One **pcdata** (value) containing the string value "Cormac".

4. One **pcdata** (value) element with string value containing the white space character data found between TNAME> and <SECOND.
5. One **tag** (name) with a name "SECONDNAME". This tag has 1 child:
   – One **pcdata** (value) containing the string value "Keogh".
6. One **pcdata** (value) element with string value containing the white space character data found between DNAME> and </PERSO.

ProcessingInstruction
This is described in "DocTypePI and ProcessingInstruction" on page 38.

WhiteSpace
This is described in "DocTypeWhiteSpace and WhiteSpace" on page 38.

Comment
This is described in "DocTypeComment and Comment" on page 39.

AsisElementContent
Normally an XML processor must replace any occurrences of the characters ampersand (&), less than (<), greater than (>), double quote ("), and apostrophe (') with an escape sequence that is used to represent them (&amp;, &lt;, &gt;, &quot;, and &apos;). The escape sequences are defined as entities.

The AsisElementContent is a value element that is similar to the pcdata element but provides a means to suppress this behavior for the content of an element. Occurrences of any of the characters in the value of an AsisElementContent element are substituted by their appropriate entity reference.

CDataSection
CData sections in the XML message are represented by the CDataSection value element. The content of the CDataSection element is the value of the CDataSection element without the <![CDATA[that marks the beginning, and without the]]> that marks the end of the Cdata section.

For example, the following Cdata section:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

is represented by a CDataSection element with a string value of:

```
"<greeting>Hello, world!</greeting>"
```

Unlike pcdata, occurrences of <, >, &, " and ' are not translated to their escape sequences when the Cdata section is written out to a serialized message.

EntityReferenceStart and EntityReferenceEnd
When an entity is encountered in the XML message it is reported in the syntax element tree in expanded form. In order to determine if a section of the tree has been derived from an expanded entity, a couple of marker elements are placed in the tree to denote the beginning and end of an entity's expansion.

- The EntityReferenceStart element is a value element that marks the beginning of an entity expansion.
- The EntityReferenceEnd element is a value element which marks the end of an entity expansion.

The value of both elements corresponds to the name of the entity being expanded. Any syntax elements found between these two place holders, and their children have been derived from the expansion of the entity in question.

Working with messages in the MRM domain

This section describes how to work with messages that you define, or definitions you import, using the facilities of the Control Center. It covers the following introductory topics:

It also describes the following tasks:

An overview of the message definition process

The message definition process is managed by the *Message Repository Manager* (MRM) component of the Control Center.

When you create or modify message definitions using the Control Center, the MRM stores them in the *message repository*, a set of tables in a database created and maintained by the Configuration Manager.[2]

Each message definition is created within, and belongs to, a *message set*, which is simply an organizational grouping of related messages. A message set includes the definitions of one or more related messages, typically those used by a single

2 The configuration repository and message repository are implemented using an IBM DB2® Universal Database® for Windows NT.

application. You construct each message using a set of building blocks, known as *message components*, some of which are supplied with MQSeries Integrator (the simple types) and some of which you define using the Control Center (the compound types).

So, for example, for a banking application you could define simple elements, such as Account Number and Account Balance, then include those simple elements in a compound element, such as Customer Record. The Account Number, Account Balance, and Customer Record elements would all be reusable by other message definitions within the same message set. The components of a message are described in detail in "The components of a message definition."

You must *assign* message sets to those brokers that need to understand them. When you *deploy* message-set assignments in the broker domain, the MRM constructs a *message dictionary* from each message set (one dictionary plus one CWF descriptor file for each set) and sends it to each broker that needs access to the message definitions.

The message model

The MQSeries Integrator message model provides a platform- and language-independent way of defining logical messages that represent structured business information.

In this message model, a message definition comprises separate, reusable *message components*. The relationship between components is defined as being either a *member* relationship or a *reference* relationship.

Reference relationship

A reference relationship is a defining relationship between two components. For example, an element component of type STRING has a reference relationship to an element length component that defines the length of the element.

Reference relationships are always mandatory.

Member relationship

A member relationship is a parent-child relationship between two components. For example, a (parent) compound type has a member relationship with one or more (child) elements. Note that the member relationship is always expressed as an attribute of the parent, not of the child.

Member relationships are always optional.

The components of a message definition

The components of a message definition are described in the remainder of this section. For each component, the reference and member relationships are identified.

Message component: The message component defines both the business meaning and the format of a single unit of information to be exchanged between applications.

- A message component has a reference relationship to a single type component (a compound type) that defines the content of the message. It can also have a reference relationship to an element qualifier.
- A message component has no member relationships.

Once a message component has been created, the reference of the type component cannot be changed.

Element component: The element component defines both the business meaning and the format of a single unit of information within a message.

- An element component has a reference relationship to a single type component (a simple type or a compound type) that defines the content of the element.

  An element component also has a reference relationship to an element length component, if the element is of simple type STRING.
- An element component can have a member relationship to one or more (child) element valid value components, which must have the same type as the element.

Once an element has been created, the identifiers of the type and element length components to which it refers cannot be changed.

Type component: The type component defines the format or content of a message or an element. A type can be a *simple* type or a *compound* type.

Simple type
Is a basic data type supported by the run-time message parsers. The simple types are: STRING, INTEGER, FLOAT, BOOLEAN, and BINARY. The simple types are created automatically when you create a message set.

Compound type
Is a structure made up of one or more element components.

- A type component has no reference relationships.
- A compound type component has member relationships to one or more (child) elements.

Element length component: The element length component defines a maximum length value that completes the definition of any element of the simple type STRING.

- An element length component has no reference relationships.
- An element length component has no member relationships.

Category component: The category component groups messages within a message set, typically by business function. The extraction and generation functions of the MRM can produce their output by category.

- A category component has no reference relationships.
- A category component can have member relationships to one or more (child) messages.

Element valid value component: The element valid value component defines either a single value, or a range of values. One or more element valid value components can be associated with an element, or with an element qualifier, or both. One element valid value can define the default value of an element.

- An element valid value component has a reference relationship to a type component that defines the content of any element component to which the value may apply. (In other words, a valid value of type STRING can only be applied to an element of type STRING, or to its element qualifier.)
- An element valid value component has no member relationships.

Element qualifier component: An element qualifier component provides additional information that qualifies the definition of an element component. An element qualifier component can be associated with a specific element component within a specific message component to qualify its use in that message component only. For example, an element qualifier can specify that a specific element is mandatory in a specific message, even though it is optional elsewhere.

- An element qualifier component has a required reference relationship to an element component. The element qualifier component can be used to qualify the use of its related element in a specific instance.
- An element qualifier component can have member relationships to zero or more element valid value components, that must have the same type as the referenced element. If valid values are present for an element qualifier in a message, they apply to the related element within the specific message only (overriding those defined for the element). One such element valid value can denote a default value.

Figure 9 on page 46 shows all possible components of a message definition and summarizes the relationships between them.

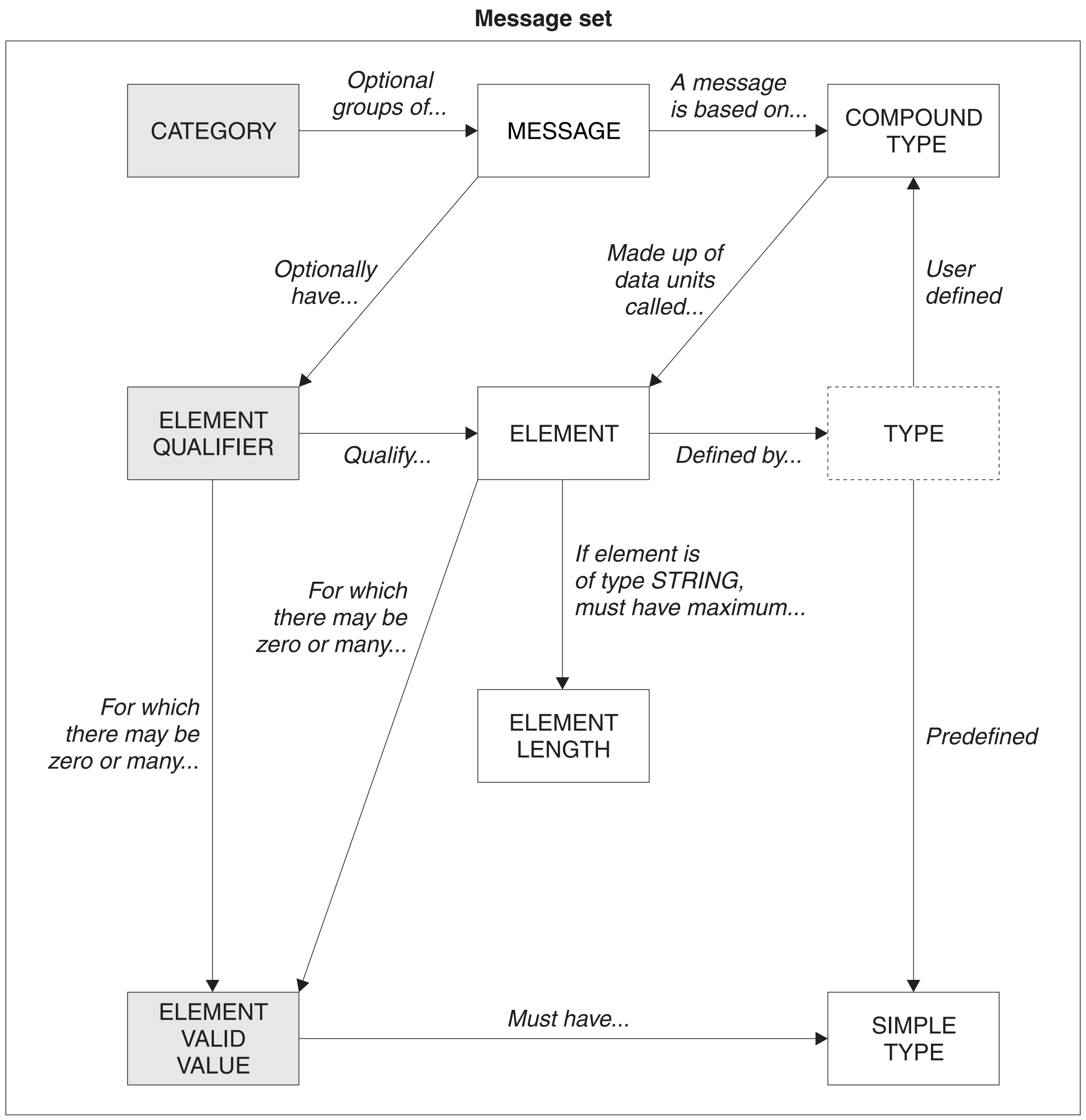


Figure 9. The components of a message

Component identifiers and names

Each component of a message definition has an identifier and a name.

Component identifier

Identifies a component uniquely within a message set. No two components in a message set can have the same identifier, and no two components of the same class (for example, two elements or two categories) can have identifiers that differ only by case. For example, you cannot define an element with the identifier “ADDRESS” and an element with the identifier “address” in the same message set.

In the case of element components, the element identifier is used in application programs to access data values assigned to the element.

An identifier must begin within an alphabetic character. The remainder of the value, up to a maximum of 254 characters, can contain alphanumeric, underscore (_), and period (.) characters. Other characters, including space characters, are not valid.

You cannot change the identifier of a component.

Component name
Is a descriptive name for a component. It is typically the full name of a component (for example, “Street Name” or “Account Number Length”), in contrast to the component identifier, which is often an abbreviated name and subject to environmental conventions.

You can change the name of a component (using the **Rename** action).

An example message definition

To illustrate the concepts introduced in this section, consider this example of a simple message:

```
AddressesMessage
   HomeAddress
      Line     (1 or more)   (STRING 50)
      Country  (1 only)      (STRING 50)
      ZipCode  (0 or 1)      (STRING 20)
   WorkAddress
      Line     (1 or more)   (STRING 50)
      Country  (1 only)      (STRING 50)
      ZipCode  (0 or 1)      (STRING 20)
```

Some items to note about this message:

- The top-level elements HomeAddress and WorkAddress have the same substructure, which you can define by creating a compound type component called Address that contains the common elements Line, Country, and ZipCode. The compound type Address would be referenced by the top-level elements HomeAddress and WorkAddress.
- The elements Line, Country, and ZipCode all reference the simple type STRING, which is created by default when a message set is created. These elements must also reference an element length component.

If you create a message definition from the bottom up (that is, starting with the lowest-level components and working up to the top of the hierarchy), you are guaranteed to create a referenced component before you create the component that contains the reference.

The components of our example AddressesMessage would be created in the following order:

1. Simple type STRING (created by default when the message set is created)
2. Element length components, in any order:
   a. Maximum Length 50
   b. Maximum Length 20

3. Element components, in any order:
   a. Element Line, referencing simple type STRING and element length Maximum Length 50.
   b. Element Country, referencing simple type STRING and element length Maximum Length 50.
   c. Element Zip Code, referencing simple type STRING and element length Maximum Length 20.
4. Compound type component Address, with member relationships to the following child elements:
   - Element component Line
   - Element component Country
   - Element component Zip Code
5. Element components, in any order:
   a. Element Home Address, referencing compound type Address
   b. Element Work Address, referencing compound type Address
6. Compound type component HomeAndWork, with member relationships to the following child elements:
   - Element Home Address
   - Element Work Address
7. Message component AddressesMessage, referencing compound type HomeAndWork.

Clearly, before you use the Control Center to define your messages, you need to have done the data analysis that will enable you to create complete and accurate definitions in an efficient manner.

For information about using the Control Center to define messages, see “Creating messages” on page 59.

Message sets

A message set contains the definitions of one or more messages, plus the definitions of the components that make up those messages. A typical message set contains the definitions of all messages required by a single application. The run-time message dictionary provided by the MRM to the run-time message parsers contains definitions for all messages in a single message set.

In common with the components of a message, message sets must have a name. They must also have a level number that identifies this version of the message set. Message set properties, related to the data model layers, are described in “Message set properties” on page 51.

The data model layers

So far, we have discussed the concepts underlying the MRM’s message model. However, the message set contains additional “layers” of information that support related MRM functions. These are:

- The documentation layer
- The C language layer
- The COBOL language layer
- The run-time layer

- The Custom Wire Format layer

These layers of information are visible to you in the Properties pane of the **Message Sets** view, as described in “The Message Sets view” on page 56. They are described in more detail in the following sections.

The documentation layer

When you define each message component and each message set, you have the opportunity to provide a short description or a long description, or both, of that component or message set. You are recommended to use these description fields to describe the business meaning of the object, and to record any business rules that govern their use.

The documentation extractors of the MRM include this information in generated documentation. For more information about generating documentation from the MRM, see “Generating documentation” on page 78.

The C language layer

The MRM can generate C header files from the message definitions you create that can be used in messaging applications developed in C language. You specify values for properties of the components to support this function. These properties are mandatory. Although you might never wish to generate language bindings, the message set is incomplete if these properties are missing.

For the category component, you specify:

Category Header File Name
: Provides the name of the header file into which structure definitions for all messages in this category are generated.

Include in Main Header
: Specifies whether this header file is included from the main header file for the message set.

For the element component, you specify:

C Language Name
: Provides the name used for this element as a field within C structure definitions. By default, the element identifier is used.

For the type component, you specify:

C Language Name
: Provides the name for the C structure definition that is generated for the type. By default, the type identifier is used.

File Name
: Provides a name for a header file to be generated containing a structure definition for the type. This value is optional, and is not usually specified: the structure definitions for type components appear only in the category header files.

The COBOL language layer

The MRM can generate COBOL copy books from the message definitions you create that can be used in messaging applications developed in COBOL language. You specify values for properties of the components to support this function. These properties are mandatory. Although you might never wish to generate language bindings, the message set is incomplete if these properties are missing.

For the category component, you specify:

Category Copy Book Name
: Provides the name of the copy book file into which structure definitions for all messages in this category are generated.

For the message component, you specify:

COBOL Language Name
: Provides the name used for the COBOL structure definition that is generated for the message. By default, the message identifier is used.

Message Copy Book Name
: Provides the name of the copy book file into which the structure definition for the message is generated.

For the element component, you specify:

COBOL Language Name
: Provides the name used for this element as a field within COBOL structure definitions. By default, the element identifier is used.

For the type component, you specify:

COBOL Language Name
: Provides the name for the COBOL structure definition that is generated for the type. By default, the type identifier is used.

Structure Copy Book Name
: Provides a copy book file name into which the structure definition for the type is generated.

The Custom Wire Format layer

The CWF layer defines additional information that is used to define the mapping between logical messages and legacy message formats defined by applications that use data structure features of languages such as C and COBOL to populate the message structure. This information is used to produce a wire format descriptor that can be used by a run-time message parser.

You specify some of the following properties for each element that is a child in a type. For example, if the logical type is string, the physical type packed decimal is not applicable. Similarly, if the logical type is float, and the physical type is extended decimal, and the signed field is set to True, then the sign orientation field is applicable. The properties that might be applicable are:

- Physical type
- Length (Count or Value Of)
- Signed and Sign Orientation
- Skip count
- Byte alignment
- String justification

- Padding character
- Virtual decimal point
- Repeat (Count or Value Of)

Message set properties

The properties of a message set are displayed on several tabs, displayed in the Properties pane of the Message Sets view. The tabs are:

Basic tab (identified by message set name)

This tab defines basic properties for the message set. These are:

Level

Is a numeric value that identifies the version of the message set. If you are creating a second (or subsequent) version, you must set this property to a value higher than the highest existing level number.

Finalized

Indicates if the message set has been finalized (true) or not (false).

Freeze Time Stamp

Indicates the date and time when the message set was frozen. If this is not set, the message set has never been frozen.

Identifier

Is the identifier by which the message is known (used in addition to its name). It is a unique value, and is automatically allocated when the message set is created.

Base Message Set

Is the base message set from which this message set's definition is derived. All components defined in the base message set are also defined in this message set.

C Language tab

The MRM can generate C header files from the message definitions you create that can be used in messaging applications developed in C language.

This tab defines properties that identify names for header files generated from this message set using the **Message Sets —> Generate** command from the Control Center. These properties are mandatory.

Main Header File Name

Is the name of the generated header file that contains C structure definitions of the messages in this message set.

Orphan Header File Name

Is the name of the generated header file that contains definitions of C structures (types) that are not used by any message in this message set.

COBOL Language tab

The MRM can generate COBOL copy books from the message definitions you create that can be used in messaging applications developed in COBOL language.

This tab defines the property Main Copy Book Name that identifies the name of the main copy book generated from this message set by using the **Message Sets —> Generate** command from the Control Center. These properties are mandatory.

Custom Wire Format tab

The CWF defines additional information that is used to define the mapping between logical messages and legacy message formats defined by applications that use data structure features of languages such as C and COBOL to populate the message structure. This information is used to produce a wire format descriptor that can be used by a run-time message parser.

The properties on this tab are:

Custom Wire Format Identifier

You are recommended to use the default value of CWF. You can extend this identifier to eight characters if you choose, but the first three characters must always be CWF. You cannot specify embedded blanks or special characters.

Byte Alignment Pad

This defines the default padding character used for this message set. The default is 0.

Boolean True and False Values

Boolean True and False Values define the True and False values to be used by every element of type Boolean in this message set. The defaults are 00000001 and 00000000.

Boolean True and False Values must be the same length, and can be between 1 and 4 bytes long. They must be defined in half-byte values and you must specify an even number (to define a number of whole bytes). For example, if you want your Boolean values to be ASCII characters Y and N, you would enter the two characters 54 in the True field and the two characters 46 in the False field.

Run Time tab

Defines the property parser that identifies the message domain for this message set, and therefore the parser that the broker invokes to interpret the messages. Two options are available:

- MRM. This is the default and usual case.
- XML. You can set the domain and parser to XML if you want to define the message set for easier manipulation and reference in the message flows in which it is used (as described in “Predefined messages in the MRM domain” on page 32.

Description tab

Includes a short description and a long description of the message set. Both are optional properties.

When you define each message component and each message set, you have the opportunity to provide a short description or a long description, or both, of that component or message set. You are recommended to use these description fields to describe the business meaning of the object, and to record any business rules that govern their use.

The documentation extractors of the MRM include this information in generated documentation. For more information about generating documentation from the MRM, see “Generating documentation” on page 78.

Message set states

The state of a message set varies in line with development, testing, and production cycles.

The states of a message set are:

Normal
: If a message set is not locked (checked out), frozen, or finalized, it is considered to be in normal working state (but this state is not specified in the view). It can be checked out, updated, and checked in. This state is not explicitly stated in the Control Center: it is inferred by the message not being locked, frozen, or finalized.

  Note: When you create a new message set, it is automatically checked in, then checked out, and you will see the key icon appear against the new message set. A message set is never shown in new state (with the new icon against it). However, components of the message set (for example, an element) do appear as new when they are first created.

Locked
: The state of a message set while it is checked out (locked) by a Control Center user. A message set must be in this state before you can change any of its properties. A message set must also be locked before you can freeze it.

Frozen
: The state of a message set that is not expected to change (for example, on entry to a test phase). Neither the message set itself nor its contents can be changed while it is in this state, nor can they be checked out. A message set can be unfrozen by selecting **Unfreeze** if subsequent change is required. Frozen state is indicated by the existence of a Freeze date in the message properties.

  An attempt to freeze a message set fails if any component of the message set is checked out or if any of the message definitions it contains is incomplete.

  A message set must move to the frozen state from the locked state, and both state changes must be requested by the same user.

Finalized
: A message set and its components in this state cannot be changed or checked out. Finalized state is indicated by the message property Finalized set to true.

  An attempt to finalize a message set fails if any component of the message set is checked out or if any of the message definitions it contains is incomplete.

  A message set can move to the finalized state from any other state. However, if it moves from the locked state to the finalized state, both state changes must be requested by the same user.

Once a message set is finalized, no further changes can be made to its contents. However, you can create a new message set based on the finalized message set, within which you can define new messages. You can also make limited changes to the existing messages in the new message set. For more information, see "Message set versioning" on page 54.

Message set versioning

A message set can be based on another message set, provided that the message set on which it is based has been finalized. You might want to use this facility to maintain separate versions of a message set, reflecting the evolution of a message set through maintenance and other fixes.

When a message set is based on another message set, it contains a copy of the complete contents of the base message set. Within the new message set, new messages can be defined, and limited modifications can be performed on existing messages. A separate run-time dictionary is produced for the new message set.

A message set can have the same name as another message set in the same message repository if:

- The new message set is based on the message set of the same name.
- The message set on which it is based has a higher level number than any other message set with the same name in the same repository.
- The level number of the new message set is higher than that of the message set on which it is based.

Importing legacy formats

The MRM provides C and COBOL language importers, which you can use to help you create a message set containing message definitions that originate from legacy applications. Such applications are typically those that use C or COBOL data structures to populate messages. The source code of those applications must be available to the import function of the MRM.

The import function parses the source code files, isolates the data structure definitions, and creates logical definitions that correspond to those data structures. It also sets the appropriate CWF properties to define the mapping between the logical definitions and the physical message format, as defined by the C or COBOL data structures.

A compound type is created for each data structure, and elements and element lengths are created for each field within the data structure. For more detailed information about the way in which C and COBOL data structures are interpreted by the MRM language importers, see Appendix B, “C and COBOL default mappings” on page 287.

A report is generated by the import function that describes all the definitions that have been created. It includes information about errors or conflicts within the definitions. You can elect to produce this report without committing any changes to the message set. You are recommended to do this and check the report before running the complete import process.

When the import process is complete, you need only to create a message component for each compound type that defines a complete message; all other components are created automatically. However, you are recommended to review your message definition, and edit it if necessary, to ensure that it meets your needs.

See “Importing message definitions” on page 75 for details of how to import these structures.

Note: You cannot import message sets created by another Control Center user into your Control Center session. This function is only supported by the message set import and export command (**mqsimrmimpexp**), which is described in the *MQSeries Integrator Administration Guide*.

Generating MRM message set Document Type Descriptors (DTDs)

A broker accesses a message set definition in a message dictionary (each message set is deployed in a separate dictionary). Client applications cannot access message dictionaries. They must use one of following two options for accessing the definitions used by the broker.

- You can generate an XML Document Type Descriptor (DTD) from the message set within the message repository. For information about this task, see “Generating MRM message set definitions in XML DTDs” on page 76.
- If you have created the MRM definitions by importing C or COBOL data structures, your applications can continue to use those data structures. For information about importing, see “Importing message definitions” on page 75.

Authorization to work with Messages

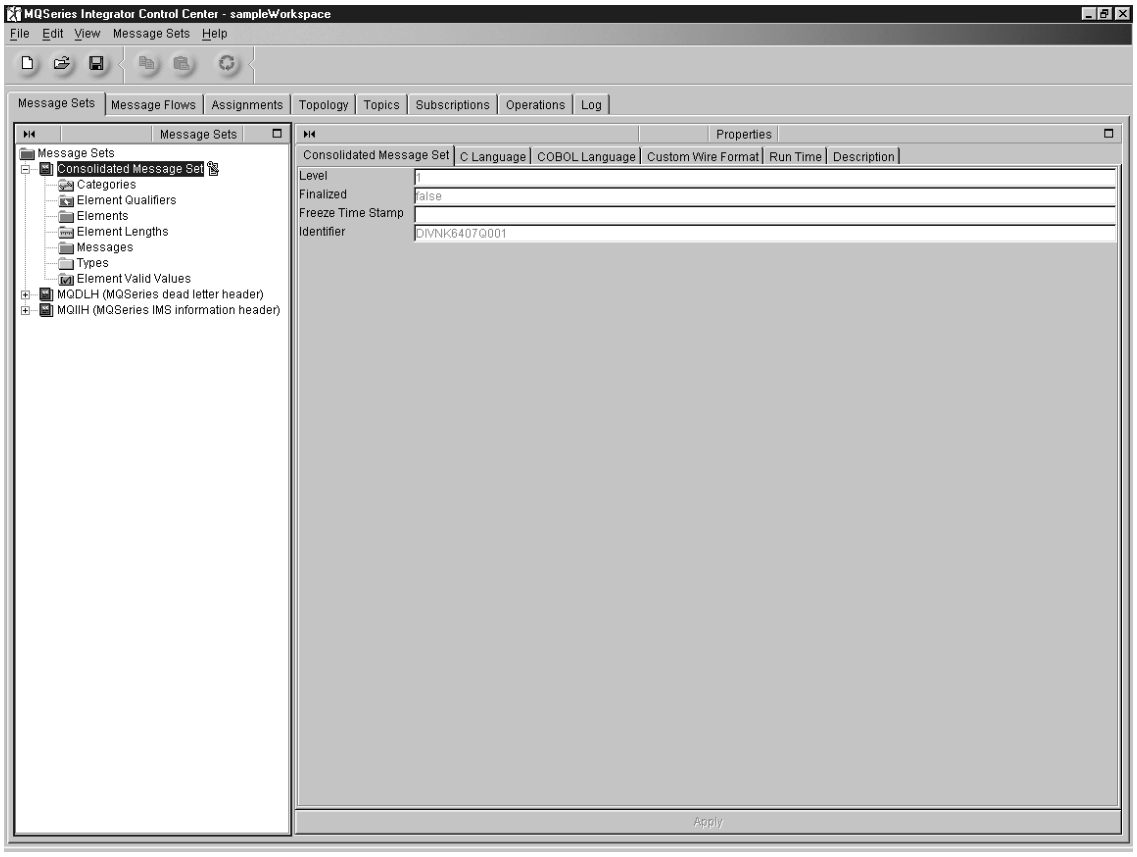
To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Message flow and message set developer**
  - **All roles**

  For information about setting your user role, see “Setting user roles” on page 13.
- Be a member of the MQSeries Integrator group **mqbrdevt**

The Message Sets view

The **Message Sets** view is the Control Center interface to the MRM. To display the **Message Sets** view, click the **Message Sets** tab in the Control Center. Figure 10 shows an example of the **Message Sets** view.



Message Sets
- Consolidated Message Set
  - Categories
  - Element Qualifiers
  - Elements
  - Element Lengths
  - Messages
  - Types
  - Element Valid Values
- MQDLH (MQSeries dead letter header)
- MQIIH (MQSeries IMS information header)

Level	1
Finalized	false
Freeze Time Stamp	
Identifier	DIVNK6407Q001

Figure 10. The Message Sets view. The left-hand pane, the Message Sets pane, shows a tree view of the message sets in your workspace. The right-hand pane, the Properties pane, displays the properties of the currently selected entry in the Message Sets pane.

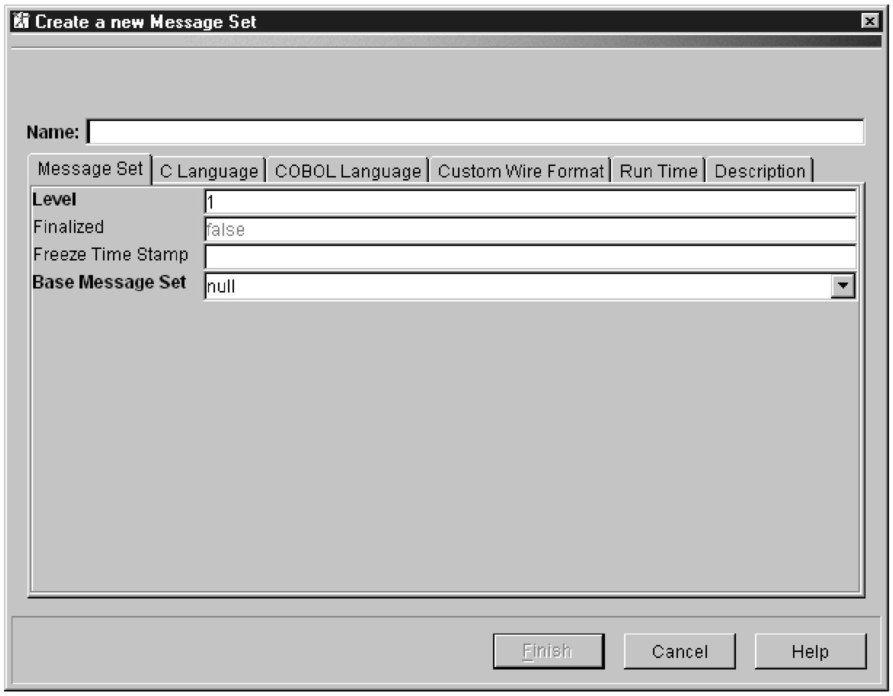
When you click on the plus sign (+) to the left of a message set folder, the contents of the folder are displayed. Each message set folder contains an entry for each of the seven message components. When you create new components within the message set, they appear under the relevant component entry. For example, if you create an element, it appears under the Elements folder within the message set. New components have the **New** icon against them.

Creating message sets

To create a new message set:

1. In the Message Sets pane, right click the **Message Sets** root and click **Create —> Message Set**.[3]

   The **Create a new Message Set** dialog is displayed, as shown in Figure 11.



Level	1
Finalized	false
Freeze Time Stamp	
Base Message Set	null

Figure 11. The Create a new Message Set dialog

2. Complete the fields on the initial panel:
   - In the **Name** field, type a name for this new message set. This must follow the naming rules described in “Naming Control Center resources” on page 20. Enter the name ’Consolidated Message Set’ or another name of your choice.
   - Specify a level number if appropriate. For information about setting the level of a message set, see “Message set versioning” on page 54.
   - This is a new message set, so the Finalized and Freeze Time Stamp fields can be ignored.
   - If the message set is to be based on another, finalized message set, select that message set from the Base Message Set drop-down list. This list shows message sets in your workspace, including those for the standard MQSeries headers, which are provided by MQSeries Integrator.
3. Click the **Run Time** tab.

   Select the message parser for messages belonging to this set from the drop-down list. The default is MRM: you can also select XML. For more information about tihs property, see “Message set properties” on page 51.

[3] Alternatively, you can click the **Message Sets** menu in the taskbar and click **Create —> Message Set**.

4. Click the **C Language** tab and complete the Main Header File Name and Orphan Header File Name fields. The defaults for these are CSTRUCTS.H and ORPHANS.H. You can overtype these defaults if you choose. These properties are mandatory, and are used when you generate C language header files from this message set. For information about these fields, see “Message set properties” on page 51.
5. Click the **COBOL Language** tab and specify a name for the copy book in the Main Copy Book Name field. The default is MAINBOOK.COPY. You can overtype this default if you choose. These properties are mandatory, and are used when you generate COBOL language header files from this message set. For information about this field, see “Message set properties” on page 51.
6. If this message set is to contain legacy messages (for example, if message definitions are to be imported into this message set), you need to specify the CWF values.

   Click the **Custom Wire Format** tab. The default Custom Wire Format Identifier is CWF. You are recommended to use this default value. If you choose the change this value, you must set a value of 8 characters or less, and the first three characters must be CWF. For more information about the CWF values, see “Message set properties” on page 51.
7. If you want to provide a description of this message set, click the **Description** tab. Any description text you provide here is included in documentation generated by the MRM, as described in “Message set properties” on page 51.

   Type a short description, or a long description, or both.
8. Click **Finish** to complete the definition of this message set.

A locked entry appears under **Message Sets** root in the Message Sets pane. When the new message set entry is highlighted in the Message Sets pane, its properties appear in the Properties pane. Notice that an identifier for the new message set has been generated automatically by the MRM.

When you are ready to share a new message set with other Control Center users, you check it into the shared configuration. You can do this before the message set contains any message definitions, if you wish. For more information about checking in message sets, see “Checking in and checking out message sets” on page 85.

Now that you have defined a message set, you are ready to define the messages that will belong to it, as described in “Creating messages” on page 59.

Creating messages

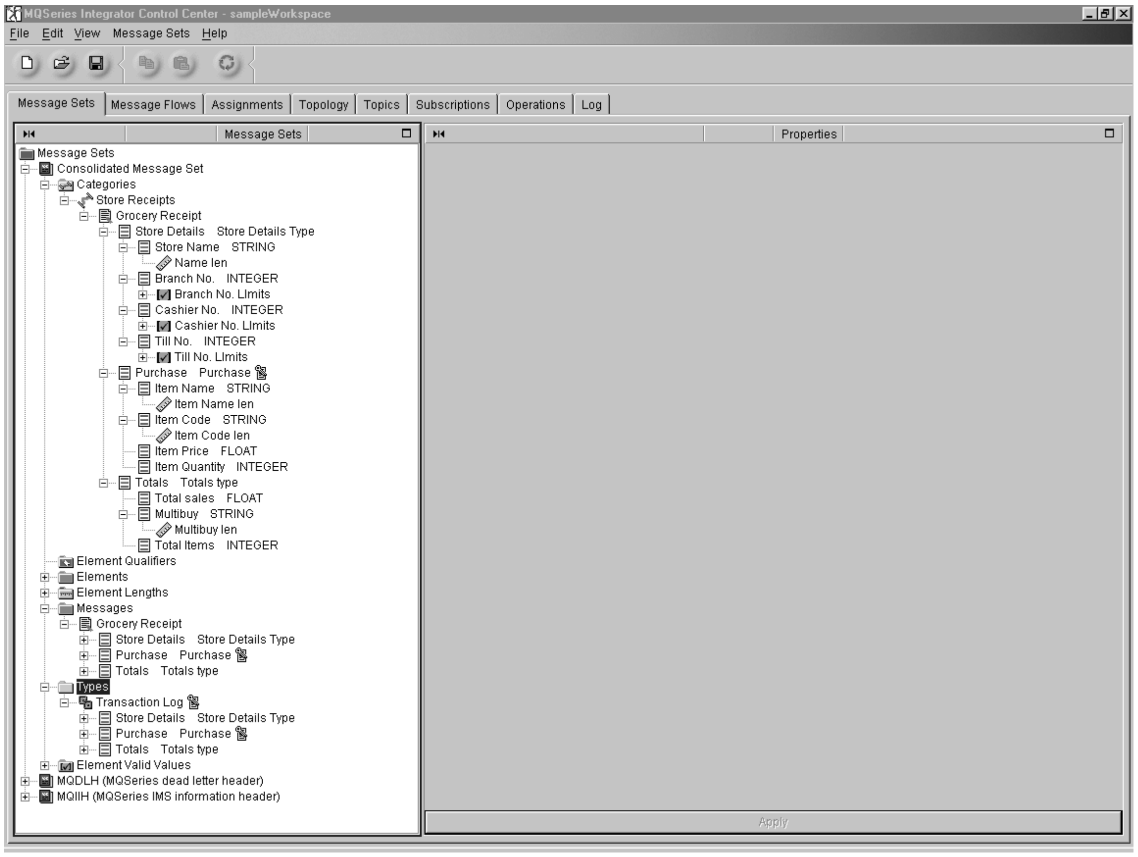
This section describes how to create a message, using the message illustrated in Figure 12 to illustrate the process. The Control Center also provides a SmartGuide for message creation, that handles simpler message formats and makes the task of creating them easier. For details of the SmartGuide, see "Using the SmartGuide to create messages" on page 69.

```
Grocery Receipt      (TransactionLog)
   Store Details
      Store Name     (STRING, Length 20, Fixed Length, Left Justified,
                      Padding character Space)
      Branch No.     (INTEGER, Extended Decimal, Length 8,
                      Unsigned 30000000 - 39999999)
      Cashier No.    (INTEGER, Extended Decimal, Length 3,
                      Unsigned 000 - 500)
      Till No.       (INTEGER, Extended Decimal, Length 8,
                      Unsigned 700 - 799)
   Purchase          (Can have up to 15 purchases on one transaction
                      log)
      Item Name      (STRING, Length 40, Fixed Length, Left Justified,
                      Padding character Space)
      Item Code      (STRING, Length 20, Fixed Length, Left Justified,
                      Padding character Space)
      Item Price     (FLOAT, Packed Decimal, Length=4, Signed, VDP=2)
      Item Quantity  (INTEGER, Packed Decimal, Length=2, Signed)
   Totals
      Total Items    (INTEGER, Packed Decimal, Length=5, Signed)
      Multibuy       (STRING, Length 5, Fixed Length, Left Justified,
                      Padding character Space)
      Total Sales    (FLOAT, Packed Decimal, Length=6, Signed, VDP=2)
```

Figure 12. Creating a message - sample message

You will work with the message set 'Consolidated Message Set'. Within this message set, you will create a message named 'Grocery Receipt', of the type 'Transaction Log'. When you have created the message, you will add it to the category 'Store Receipts'. It is assumed you have already created the message set, and have checked this out of the message repository (that is, the message set is displayed with the **Key** icon against it).

Figure 13 on page 60 shows the **Message Sets** view populated with this message and message set. This is the setup you will create if you complete the following message creation instructions.



```
Message Sets
  Consolidated Message Set
    Categories
      Store Receipts
        Grocery Receipt
          Store Details   Store Details Type
            Store Name   STRING
              Name len
            Branch No.   INTEGER
              Branch No. Limits
            Cashier No.   INTEGER
              Cashier No. Limits
            Till No.   INTEGER
              Till No. Limits
          Purchase   Purchase
            Item Name   STRING
              Item Name len
            Item Code   STRING
              Item Code len
            Item Price   FLOAT
            Item Quantity   INTEGER
          Totals   Totals type
            Total sales   FLOAT
            Multibuy   STRING
              Multibuy len
            Total Items   INTEGER
    Element Qualifiers
    Elements
    Element Lengths
    Messages
      Grocery Receipt
        Store Details   Store Details Type
        Purchase   Purchase
        Totals   Totals type
    Types
      Transaction Log
        Store Details   Store Details Type
        Purchase   Purchase
        Totals   Totals type
    Element Valid Values
  MQDLH (MQSeries dead letter header)
  MQIIH (MQSeries IMS information header)
```

Figure 13. A message defined in the Message Sets view

These instructions demonstrate how to create a message definition from the bottom up (that is, starting with the lowest-level elements and working towards the top of the message hierarchy). All of these tasks are performed in the context of a single message set.

To define this message:

1. Define element length components for all STRING elements.
   a. In the Message Sets pane, right click the entry of the message set Consolidated Message Set and select **Create —> Length**.

      The **Create a new Length** dialog is displayed.
   b. In the **Create a new Length** dialog, type String Length 5 in the **Name** field; type StrLen5 in the **Identifier** field; and type 5 in the **Maximum Length** field.
   c. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.
   d. Click **Finish** to complete the definition of this element length component.

      An entry for this new element length component appears in the Element lengths folder in the message set Receipts.

   Repeat this procedure for the String Length 20 and String Length 40 element length components.

2. Define element valid value components for the Branch No., Cashier No., and Till No. elements.

   a. In the Message Sets pane, right click the entry of the message set and select **Create —> Element Valid Value**.

      The **Create a new Element Valid Value** dialog is displayed.

   b. In the **Create a new Element Valid Value** dialog, type Cashier No. Limits in the **Name** field; type Cashier_VV in the **Identifier** field; select type INTEGER from the **Type** drop-down list; type 000 in the **Minimum Value** field; and type 500 in the **Maximum Value** field.

   c. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.

   d. Click **Finish** to complete the definition of this element valid value component.

      An entry for this new element valid value component appears in the Element valid values folder in the Message Sets pane.

   Repeat this procedure for Branch No. Limits, specifying the identifier Branch_VV, minimum value 30000000, and maximum value 39999999. Repeat this procedure for Till No. Limits, specifying the identifier Till_VV, minimum value 700, and maximum value 799.

3. Create all elements of simple type.

   a. In the Message Sets pane, right click the entry of the message set and select **Create —> Element**.

      The **Create a new Element** dialog is displayed, as shown in Figure 14.

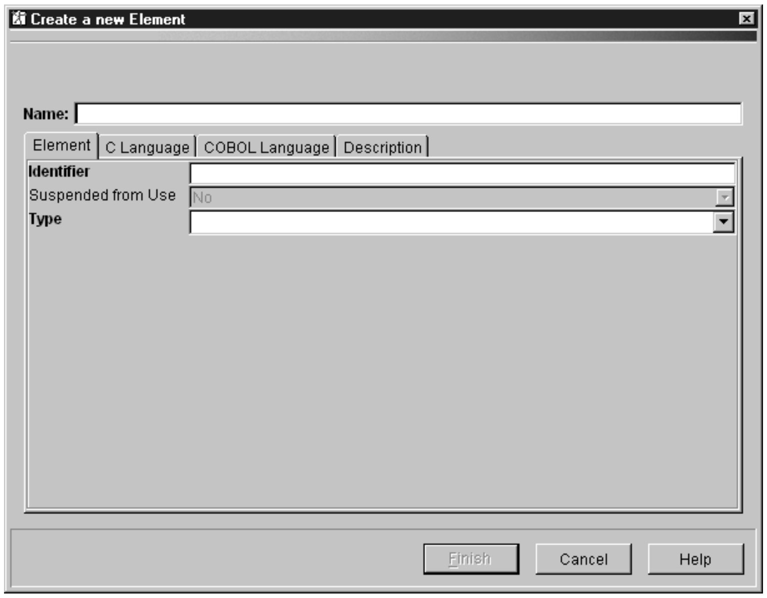


Figure 14. The Create a new Element dialog

   b. In the **Create a new Element** dialog, type Store Name in the **Name** field; type StoreName in the **Identifier** field; and select type STRING from the **Type** drop-down list.

c. Click the **C Language** tab and enter a C name for this element in the C Language Name field. This property is mandatory and is used when you include this element in a C language structure generated from the message repository.

d. Click the **COBOL Language** tab and enter a COBOL name for this element in the COBOL Language Name field. This property is mandatory and is used when you include this element in a COBOL language structure generated from the message repository.

e. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.

f. Click **Finish** to complete the definition of this element component.

An entry for this new element component appears in the Elements folder of the Receipts message set.

Repeat this process for the remaining elements of simple type:

Name	**Identifier**	**Type**
Branch No.	BranchNo	INTEGER
Cashier No.	CashierNo	INTEGER
Till No.	TillNo	INTEGER
Item Name	ItemName	STRING
Item Code	ItemCode	STRING
Item Price	ItemPrice	FLOAT
Item Quantity	ItemQty	INTEGER
Total Items	TotalItems	INTEGER
Multibuy	Multibuy	STRING
Total Sales	TotalSales	FLOAT

4. Add a length reference to elements of type STRING.

| a. In the Message Sets pane, right click the entry for the Store Name element
| in the Elements folder of the message set. Click **Add —> Length**.

The **Add an existing Length** dialog is displayed.

b. From the list of element length components in the **Add an existing Length** dialog, select String Length 20. Click **Finish**.

An entry for the String Length 20 component appears under the Store Name entry in the Elements folder.

Repeat this procedure for the elements Item Name (String Length 40), Item Code (String Length 20), and Multibuy (String Length 5).

5. Add a valid value reference to elements Branch No., Cashier No., and Till No.:

a. In the Message Sets pane, right click the entry for Branch No. in the Elements folder, and click **Add —> Element Valid Value**.

The **Add an existing Element Valid Value** dialog is displayed.

b. From the list of element valid value components in the **Add an existing Element Valid Value** dialog, select Branch No. Limits. Click **Finish**.

An entry for Branch_VV appears beneath the Branch No. entry in the Elements folder.

Repeat this procedure for Cashier No. (Cashier No. Limits) and Till No. (Till No. Limits).

6. Create the compound types Store Details Type, Purchase Type, and Totals Type.
   a. In the Message Sets pane, right click the entry of the message set and click **Create —> Compound Type**.

      The **Create a new Compound Type** dialog is displayed.
   b. In the **Name** field enter Store Details Type, and in the **Identifier** field enter StoreDetsType.
   c. Click the **C Language** tab and enter a C name for this element in the C Language Name field. This property is mandatory and is used when you include this element in a C language structure generated from the message repository. Type the file name in the File Name field.
   d. Click the **COBOL Language** tab and enter a COBOL name for this element in the COBOL Language Name field. This property is mandatory and is used when you include this element in a COBOL language structure generated from the message repository. Type a copy book name in the Structure Copy Book Name field.
   e. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.
   f. Click **Finish** to complete the definition of this compound type component.

      The new compound type appears in the Types folder of the Receipts message set.

   Repeat this process for the compound types Purchase Type and Totals Type.

7. Create the elements Store Details, Purchase, and Totals.
   a. In the Message Sets pane, right click the entry of the message set and select **Create —> Element**.

      The **Create a new Element** dialog is displayed, as shown in Figure 14 on page 61.
   b. In the **Create a new Element** dialog, type Store Details in the **Name** field; type StoreDets in the **Identifier** field; and select type Store Details Type from the **Type** drop-down list.
   c. Click the **C Language** tab and enter a C name for this element in the C Language Name field. This property is mandatory and is used when you include this element in a C language structure generated from the message repository.
   d. Click the **COBOL Language** tab and enter a COBOL name for this element in the COBOL Language Name field. This property is mandatory and is used when you include this element in a COBOL language structure generated from the message repository.
   e. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.
   f. Click **Finish** to complete the definition of this element component.

An entry for this new element component appears in the Elements folder of the Receipts message set.

Repeat this procedure for the elements Purchase and Totals.

8. Add child elements to elements Store Details, Purchase, and Totals.
   a. Right click the element Store Details in the Messages Pane. Click **Add —> Element**.

      The **Add an existing Element** dialog is displayed, showing all elements in your workspace.
   b. Hold down the Ctrl key and select the elements Store Name, Branch No., Cashier No., and Till No. from this list. Click **Finish**.

      The selected elements appear under the entry Store Details in the Elements folder.

   Repeat this procedure to populate the Purchase and Totals elements.
9. To change the order of the child elements in an element, right click the parent element entry in the Messages Pane, and click **Reorder —> Element**. Change the order of the displayed elements, and click **Finish**.

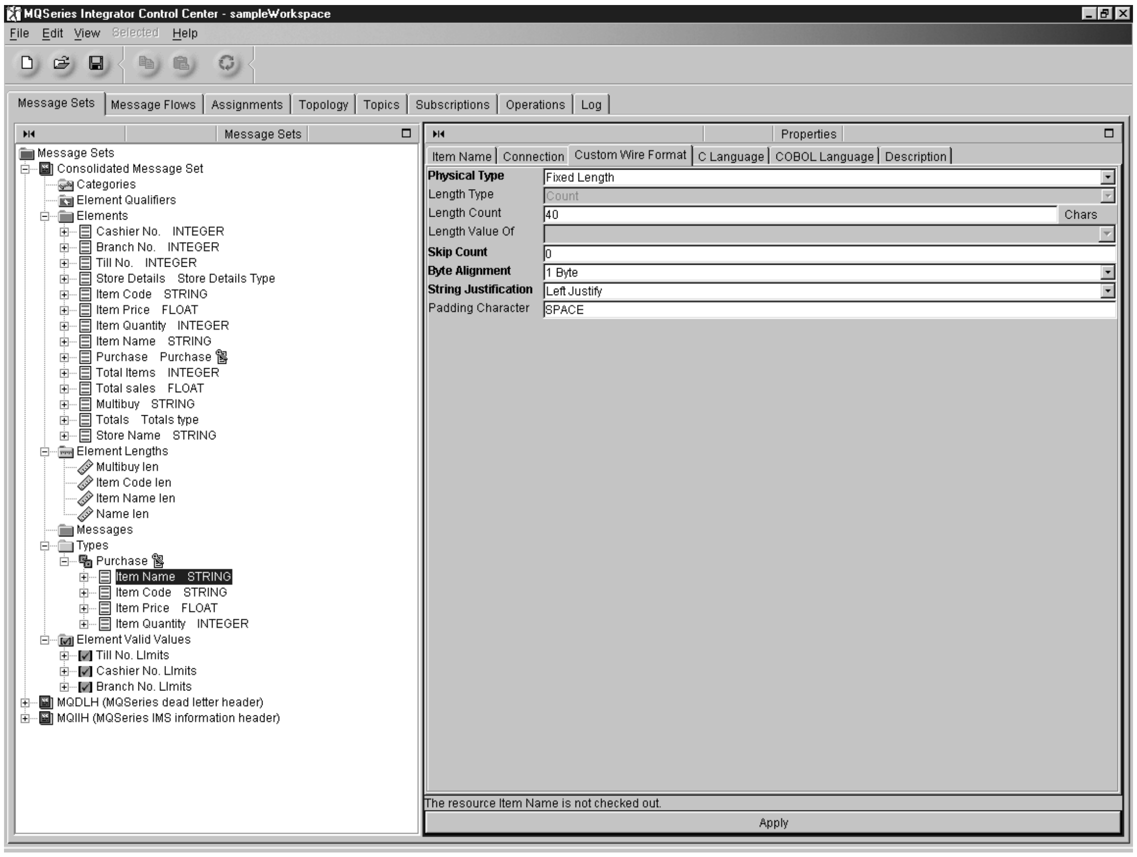
   The reordered elements appear in their new order under the entry for the parent element.
10. Add the CWF characteristics to the child elements in each compound type.
    a. Type Purchase must be checked out.
    b. In the Types folder, expand the Purchase entry and click the child Item Name to select it.
    c. Click the **Custom Wire Format** tab in the Properties pane. The contents of the tab are shown in Figure 15 on page 65

       In the Length field enter 40, and in the Padding Character field type the word Space.

       Click the **Apply** bar at the bottom of the Properties pane.
    d. In the Types folder, expand the Purchase entry and click the child Item Price to select it.
    e. Click the **Custom Wire Format** tab in the Properties pane.

       In the Physical type field, select the type Packed decimal. In the Length field, type 4. In the Signed field, type Yes. In the VDP field, type 2.

       Click the **Apply** bar at the bottom of the Properties pane.


MQSeries Integrator Control Center - sampleWorkspace

File Edit View Selected Help

Message Sets | Message Flows | Assignments | Topology | Topics | Subscriptions | Operations | Log

Message Sets

```
Message Sets
  Consolidated Message Set
    Categories
    Element Qualifiers
    Elements
      Cashier No.   INTEGER
      Branch No.   INTEGER
      Till No.   INTEGER
      Store Details   Store Details Type
      Item Code   STRING
      Item Price   FLOAT
      Item Quantity   INTEGER
      Item Name   STRING
      Purchase   Purchase
      Total Items   INTEGER
      Total sales   FLOAT
      Multibuy   STRING
      Totals   Totals type
      Store Name   STRING
    Element Lengths
      Multibuy len
      Item Code len
      Item Name len
      Name len
    Messages
    Types
      Purchase
        Item Name   STRING
        Item Code   STRING
        Item Price   FLOAT
        Item Quantity   INTEGER
    Element Valid Values
      Till No. Limits
      Cashier No. Limits
      Branch No. Limits
  MQDLH (MQSeries dead letter header)
  MQIIH (MQSeries IMS information header)
```

Properties

Item Name | Connection | Custom Wire Format | C Language | COBOL Language | Description

Physical Type	Fixed Length
Length Type	Count
Length Count	40 Chars
Length Value Of	
Skip Count	0
Byte Alignment	1 Byte
String Justification	Left Justify
Padding Character	SPACE

The resource Item Name is not checked out.

Apply

Figure 15. The Custom Wire Format tab

Follow this procedure for the child elements of the compound types Store Details and Totals, and for the remaining child elements in compound type Purchase.

Note: The CWF characteristics do not belong to an element in isolation. They belong to an element in its context within a type.

Check in any compound types and child elements that are checked out.

11. Create the compound type Transaction Log Type.

    a. In the Message Sets pane, right click the entry of the Receipts message set and click **Create —> Compound Type**.

       The **Create a new Compound Type** dialog is displayed.

    b. In the **Name** field enter Transaction Log Type, and in the **Identifier** field, enter TransLogType.

    c. Click the **C Language** tab and enter a C name for this type in the C Language Name field. This property is mandatory and is used when you include this type in a C language structure generated from the message repository. Type the file name in the File Name field.

    d. Click the **COBOL Language** tab and enter a COBOL name for this type in the COBOL Language Name field. This property is mandatory and is used when you include this type in a COBOL language structure generated from the message repository. Type a copy book name in the Structure Copy Book Name field.

   e. If you want to provide a description of this component, click the **Description** tab. Type a short description, or a long description, or both.

   f. Click **Finish** to complete the definition of this compound type component.

   The new compound type appears in the Types folder of the Receipts message set.

12. Add elements to type Transaction Log.

    a. Right click the entry for the Transaction Log type in the Types folder of the message set Consolidated Message Set. Click **Add —> Element**.

       The **Add an existing Element** dialog is displayed, showing all elements in your workspace.

    b. Hold down the Ctrl key and select the elements Store Details, Purchase, and Totals from this list. Click **Finish**.

    The selected elements appear under the entry Transaction Log in the Types folder of the Consolidated Message Set message set.

13. To change the order of the child elements in a type, right click the Transaction Log type entry in the Types folder of the message set Consolidated Message Set, and click **Reorder —> Element**. Change the order of the displayed elements to that shown in Figure 12 on page 59, and click **Finish**.

    The reordered elements appear in their new order under the entry for the parent element.

14. Add repeat information to child element Purchase in the compound type Transaction Log.

    a. Type Transaction Log must be checked out.

    b. In the Types folder, expand the entry Transaction Log and click the child Purchase to select it.

    c. Click the **Connection** tab in the Properties pane.

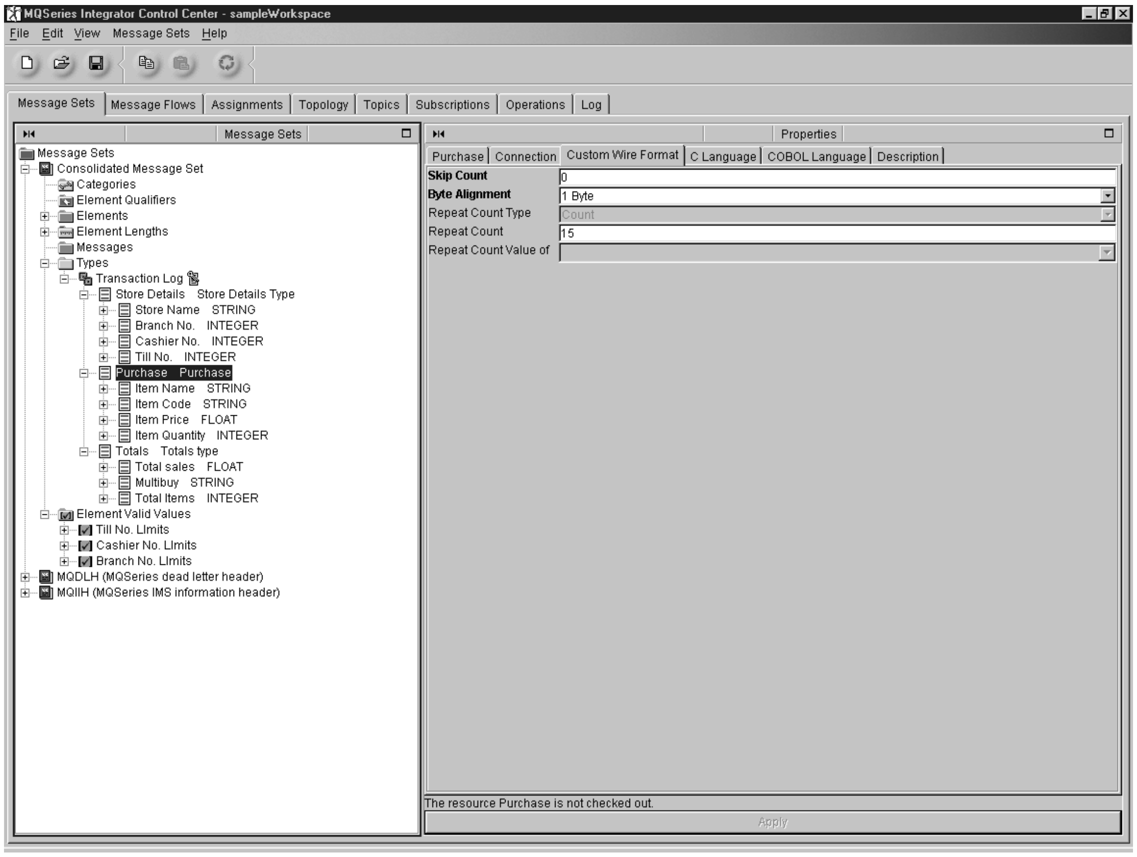
       In the Repeat field, type Yes. Click the **Apply** bar at the bottom of the Properties pane.

    d. Click the **Custom Wire Format** tab in the Properties pane.

       In the Repeat Count field, type 15. This is illustrated in Figure 16 on page 67. Click the **Apply** bar at the bottom of the Properties pane.

    e. Check in Transaction Log.

    Note: The repeat information does not belong to an element in isolation. It belongs to an element in its context within a type.



```
Message Sets
  Consolidated Message Set
    Categories
    Element Qualifiers
    Elements
    Element Lengths
    Messages
    Types
      Transaction Log
        Store Details   Store Details Type
          Store Name   STRING
          Branch No.   INTEGER
          Cashier No.   INTEGER
          Till No.   INTEGER
        Purchase   Purchase
          Item Name   STRING
          Item Code   STRING
          Item Price   FLOAT
          Item Quantity   INTEGER
        Totals   Totals type
          Total sales   FLOAT
          Multibuy   STRING
          Total Items   INTEGER
    Element Valid Values
      Till No. LImits
      Cashier No. LImits
      Branch No. LImits
  MQDLH (MQSeries dead letter header)
  MQIIH (MQSeries IMS information header)
```

Skip Count	0
Byte Alignment	1 Byte
Repeat Count Type	Count
Repeat Count	15
Repeat Count Value of	

The resource Purchase is not checked out.

Figure 16. Setting up repeating fields

15. Create the message component Grocery Receipt.

    a. In the Message Sets pane, right click the entry of the message set and select **Create —> Message**.

       The **Create a new Message** dialog is displayed.

    b. In the **Name** field, enter Grocery Receipt. In the **Identifier** field, enter GroceryReceipt. From the **Type** field drop-down list, select the value Transaction Log Type. Click **Finish**.

    The new message appears in the Messages folder of the Receipts message sets.

16. Create a message category.

    a. In the Message Sets pane, right click the entry of the message set and select **Create —> Category** to define the message category.

       The **Create a new Category** dialog is displayed.

    b. In the **Name** field, enter Store Receipts. In the **Identifier** field, enter StoreReceipts. Click **OK**.

    The new category appears in the Categories folder in the Message Sets pane.

17. Add the message Grocery Receipt to the category Store Receipts.

    a. Right click the category element in the Message Pane. Click **Add —> Message**.

The **Add an existing Message** dialog is displayed, showing all messages in your workspace.

b. Select the message Grocery Receipt. Click **Finish**.

The selected message appears under the entry for category Store Receipts in the Receipts message set.

18. The Multibuy element is optional: it is included only when the customer earns a discount by purchasing a specified multiple of any item. To specify that the element is optional:

    a. Highlight the Multibuy element in the Totals compound element of the Receipts message set so that its properties are displayed in the Properties pane.

    b. Click the **Connection** tab in the Properties pane. Set the Mandatory field to No.

    c. Click the **Apply** bar at the bottom of the Properties pane to apply the change.

Other types of message could be added to this category within this message set. For example, messages describing receipts from clothing stores or from book stores could be added to the category Store Receipts. The messages themselves could be constructed using many of the message components defined for the message Grocery Receipt.

| For a further example of defining messages to the message repository, see
| Appendix A, An example scenario (section “Defining the message in the message
| repository” on page 259).

| When you are ready to share a new message set with other Control Center users,
| you check it into the shared configuration. You must also check in the components
| you have created within the message set, for example, the messages, elements,
| and compound types you have created to complete the message set. For more
| information about checking in message sets, see “Checking in and checking out
| message sets” on page 85.

Using the SmartGuide to create messages

| The MQSeries Integrator Control Center includes a SmartGuide that you can use to
| create messages from the top down. The SmartGuide also allows you to create
| compound types, and lets you specify that the compound type created is itself
| created as a message.

The SmartGuide provides a faster method of defining messages than the process described in "Creating messages" on page 59, not least because it assumes that all the building blocks of the message or compound type are available and do not have to be defined.

| The SmartGuide also allows you to reorder elements within the message or
| compound type you are creating: this makes it easier to view and check the order
| of elements while you complete the message or compound type structure.

The process for defining a message and defining a compound type are almost identical: this process is described below.

- To create a compound type using the SmartGuide:
  1. In the Message Sets pane of the **Message Sets** view, right click the folder of the message set you want to add definitions to and click **Create with SmartGuide —> Compound Type**.

     The **Create a new Compound Type** dialog is displayed, as shown in Figure 17.

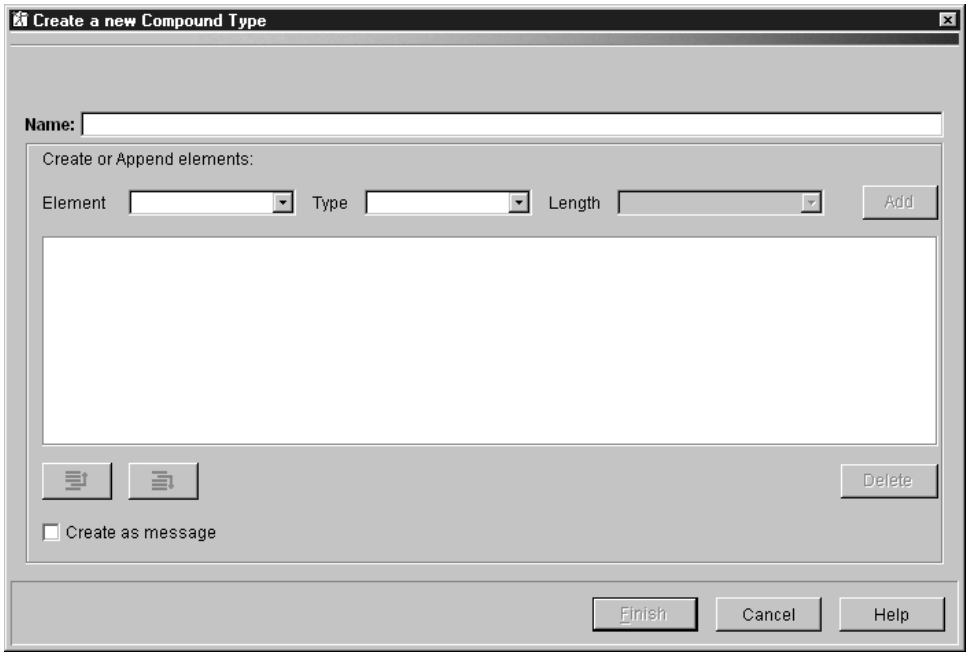


Figure 17. Create a new Compound Type using the SmartGuide

  2. Complete the dialog:
     a. In the **Name** field, enter the name for this new compound type.
     b. From the Element drop-down list, select an element to add to the compound type. If you want to create a new element, type the name of
| the new element in the list box. If you select an existing element from

the list, the Type and Length fields are disabled (because you cannot change these values for a defined element).

c. From the Type drop-down list, select a type for the element.

d. If you select the STRING type, the **Length** field is enabled. You can select a length value from the drop-down list, or you can enter a new numeric value (for example, 25). The value you enter becomes the name as well as the value of the length.

e. Click **Add**.

f. Repeat this selection process for additional elements that you want to add to this compound type. To delete any element you have defined, select the element and click **Delete**.

g. You can reorder the elements you have defined in the compound type by highlighting the element, and using the up and down buttons to move the element to where you want it in the structure.

h. Select the Create as message check box if you want this entire compound type to be a created as a message.

i. Click **Finish**. The compound type you have created is now included in the Types folder of the message set. It has been given an identifier of `t_` followed by the name you gave the type. For example, if the name of the type you created is `ctype1`, the identifier assigned for you is `t_ctype1`.

   Similarly, any elements you have created are given an identifier of `e_` followed by the name of the element (for example, `e_element1`) and the length values you create are given an identifier of `l_` followed by the length value (for example, `_33`).

   Note that any elements and lengths you create within the new compound type are not automatically added to the elements and element lengths folders in the message set, but you can add them if you want to.

The new compound type is added to the Types folder of the message set.

- To create a message using the SmartGuide, use the same procedure as that described above for a Compound Type. Click **Create with SmartGuide —> Message** from the message set actions list, and complete the **Create a new Message** dialog. (The only difference between the two dialogs is that the Create as message check box is not included in the **Create a new Message** dialog.)

Working with message sets

You can use the following actions on message sets or components:

- Copy and Paste
  - Copy and Paste within a message set

    When you copy and paste within a container (folder or a parent component), an exact renamed copy of the component is created. Only the parent component is renamed (and a new uuid is created). If the parent has any child component, these are not renamed: all references to child components (the uuids) remain the same.

    If you copy and paste across containers, a reference to the copied component is pasted into the target component. However, if a reference to the copied component already exists in the target component, a new, renamed copy of the source component is created and a reference to it is added to the target component. The renamed component takes a name that is not already used by another component occurring under the collection folder.

    For example, if the Elements folder includes e1, copy and paste e1 from compound type T1 to itself creates a new element e1_1. If the Elements folder has e1 and e1_1 under it, copy and paste e1 from T1 to T2 where T2 has no children results in e1 as a reference. If you paste e1 again under T2, then a new e1_2 is created.
  - Copy and paste between two message sets

    If you copy and paste a parent component from one message set to another, and the identifier of the parent already exists within the target message set, you are presented with a message dialog that indicates that the component already exists and the paste is not allowed.

    Child components or references of the copied parent component are also copied across if they do not already exist within the target message set. If they do already exist, references to these existing components are used. However, this could result in a copied component differing from the source component (because child components or references have the same identifier but are not structurally the same). If this happens, you are presented with a message dialog warning you that existing referenced components already exist in the target message set will be used and not overwritten.

    There is no link between the copied components in the target message set and the source message set components. All copied components appear as if newly created in the target message set.
  - Copy and Paste of complete message sets

    This function is not available through the Control Center: you must use the command **mqsimrmcopymsgset** to perform this task. This command creates a second message set with a unique name that is identical in every way to the first message set. When the copy has completed, you must restart both the Control Center and the Configuration Manager. The new message set is then available in the message repository and you can add it to your workspace for update and assignment.

    For further details of this command, see the *MQSeries Integrator Administration Guide*.

- Reorder

  The reorder action is only supported for compound types. Right click the compound type you want to reorder, and select **Reorder->Element**. The *Reorder Elements* SmartGuide is displayed listing the elements defined within the compound type. You can move these elements up or down to change the order as you choose. You cannot add new elements using this option. You cannot reorder external elements (those with the global icon).

  Press **Finish** to confirm you new order: the new order is reflected in the tree in the left hand pane.

  You can use this action if you want to reorder elements to change the rules that apply when the CWF attributes are set.

- Undo

  The undo action for message sets is consistent with undo across the Control Center, but you are unable to undo the following actions:

  – Deleting a message set
  – Deleting a compound type
  – Deleting an element

Adding message sets and message components to the workspace

You can add any message set or component that is defined in the message repository to your workspace. The message repository includes all objects defined by current users of the Control Center, and all objects imported into the message repository through the use of the **mqsimrmimpexp** command. To add an existing message set to the workspace:

1. In the Message Sets pane, right click the **Message Sets** root.
2. Click **Add to Workspace—> Message Set**.

   The **Add an existing Message Set** dialog is displayed, showing all message sets that you can add to your workspace (and that aren't already in the workspace).
3. Select message sets from this list as follows:
   - To select a single message set, click the message set name.
   - To select multiple message sets that appear sequentially in the list, click on the first message set you want, press and hold the Shift key, then click on the last message set you want. This action selects the two message sets you highlighted, plus any that appear between these two in the list.
   - To select multiple message sets that do not appear in a sequence in the list, hold down Ctrl and click each message set you want.
4. When you have selected the message sets you want, click **Finish**.

You now see the selected message sets in the Message Sets view. All of the components of the message set (messages, elements, and so on) are now available to your workspace, but are not automatically added. This is because message sets can be very complex, and it is likely that you do not need to view or access many of the subcomponents. If you add large numbers of components to the workspace, this can cause slow response times and out-of-memory problems.

You can add just those components that you want to work with, or view, by selecting the appropriate folder and adding the components when and as you need them.

For example, to add an element to your workspace:

1. Right click the Element folder of the message set to which you want to make the element available, and click **Add to Workspace —> Element**.

   The **Add an existing Element** dialog is displayed, showing all elements that you can add, that is, all the elements that are defined in the message set.
2. Select one or more elements from the list:
   - To select a single element, click the element name.
   - To select multiple elements that appear sequentially in the list, click on the first element you want, press and hold the Shift key, then click on the last element you want. This action selects the two message sets you highlighted, plus any that appear between these two in the list.
   - To select multiple elements that do not appear in a sequence in the list, hold down Ctrl and click each element set you want.

   The selected elements are added to the Elements folder of the appropriate message set in the Message Sets pane.

You can add categories, element qualifiers, element lengths, messages, types, and element valid values to your workspace in the same way.

Importing message definitions

Legacy definitions can be imported into the message repository, as described in "Importing legacy formats" on page 54.

To import a message definition:

1. In the Message Sets pane of the **Message Sets** view, right click the message set into which you want to import the definition and click **Import to Message Set —> C** or **Import to Message Set —> COBOL**. The action is also available if you select **Message Sets** from the task bar and click **Import to Message Set —> C** or **Import to Message Set —> COBOL**.

   The **C Language Importer** dialog or **COBOL Language Importer** dialog is displayed.
2. Type the fully qualified name of the source file you are importing in the Import Source File field, or use the **Browse** button to search for and select the file you want to import. If you want only to generate a report at this time, select the Report only check box. Click **OK**.

This process imports the specified structures and creates definitions as a new message set in the message repository. To complete the process, you must create a message component for each of the compound types that define a complete message, as described in step 15 on page 67. You do not need to create any other message component.

Note: You cannot import message sets created by another Control Center user into your Control Center session. This function is only supported by the message set import and export command (**mqsimrmimpexp**), which is described in the *MQSeries Integrator Administration Guide*.

Generating MRM message set definitions in XML DTDs

If you have defined messages with an XML message format in the message repository, you can request a Document Type Descriptor (DTD) to be generated by the MRM.

To generate a DTD:

1. In the Message Sets pane of the **Message Sets** view, right click the folder of the message set for which you want to generate the DTD. Click **Generate —> DTD**.

   The **Generate DTD** dialog is displayed.

2. In the **Generate DTD** dialog, enter the name of the DTD file in the DTD Filename field. Click **Start**.

The DTD for this message set is generated as requested and written to the specified location.

Generating language bindings

You can generate C or COBOL language bindings from message definitions you have created using the Control Center:

To generate C language bindings:

1. In the Message Sets pane of the **Message Sets** view, right click the folder of the message set for which you want to generate language bindings. Click **Generate —> Language Bindings —> C**.

   The **C Language Extractor** dialog is displayed.

2. In the **C Language Extractor** dialog, enter the fully qualified name of the directory of the generated file in the Generated File Location field. If you want to freeze the message set at this time, select the Freeze Message Set check box. (If the message set is already finalized, you cannot select this check box.)

   The categories defined in this message set are listed in the Categories field. You can select a subset of these for inclusion in the language bindings. Alternatively, to include them all, select the Select All check box.

   You must select at least one category for successful generation of language bindings. If no category is listed in this dialog, you must create one.

3. Click **Start**.

   The requested language bindings are generated and written to the specified location.

The process for generating COBOL bindings is identical.

Generating documentation

You can generate the following documentation in HTML format from the message repository:

- A message book, which contains an entry for each message in a message set or specified category, showing its hierarchical structure.
- A glossary, which contains descriptions of all elements in a message set or specified category, ordered alphabetically by name.

To generate a message book:

1. In the Message Sets pane of the **Message Sets** view, right click the folder of the message set or message category for which you want to generate documentation.
2. Click **Generate —> Documentation —> Message Book**.

   The **Message Definition Book** dialog is displayed.
3. In the **Message Definition Book** dialog, type the fully qualified name of the generated documentation file in the Generated File Location field. This file must be created on the system on which the Configuration Manager is running, not on the local system.

   If you want to freeze the message set at this time, select the Freeze Message Set check box. (If the message set is already finalized, you cannot select this check box.)

   You can generate documentation based on categories or messages. The categories or messages (depending on which you select) defined in this message set are listed in the Categories or Messages field. You can select a subset of these for inclusion in the language bindings. Alternatively, to include them all, select the Select All check box.

   You must select at least one category or one message for successful generation of documentation. If there is no category listed, you must create one.
4. Click **Start**.

The Message Book is generated and written to the specified location: the file MRM-MAIN.HTML is created, along with a subdirectory named `Private`. This subdirectory contains a large number of files, for example, image files and indexes.

The description of each element within each compound type included in the Message Book indicates whether the element is mandatory or optional. There are four possible settings:

1. The element can be `Optional`.

   This value is set if the element is optional within its type.
2. The element can be `Mandatory if parent present`.

   This value is set if the element is mandatory within its type.
3. The element can be `Always Mandatory`.

   This is set if the element is associated with a context tag and that context tag is mandatory within the message for which the documentation has been created. This setting overrides the values `Optional` and `Mandatory if parent present`.

If the element is associated with a context tag and that context tag is optional within the message, the current setting of `Optional` or `Mandatory if parent present` is not overridden.

4. The element can be `Implied Mandatory`.

   This value is set if one of its descendants has been given the value `Always Mandatory`. This value will override the value set for the element in isolation unless the element itself has the `Always Mandatory` value.

Repeating elements are indicated by the characters `***` after the element name.

To generate a glossary:

1. In the Message Sets pane of the **Message Sets** view, right click the folder of the message set or message category for which you want to generate documentation.
2. Click **Generate —> Documentation —> Glossary** from the action list of the message set.

   The **Glossary** dialog is displayed.

   This dialog is identical to the **Message Definition Book** dialog, except that only categories are available.
3. Complete the dialog and click **Start**.

The Glossary is generated and written to the specified location: the file MAINGLOS.HTML is created, along with a subdirectory named `Private`. This subdirectory contains a large number of files, for example, html files.

Editing message sets and components

You can edit the properties of message sets and components. You can also edit the relationships between components (for example, you can remove an element from a compound element), and you can delete components or remove them from the workspace.

All properties you can edit are displayed in the Properties pane of the **Message Sets** view. For example, if you highlight an element in the Message Sets pane, its properties, including those you can edit, are displayed in the Properties pane. When you change the value of a property, you click the **Apply** bar at the bottom of the Properties pane to make the change take effect.

An individual message component can be removed from the workspace or deleted from the shared configuration. For example, to remove an element from the workspace, right click the element in the Messages Pane and click **Remove**. Note, however, that whether a component is checked out dictates whether you can edit its properties, remove it from the workspace, or delete it, as does the check-out status of any *related* component. Table 1 summarizes the available edit actions and shows for each action:

- Which component needs to be checked out
- What happens when you make the change

Table 1 (Page 1 of 4). Editing relationships and properties: check-out requirements

If you want to:	You must check out:	Then:
Edit the basic properties of a component	The component you want to edit	You can edit the component and check it back in
Edit the connection tab of a child element	The compound type that is the parent of the element	You can edit the connection tab then check the parent back in.
Edit the CWF of a child element	The compound type that is the parent of the element	You can edit the CWF tab then check the parent back in.
Edit the C language tab, COBOL language tab, or Description tab of a component	The component you want to edit	You can edit all three tabs. The name is C-validated or COBOL validated by the Control Center; you cannot click **Apply** if they are invalid. If they are valid, you can check the component back in.
Edit an element qualifier assignment	The associated message	You can edit the message then check it back in.
Delete an element length from the Element Lengths folder	Nothing	If nobody has the element length checked out, and if no string element depends on the element length, the element length is deleted from the shared repository. Otherwise, you get an error message and are not allowed to delete the element length.
Remove an element length from the Element Lengths folder.	Check-out status is not significant	The element length is removed from the workspace and there is no change in the shared repository. The element length can be added to the workspace again.

Table 1 (Page 2 of 4). Editing relationships and properties: check-out requirements

If you want to:	You must check out:	Then:
Delete a compound type from the Types folder	Nothing	If any user has an element or a message of this type checked out, or if any user has this type checked out. you get an error message and are not allowed to delete. Otherwise, the type is deleted and all elements of this type are also deleted throughout the message set.
Remove a compound type from the Types folder	Check-out status is not significant	The type and its children are removed from the workspace under the Types folder. Nothing else is affected. The compound type can be added to the workspace again.
Remove a simple type from the Types folder	Nothing	The type is removed from the workspace under the Types folder. Nothing else is affected. The simple type can be added to the workspace again.
Remove a child simple element from a type in the Types folder	The type from which you will remove the element	The child is deleted from the type. When you check the type in, it is updated in the shared repository, but the child element continues to exist. Other types that contain the element as a child are not affected.
Delete a child simple element from a type in the Types folder	Nothing	If nobody has the child simple element checked out and if nobody has any type or element qualifier that is a parent of the simple element checked out, the element is deleted from the shared repository and all the types that previously used it as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
Delete a child compound element from a type in the Types folder	Nothing	If nobody has the child compound element checked out and if nobody has any type or element qualifier that is a parent of the compound element checked out, the element is deleted from the shared repository and all the types that previously used it as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
Remove a child compound element from a type in the Types folder	The type from which you will remove the element	The child is deleted from the type and, on check in, the type is updated in the shared repository but the child element continues to exist. Other types that contain the element as a child are unaffected.
Delete a simple element from the Elements folder	Nothing	If nobody has the element checked out, and if nobody has any type or element qualifier that is a parent of the element checked out, the element is deleted from the shared repository and all the types that previously used it as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
Remove a simple element from the Elements folder	Check-out status is not significant	The element is removed from the workspace under the Elements folder. Nothing else is affected. The element can be added to the workspace again.
Delete a top-level compound element in the Elements folder	Nothing	If nobody has the element checked out, and if nobody has any type or element qualifier that is a parent of the element checked out, the element is deleted from the shared repository, and all the types that used the element as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.

Table 1 (Page 3 of 4). Editing relationships and properties: check-out requirements

If you want to:	You must check out:	Then:
Remove a top-level compound element in the Elements folder	Check-out status is not significant.	The element and its children are removed from the workspace under the Elements folder. Nothing else is affected. The compound element can be added to the workspace again.
In the Elements folder, alter a compound element by deleting a child simple element	Nothing	If nobody has the child element checked out; and if nobody has any type or element qualifier that is a parent of the element checked out; and if nobody has the type of the compound element checked out; the element is deleted from the shared repository, and all the types that previously used the element as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
In the Elements folder, alter a compound element by removing a child simple element	The type associated with the compound element	The child is deleted from the type, and when you check the type back in, it is updated in the shared repository but the child element continues to exist. Other types that contain the element as a child are not affected.
In the Elements folder, alter a compound element by deleting a child compound element	Nothing	If nobody has the child element checked out; and if nobody has any type or element qualifier, of which the compound element is a child, checked out; and if nobody has the type of the parent compound element checked out; then the element is deleted from the shared repository and all the types that used the element as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
In the Elements folder, alter a compound element by removing a child compound element	The type associated with the parent compound element	The child compound element is removed from the workspace. Nothing else is affected. The compound element can be added to the workspace again.
Delete a message in the Messages folder	Nothing	If nobody has the message checked out, and if nobody has any category of which the message is a child checked out, the message is deleted from the shared repository, Otherwise, an error message is issued and you are not allowed to perform the delete.
Remove a message from the Messages folder	Check-out status is not significant	The message and its children are removed from the workspace under the Messages folder. Nothing else is affected. The message can be added to the workspace again.
Alter a message by deleting a simple child element in the Messages folder	Nothing	If nobody has the child element checked out, and if nobody has the type of the message checked out, and if nobody has any type, of which the element is a child, checked out, then the element is deleted from the shared repository and all the types that used the element as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
Alter a message by removing a child simple element in the Messages folder	The type associated with the message that contains the element	The child is deleted from the type, and on check in the type is updated in the shared repository, but the child element continues to exist and other types that contain the element as a child are not affected.

Table 1 (Page 4 of 4). Editing relationships and properties: check-out requirements

If you want to:	**You must check out:**	**Then:**
Alter a message by deleting a child compound element in the Messages folder.	Nothing	If nobody has the child element checked out; and if nobody has any type or element qualifier that is a parent of the child compound element checked out; and if nobody has the type of the message checked out; then the element is deleted from the shared repository and all the types that used the element as a child are updated. Otherwise, an error message is issued and you are not allowed to perform the delete.
Alter a message by removing a child compound element in the Messages folder	The type associated with the message that contains the element	The child compound element is removed from the workspace. Nothing else is affected. The child compound element can be added to the workspace again.

Changing the state of a message set

When a message state is created, its state is normal. During their development, message sets can be frozen, unfrozen, and finalized, as described in “Message set states” on page 53.

To change the state of a message set:

1. In the Message Sets pane, right click the message set whose state you want to change.
2. Click the state you want. For example, to freeze a message set, click **Freeze**.

Note the following:

- When you freeze a message set, the freeze timestamp is added to the properties of the message set.
- If you unfreeze a message set, the freeze timestamp in the properties of the message set is reset to blank.
- When you finalize a message set, the Finalized field in the properties of the message set is set to True and the freeze timestamp is set. Finalize cannot be reversed. For more information, see “Message set states” on page 53.
- You cannot freeze or finalize a message set if any of the elements it contains is checked out.

Checking in and checking out message sets

When you have created and populated a message set, you can assign it to a broker (as described in “Assigning message sets to brokers” on page 205). You do not need to have checked the message set into the shared configuration before assigning it. However, you must check it in before the assignment of message set to broker can be deployed in the broker domain.

To check in a message set, in the Message Sets pane right click the folder of the message set you want to check in, and click **Check In**.

The message set is checked into the shared configuration. It still appears in your workspace, but the **Key** icon against its folder has disappeared.

| When you check in a message set, any checked out objects in the message set
| are not checked in by this action. You must check in these objects individually.
| Alternatively, you can select one of the more comprehensive check in options
| (available from the File menu) when you check in the message set:

- **File —> Check In —> All in Current Workspace** checks in all objects that are contained within your current workspace (this is identified on the title bar of the Control Center).
- **File —> Check In —> All (Save to Shared)** checks in all objects in your local repository (that is, within all of your available workspaces).

Once you have checked in a message set, it is available to other users from the shared configuration. If you want to make further changes to the message set, you must first check it out of the shared configuration:

1. In the Message Sets pane, right click the folder of the message set you want to edit.
2. Click **Check Out**.

   The message set is checked out of the shared configuration. Its entry in the Messages Pane has a **Key** icon against it to remind you that the definition is checked out.

Chapter 4. Defining message flows

This chapter describes the tasks you need to perform to create message flows. These are:

- "Creating a message flow" on page 89
- "Creating a message flow category" on page 93
- "Adding a message flow to your workspace" on page 94
- "Checking a message flow" on page 95
- "Including one message flow in another" on page 97
- "Promoting message flow node properties" on page 98
- "Checking in message flows" on page 102
- "Creating your own message nodes" on page 103
- "The IBM Primitives" on page 104
- "Using the IBM-supplied message flows" on page 175

Appendix A, "An example scenario" on page 257 provides an example that shows how to construct a message flow.

Authorization to work with message flows

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Message flow and message set developer**
  - **All roles**

  For information about setting your user role, see "Setting user roles" on page 13.
- Be a member of the MQSeries Integrator group **mqbrdevt**

The Message Flows view

To display the **Message Flows** view, click the **Message Flows** tab in the Control Center. Figure 18 on page 88 shows an example of the **Message Flows** view.

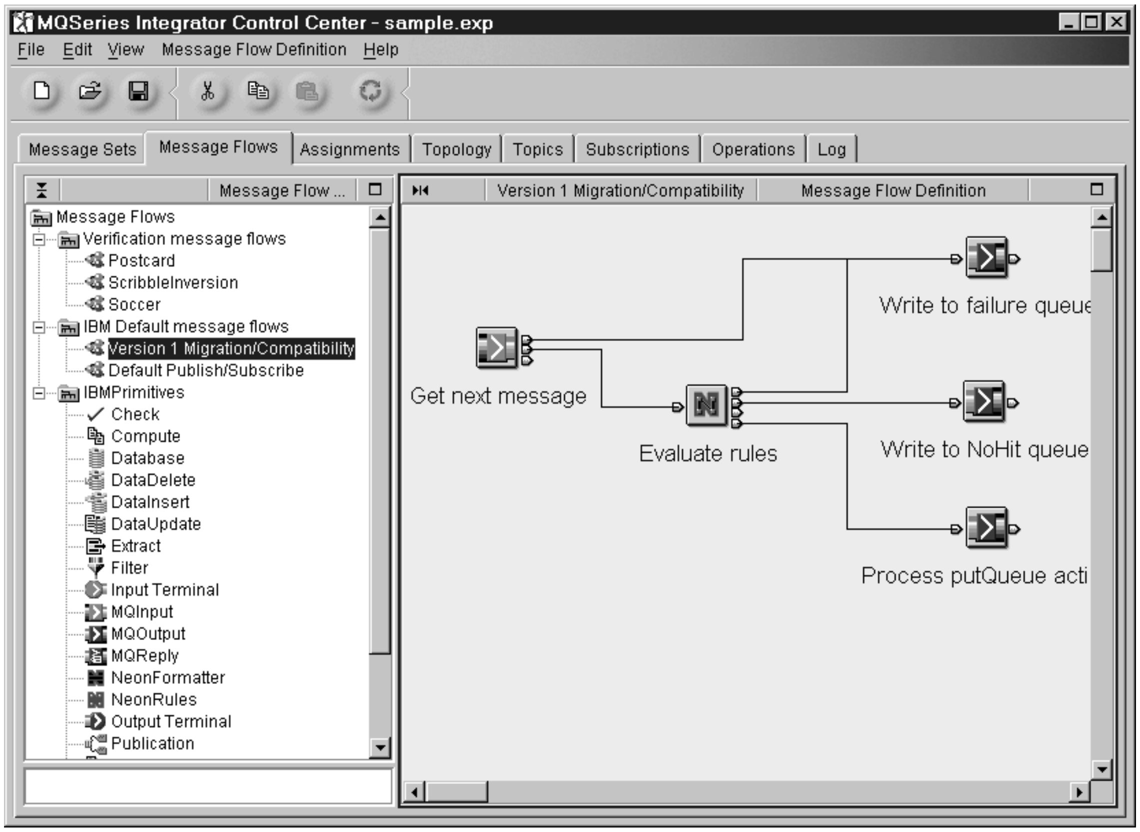


Figure 18. The Message Flows view. The left-hand pane, the Message Flow Types pane, shows a tree view of the message flows in your workspace. The right-hand pane, the Message Flow Definition pane, contains an arrangement of graphical symbols that represent the message flow nodes in a selected message flow.

Controlling the appearance of the Message Flow Definition pane

When you add an instance of a message flow node into a message flow by dragging it into the Message Flow Definition pane, the graphical symbol representing the node is created. You can control the appearance and arrangement of these symbols by right-clicking in the Message Flow Definition pane and selecting from the following actions:

Layout graph	Arranges the nodes in the Message Flow Definition pane from left to right, right to left, top to bottom, or bottom to top.
Zoom	Alters the size of all node symbols in the Message Flow Definition pane.
Manhattan style	Shows connections between nodes as lines at right angles (as shown in Figure 18).
Snap to grid	Aligns the symbols in the Message Flow Definition pane on an invisible grid.

Creating a message flow

1. In the Message Flow Types pane of the **Message Flows** view, right click the Message Flows root, and click **Create —> Message Flow**.

   The **Create a new Message Flow** dialog is displayed.

2. In the **Name** field, type the name of your new message flow. This must follow the naming rules described in "Naming Control Center resources" on page 20. Click **Finish**.

   Confirmation that the message flow has been created appears in two places in the **Message Flows** view:

   - The name of the new message flow appears in the title bar of the Message Flow Definition pane.
   - An entry for the new message flow appears in the Message Flow Types pane with a **New** icon against it.

   If you copy and paste a message flow, this action creates a new reference to the original message flow, but does not create a new object. Therefore any updates you make to the copy (including rename and delete) are automatically made to the original message flow. If you want a new message flow based on an existing message flow, but independent of it, you must use duplicate.

   You are now ready to assemble the message flow from the available message flow nodes.

3. In the Message Flow Types pane, drag each of the message flow nodes you want to use into the Message Flow Definition pane. (This step fails if you have not defined a message flow into which you can drag the message flow nodes.)

   A graphical symbol representing each of the nodes you select is shown in the Message Flow Definition pane. The first node of each type that you select has the number "1" appended to its name. For example, if you construct a simple message flow using the MQInput, DataUpdate, and MQOutput message nodes, each appears in the Message Flow Definition pane as shown in Figure 19 on page 90.

   If you use more than one instance of any of these nodes within a single message flow, and do not rename each node immediately (as described below), the number appended is incremented each time (the second instance has "2", the third has "3", and so on).

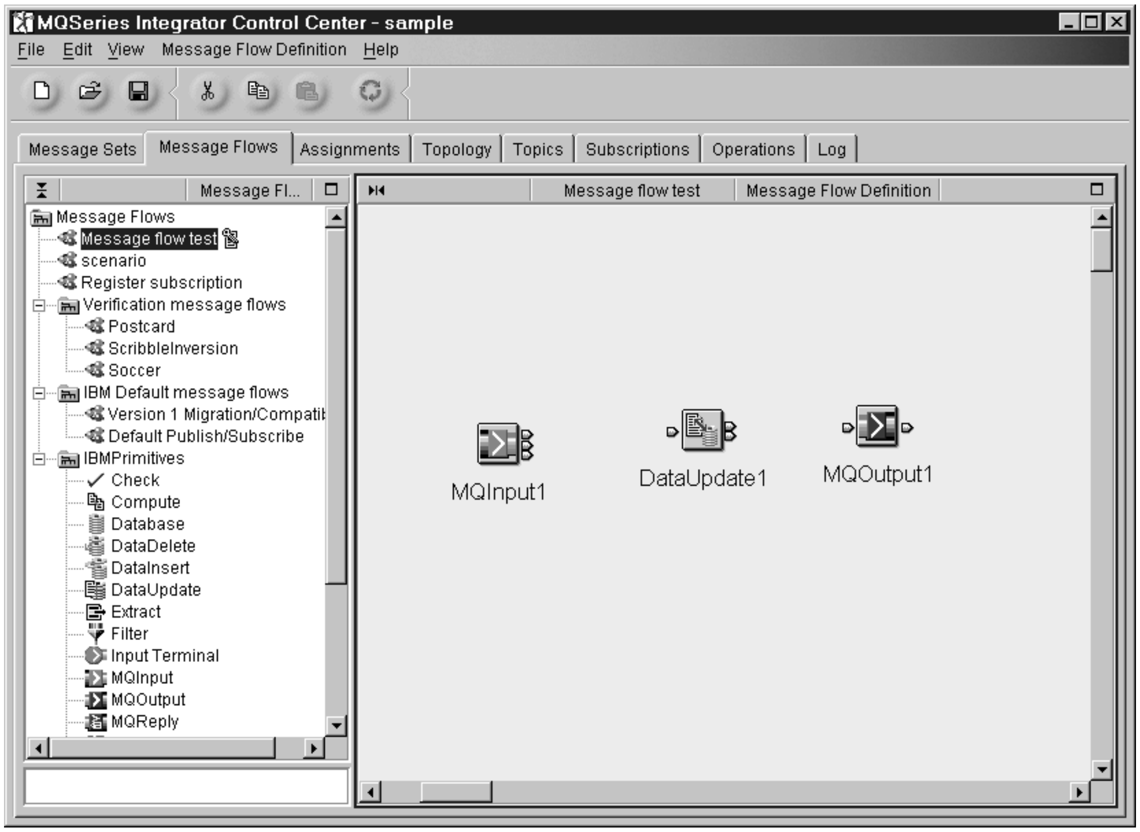


Figure 19. Dragging message flow nodes into the Message Flow Definition pane. These message flow nodes are instances of the IBM Primitives MQInput, DataUpdate, and MQOutput. The "1" appended to their names ensures unique naming.

4. If you want these nodes to have different names from those assigned, you can rename them by following this procedure:

   a. In the Message Flow Definition pane, right click on one of the message flow node symbols, and click **Rename**.

      The **Rename MessageProcessingNode** dialog is displayed.

   b. In the New Name field, type the new name of this instance of the message node. Click **Finish**.

   The new name of the message flow node appears beneath its symbol in the Message Flow Definition pane. Repeat this process for other message nodes you want to rename.

   Now you are ready to connect the message nodes in your message flow in a way that will provide the processing logic you require. For the remainder of this section, let's assume that you are connecting the MQInput message flow node to the DataUpdate message flow node.

5. To connect the out terminal of MQInput to the in terminal of DataUpdate, right click the MQInput symbol in the Message Flow Definition pane, and click **Connect —> Out**. (All terminals available to this node appear in this list.) The cursor becomes a cross-hair attached by a red line to the out terminal.

6. Move the cross-hair to the in terminal shown on the symbol of the DataUpdate node, and click. A line now connects the out terminal of the MQInput node to the in terminal of the DataUpdate node.[4]

   Follow this process for all terminals within the message flow between which you want to establish connections.

   Figure 20 shows a simple message flow with connections between message flow nodes.

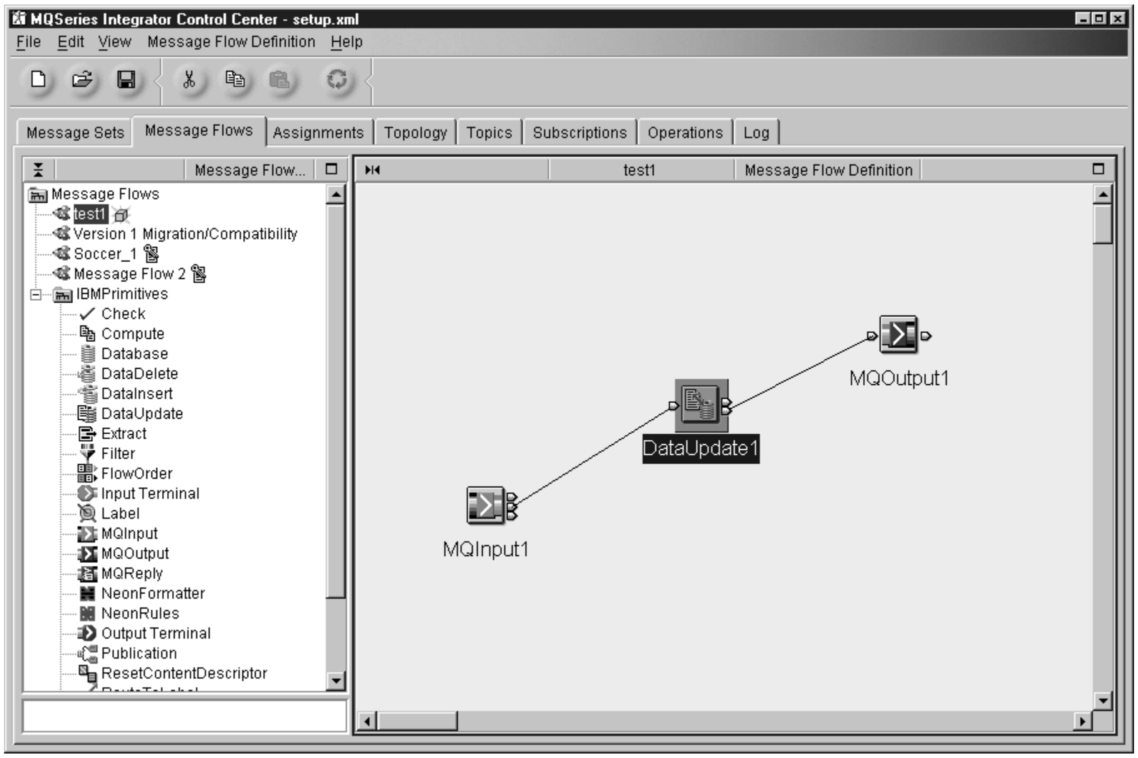


Figure 20. A message flow showing connections between terminals. In this example, the out terminal of MQInput1 has been connected to the in terminal of DataUpdate1, and the out terminal of DataUpdate has been connected to the in terminal of MQOutput1.

7. You must configure the nodes in your message flow to match your processing requirements. Configuration instructions for each IBM Primitive node are provided in order of message-node name, beginning with "Check node" on page 106. Note that, once you have assembled the message nodes you want to use in the Message Flow Definition pane, the order in which you rename, connect, and configure them is unimportant.

8. If you are ready to make this message flow generally available within the broker domain, check it into the shared configuration as described in "Checking in message flows" on page 102.

[4] An alternative way of connecting terminals is to move the cursor slowly over the terminal icons of the node until the label of the terminal you want to connect is displayed, then press the right mouse button. This action converts the cursor to a cross-hair attached by a red line to the node, which you can move to the appropriate terminal of the next node. Release the mouse button to connect the line. This method requires a certain dexterity.

You must be aware of the following information regarding those message flows that access external databases:

1. When a database is accessed from a message flow, data is converted from the local code page of the broker process to Unicode (as used internally by the broker), and vice versa. ESQL statements are converted from Unicode to the local code page of the process prior to execution. The data in a result set produced by an SQL SELECT is converted into Unicode from the code page of the process. If the database being accessed is configured with a different code page from that of the broker process, a data conversion is performed by the ODBC driver or the database, based on the conversions supported by that database.
2. Fully globally coordinated message flows that involve a DB2 resource manager are supported on DB2 Universal Database V6.1 only.
3. The message flow thread connects to the specified data source, unless it is already connected. Once a thread has acquired a connection to an ODBC data source, the connection is not relinquished.

   You are recommended to determine the number of database connections required by a broker for capacity and resource planning purposes. The default action taken by DB2 is to limit the number of concurrent connections to a database to the value of the *maxappls* configuration parameter. The default for *maxappls* is 40. Check the appropriate documentation for connections to databases from other suppliers.

   The connection requirements for a single message flow are:

   - One required per message flow thread that contains a publication node.
   - One required per database access node to separate ODBC data source names per message flow thread (that is, if the same DSN is used by a different node, the same connection is used).

Note: These database connections are in addition to the run-time connections required by the broker (to the DB2 or SQL Server database that is defined to hold its internal information). For details of these connections, refer to the MQSeries SupportPac MHI1 (see “MQSeries information available on the Internet” on page xii for access to this and other MQSeries SupportPacs).

Creating a message flow category

When you have a large number of message flows in your workspace, the Message Flows tree in the Message Flow Types pane can become difficult to navigate. To introduce some structure into the list, you can define message flow categories, under which you can organize related message flows. (The IBM Primitives, for example, belong to the IBMPrimitives message flow category.)

To create a message flow category:

1. In the Message Flow Types pane, right click the root of the Message Flows tree, and click **Create —> Message Flow Category**.

   The **Create a new Message Flow Category** dialog is displayed.

2. In the **Name** field, type the name of your message flow category. Click **Finish**.

An entry for the new message flow category appears in the Message Flow Types pane.

You can create new message flows within this new message flow category, as follows:

1. Right click on the message flow category folder in the Message Flow Types pane, and click **Create —> Message Flow**.
2. Follow the instructions for creating a message flow from step 2 on page 89.

You can also add existing message flows to a message flow category, as described in “Adding a message flow to your workspace” on page 94.

Adding a message flow to your workspace

If you want to incorporate message flows created by other Control Center users in your own message flows, you need to begin by adding them to your workspace. When you add definitions to your workspace, a reference to each definition is created in your workspace.

To add a message flow to your workspace:

1. Right click the Message Flows root in the Message Flow Types pane, and click **Add to Workspace —> Message Flow**.

   The **Add an existing Message Flow** dialog is displayed, as shown in Figure 21.

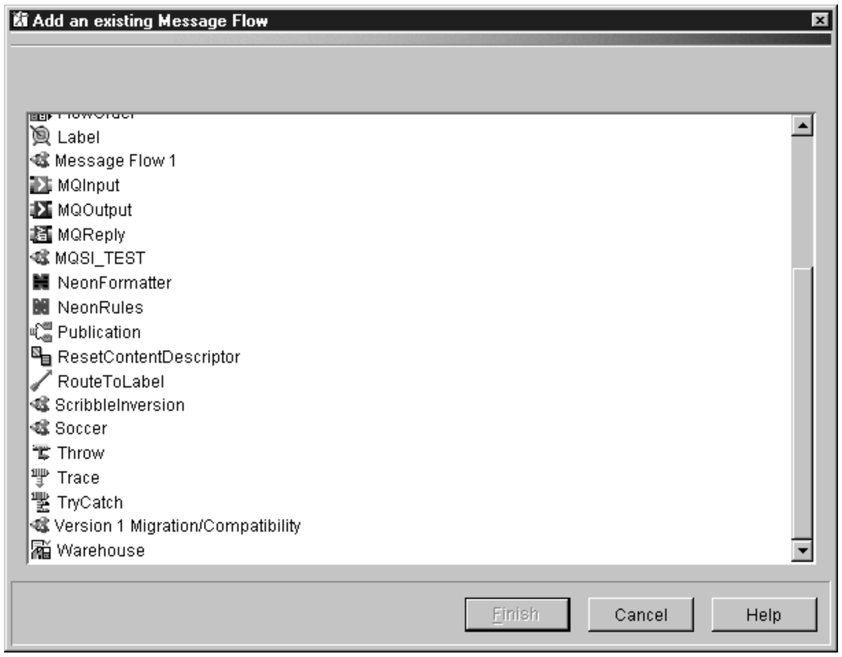


Figure 21. Add an existing Message Flow dialog. The dialog displays all message flows that have been checked into the shared configuration in this broker domain.

   - To select a single entry from this list, click the message flow name.
   - To select multiple entries that appear sequentially in the list, click on the first message flow you want, press and hold the Shift key, then click on the last one you want. This action selects the two message flows you highlighted, plus any that appear between the two in the list.
   - To select multiple message flows that do not appear in a sequence in the list, hold down Ctrl and click each entry you want.

2. When you have selected the message flows you want, click **Finish**.

   The items you selected are added to the Message Flow Types pane, from where you can include them in new message flows.

If you perform this task by right clicking on a message flow category in the Message Flow Types pane and clicking **Add —> Message Flow**, the items you select are added to the folder of the message flow category in the Message Flow Types pane.

Checking a message flow

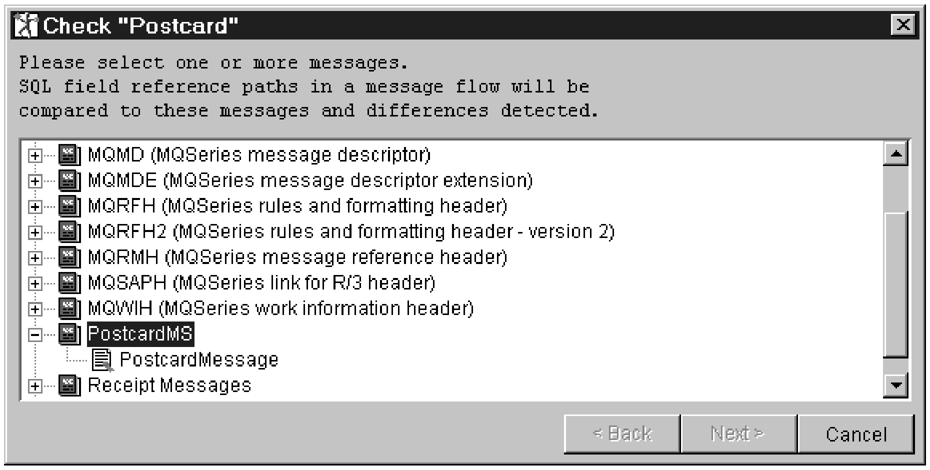
When you have created a message flow, you can use the message flow SmartGuide to check the following:

- All ESQL syntax is correct.
- All references to message fields are resolved.
- All message flow properties (including promoted properties) are valid.
- All message field names are recognized.

This option allows you to check for errors before you deploy the message flow, thus saving time and inconvenience.

To use the message flow check SmartGuide:

1. Right click the message flow you want to check, and select **Check message flow...**.
2. Select the message set and messages that are used in the message flow from the list presented in the dialog.

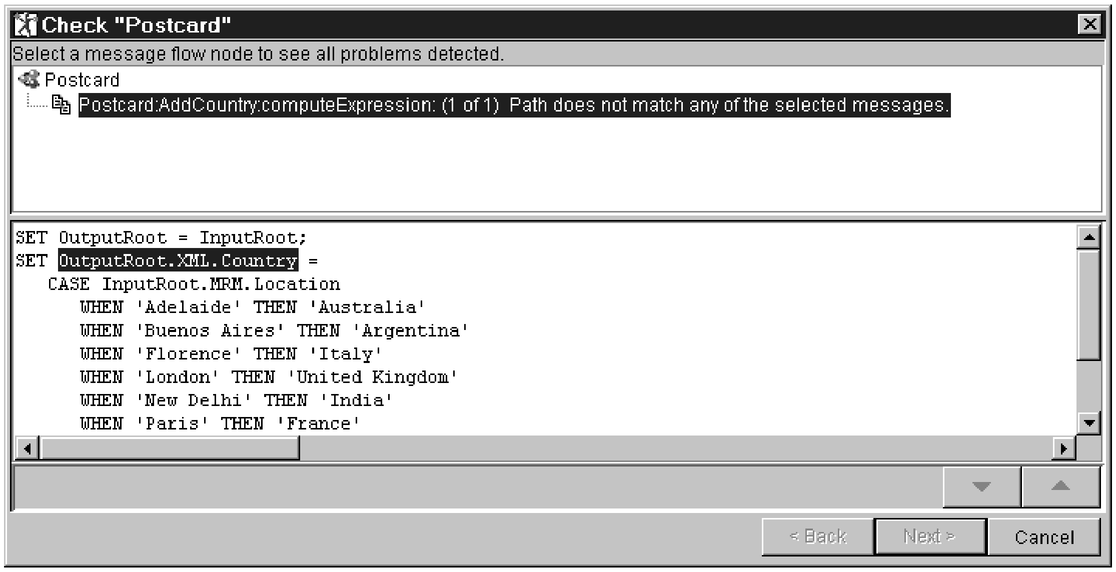


```
Please select one or more messages.
SQL field reference paths in a message flow will be
compared to these messages and differences detected.
```

- MQMD (MQSeries message descriptor)
- MQMDE (MQSeries message descriptor extension)
- MQRFH (MQSeries rules and formatting header)
- MQRFH2 (MQSeries rules and formatting header - version 2)
- MQRMH (MQSeries message reference header)
- MQSAPH (MQSeries link for R/3 header)
- MQWIH (MQSeries work information header)
- PostcardMS
  - PostcardMessage
- Receipt Messages

Figure 22. Check message flow dialog

3. Click **Next**. The SmartGuide performs the checks identified above, and presents a results window.



Postcard:AddCountry:computeExpression: (1 of 1) Path does not match any of the selected messages.

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Country =
   CASE InputRoot.MRM.Location
      WHEN 'Adelaide' THEN 'Australia'
      WHEN 'Buenos Aires' THEN 'Argentina'
      WHEN 'Florence' THEN 'Italy'
      WHEN 'London' THEN 'United Kingdom'
      WHEN 'New Delhi' THEN 'India'
      WHEN 'Paris' THEN 'France'
```

Figure 23. Check message flow results

You can use the information displayed in this window to identify the node that is incorrect, and the nature of the error. The upper part of the screen lists the errors found: the lower part of the screen provides the details of the error selected in the upper part. You can move through the list of errors using the arrow keys on the right hand side of the window.

You can leave this window active as a reference while you make the changes and corrections.

Including one message flow in another

You can create a message flow that includes a mixture of message flow nodes and existing message flows. You might want to do this, for example, if you have created a standard message flow to process errors or to perform a particular calculation. You can define this standard message flow once and include it in other message flows wherever it is required, which is easier than redefining the same sequence of nodes in each message flow that uses them.

A set of nodes created as a partial message flow for use in this way is also known as a subflow, or embedded flow.

Note: Any message flow that you intend to reuse in this way does not normally use the standard MQInput and MQOutput nodes to start and end the flow. Instead, it uses the Input Terminal and Output Terminal nodes that are included in the IBMPrimitives message category.

1. To include an existing, reusable message flow in a new message flow, you must begin by adding that message flow to your workspace, if it isn't already there, as described in "Adding a message flow to your workspace" on page 94.
2. Create the new message flow, following steps 1 and 2 on page 89.
3. In the Message Flow Types pane, drag the message flows that will make up your new flow into the Message Flow Definition pane.

   Embedded message flows have terminal icons that represent the Input Terminal and Output Terminal nodes they contain. For example, if the nested message flow has one Input Terminal node and two Output Terminal nodes, the message flow icon will have one input terminal and two output terminals, which you connect to other nodes in the higher-level flow in the usual way. You can rename these terminals if you want: for example, one of the Output Terminals might be for an error path, and you might rename this 'failure'.

The retail scenario described in Appendix A, "An example scenario" on page 257 uses an embedded flow to define an error routine. The subflow, and its inclusion into a larger flow, are illustrated in Figure 80 on page 285 and Figure 63 on page 257.

You can work with the subflow by adding it to your workspace and either selecting it in the Message Flow Types pane (as you can with any message flow), or by double-clicking on the node icon for the embedded flow displayed within the main flow in the Message Flow Definition pane.

The right-hand pane changes to display the configuration for that subflow. You can also reach this view by right-clicking the subflow node, and selecting *Open sub-flow*.

For example, if you double-click on the node **Exception handle1** in the main retail flow, which is a named instance of a subflow called **Exception Handling**, you will see the view shown in Figure 80 on page 285.

You can return to the parent flow by right-clicking the Message Flow Definition pane and selecting the menu item *Return to parent flow*.

Promoting message flow node properties

A message flow contains one or more message flow nodes, each of which is an instance of a message flow type (either an IBM Primitive, or one you have defined). You can promote the properties of these message flow nodes to apply to the message flow to which they belong. If you do this, any user of the message flow can set values for the properties of the nodes in the message flow, by setting them at the message flow level, without being aware of the message flow's internal structure.

For example, you might want to set the name of a data source as a property of the message flow, rather than a property of each individual node in the message flow that references that data source.

You are creating a message flow that accesses a database called SALESDATA. However, while you are testing the message flow, you want to use a test database called TESTDATA. If you set the data source properties of each individual node within the message flow to reference TESTDATA, you will have to update all these references when you put your message flow into production.

If you promote the data source property, you can set the properties for all of the individual nodes to be SALESDATA, and set the value of TESTDATA for the promoted property to override the node data source values while you test the message flow (the promoted property always takes precedence over the settings for the properties within any relevant nodes).

To promote message flow node properties to a message flow:

1. You must check out the message flow for which you want to promote properties. If it is not checked out, right click the entry for the message flow in the Message Flow Types pane, and click **Check Out**.

   The message flow contents are now displayed in the Message Flow Definition pane.

2. Right click the symbol of the message flow node whose properties you want to promote, and click **Promote Property**.

   The **Promote Property** dialog is displayed.

3. In the **Promote Property** dialog, the names of the properties of the message flow node are displayed in the left-hand pane. This pane is always fully expanded to show all properties that are available for promotion. If you have already promoted properties from this node, they do not appear in the left-hand pane, but in the right-hand pane.

   The names of the properties of the message flow itself, of which the message flow node is a part, are displayed in the right-hand pane. These are properties that have already been promoted up to the message flow. The original name of the property and the name of the message flow node from which it came, are shown beneath the property entry. This allows you to determine the specific node that is the origin of each promoted property, regardless of the name of the promoted property. See “Renaming promoted properties” on page 100 for information about renaming properties.

4. To promote a property from the message flow node to the message flow, drag its entry from the left-hand pane of the **Promote Property** dialog to the

right-hand pane and drop it in an empty part of the pane. It then appears at the top of the pane.

Figure 24 shows an example of the **Promote Property** dialog.

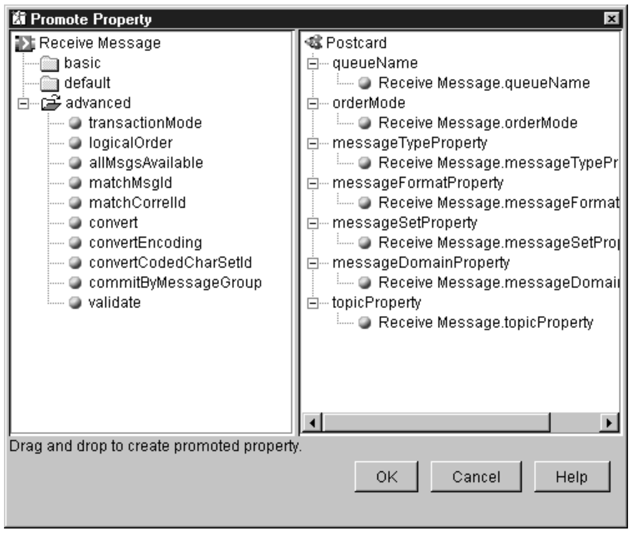


Figure 24. The Promote Property dialog. Some of the properties of the message flow node have been dragged across to the message flow and thus promoted.

5. When you have selected the properties you want to promote to the message flow, click **OK**.

The message flow node properties have been promoted to the message flow. To confirm this, in the Message Flow Types pane, right click the entry for this message flow and click **Properties**.

The **Properties** dialog of the message flow is displayed, showing the message flow node properties you promoted. If you now set a value for one of these properties, that value appears as the default value for the property whenever the message flow is itself included in other message flows.

When you select an embedded message flow within another message flow and view its properties, you see the promoted property values. If you look inside the embedded flow (that is, if you select **Open sub-flow**), you see the original values for the properties. The value of a promoted property does not replace the original property, but it takes precedence at deploy time.

Promoting properties through a hierarchy of message flows

The process of promoting message flow node properties can be repeated as you construct a hierarchy of message flows. You can promote properties from any level in the hierarchy to the next level above, and so on through the hierarchy. The value of a property is propagated from the highest point in the hierarchy at which it is set down to the original message flow node when the message flow is deployed to a broker. The value of that property on the original message flow node is overridden.

Converging multiple properties

It is possible for a promoted property to provide a value for several message flow node properties at once. For example, if a message flow contains two Database nodes that each refer to the same physical database, you have to define the physical database only once on the message flow. To do this, you promote several message flow properties to a single promoted property. Drag the property entry from the left-hand pane to the right-hand pane, and drop it onto an existing promoted property (instead of into the empty pane). You can now see the new property added under the existing promoted property.

Note: If the type of the property you are promoting does not match the type of the existing promoted property, when you drop the property onto the existing property, a new promoted property is created at the top of the pane.

Renaming promoted properties

To rename a promoted property:

1. In the **Promote Property** dialog, right click the promoted property, and click **Rename**.
2. In the **Rename** dialog, type the new name for the property. Click **OK**.

The new name of the property appears in the right-hand pane of the **Promote Property** dialog.

Deleting a promoted property from a message flow

To delete a promoted property from a message flow, in the **Promote Property** dialog, right click the promoted property, and click **Delete**.

Note: Any higher level message flow that has used this message flow, and that has set a value for the deleted property, is not automatically updated to reflect the deletion. However, when you deploy that message flow in the broker domain, the deleted property is ignored.

Promoting mandatory properties

If you promote a property that is mandatory (that is, the name appears in bold type in the properties dialog of the message flow node), the mandatory characteristic of the property is not preserved. You are recommended always to provide a default value for the property via the properties dialog of the message flow node from which the property originated.

Example: promoting message flow node properties

This example demonstrates how to promote message flow node properties.

1. Create a new message flow called Base.
2. Drag an MQInput node and an MQOutput node from the Message Flow Types pane into the Message Flow Definition pane.
3. In the Message Flow Definition pane, right click the symbol of the MQInput node, and click **Promote Property**.

   The **Promote Property** dialog is displayed.
4. Drag the properties you want to promote from the left-hand pane into the right-hand pane. Click **OK**.

5. Repeat steps 3 through 4 for the MQOutput node.
6. Create a new message flow called Middle.
7. Click on the entry for the message flow Middle in the Message Flow Types pane, then drag the message flow Base into the Message Flow Definition pane.

   A graphical symbol of the message flow labelled Base1 appears in the Message Flow Definition pane.
8. In the Message Flow Definition pane, right click the symbol of the Base1 message flow, and click **Properties**.

   The properties you promoted from the MQInput and MQOutput nodes appear as properties of the message flow Base1. You will notice that the values are not displayed: however, they will be acted on when the message flow is deployed.
9. Click **Cancel**.
10. In the Message Flow Definition pane, right click the symbol of the Base1 message flow again, and click **Promote Property**.

    The properties that appear in the left-hand pane of the **Promote Property** dialog are those you promoted from the message flow nodes in the Base1 message flow. You can promote these properties to the message flow Middle, displayed in the right-hand pane. If you do this, note that Base1 is listed as the originating message flow.
11. Repeat this procedure to add further levels to the hierarchy of message flows and to promote properties throughout the hierarchy.

Checking in message flows

When you have created a message flow, you can assign it to an execution group (as described in "Assigning message flows to execution groups" on page 202). You do not need to have checked the message flow into the shared configuration before assigning it. However, you must check it in before you can deploy it to one or more brokers in the message domain.

To check in a message flow, right click the folder of the message flow you want to check in within the Message Flow Types pane, or right click on the background of the Message Flow Definition pane, and click **Check In**.

The message flow is checked into the shared configuration. It still appears in your workspace (as evidenced by the inclusion of its folder in the Message Flow Types pane), but the **New** icon or the **Key** icon against its folder has disappeared.

Once you have checked in a message flow, it is available to other users from the shared configuration. If you want to make further changes to the message flow, you must first check it out of the shared configuration:

1. In the Message Flow Types pane, right click the folder of the message flow you want to edit.
2. Click **Check Out**.

   The message flow is checked out of the shared configuration. Its entry in the Message Flow Types pane has a **Key** icon against it to remind you that the definition is checked out.

Creating your own message nodes

For a full description of this task, including instructions for installing a node in the Control Center, see the *MQSeries Integrator Programming Guide*.

The IBM Primitives

Table 2 identifies the message flow nodes supplied with MQSeries Integrator V2.0.1, which are known as the IBM Primitives, and directs you to a detailed description of each.

With the exception of the Compute node, the input message received by a node, and the output message sent on by the node, are identical.

A number of these nodes allow manipulation of the message using ESQL. For further details about ESQL, and examples of its use, see Appendix C, "ESQL reference" on page 293.

You are recommended to refer to the MQSeries Integrator SupportPac MHI1 which is updated periodically to include further usage information for the primitive nodes. MHI1 is available on the MQSeries Web site at:

`http://www.ibm.com/software/ts/mqseries/txppacs/`

Table 2 (Page 1 of 2). The IBM Primitives

IBM Primitive	Function	See:
Check node	Compares the format of an incoming message with a predefined message specification.	Page 106.
Compute node	Derives an output message from an input message and, optionally, from data taken from a external database. A computation can be applied to each element of the input message before the output message is constructed.	Page 108.
Database node	Combines database operations with message processing.	Page 113.
DataDelete node	Deletes one or more rows from a database table.	Page 116.
DataInsert node	Inserts one or more rows in a database table.	Page 119.
DataUpdate node	Updates the contents of one or more rows in a database table.	Page 122.
Extract node	Derives an output message from the fields in an input message.	Page 125.
Filter node	Evaluates an input message against an ESQL expression.	Page 127.
FlowOrder node	Determines a specific order for the processing of a message.	Page 130.
Input Terminal	Provides an in terminal for an embedded message flow.	Page 132.
Label node	Receives a message from a RouteToLabel node.	Page 134.
MQInput node	Reads MQSeries messages from a specified message queue.	Page 136.
MQOutput node	Writes MQSeries messages to a specified message queue.	Page 141.
MQReply node	Sends a response message to the originator of the message that caused this message flow to be invoked.	Page 145.
NEONFormatter node	Transforms an input message using the NEON Formatter engine.	Page 148.

Table 2 (Page 2 of 2). The IBM Primitives

IBM Primitive	Function	See:
NEONRules node	Passes an input message to the NEON Rules engine for evaluation.	Page 150.
Output Terminal	Provides an out terminal for an embedded message flow.	Page 152.
Publication node	Publishes a message to subscribers.	Page 154.
ResetContentDescriptor node	Reparses the bit stream of an input message.	Page 156.
RouteToLabel	Routes a message to one or more specific destinations that are identified in the message.	Page 159.
Throw node	Throws an exception within a message flow.	Page 163.
Trace node	Generates a trace record.	Page 165.
TryCatch node	Catches any exceptions that are thrown by nodes further on in the message flow.	Page 168.
Warehouse node	Stores message data in a data repository.	Page 170.

Check node

Description

The Check node compares the format of a message arriving on its input terminal with a message-type specification that you supply when you configure the Check node. The message-type specification comprises any combination of the message domain, message set, and message type. The Check node checks only the message-type specification; it does not check the message body.

Check node terminals

Table 3 describes the terminals of the check node.

Table 3. Check node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
match	The output terminal to which the message is routed if its properties match the message-type specification.
failure	The output terminal to which the message is routed if its properties do not match the message-type specification. If the failure terminal is not connected to another message flow node, an exception is thrown.

Check node properties

These properties are displayed when you right click a Check node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Domain
Identifies the parser for the message. Values supported by MQSeries Integrator are MRM, XML, NEON, and BLOB.

Check Domain
If this field contains a check mark (√), the Domain value is to be considered part of the message-type specification.

Set
Identifies the message set containing the definition of the message.

Check Set
If this field contains a check mark (√), the Set value is to be considered part of the message-type specification.

Type
Identifies the message definition within the message set.

Check Type
If this field contains a check mark (√), the Type value is to be considered part of the message-type specification.

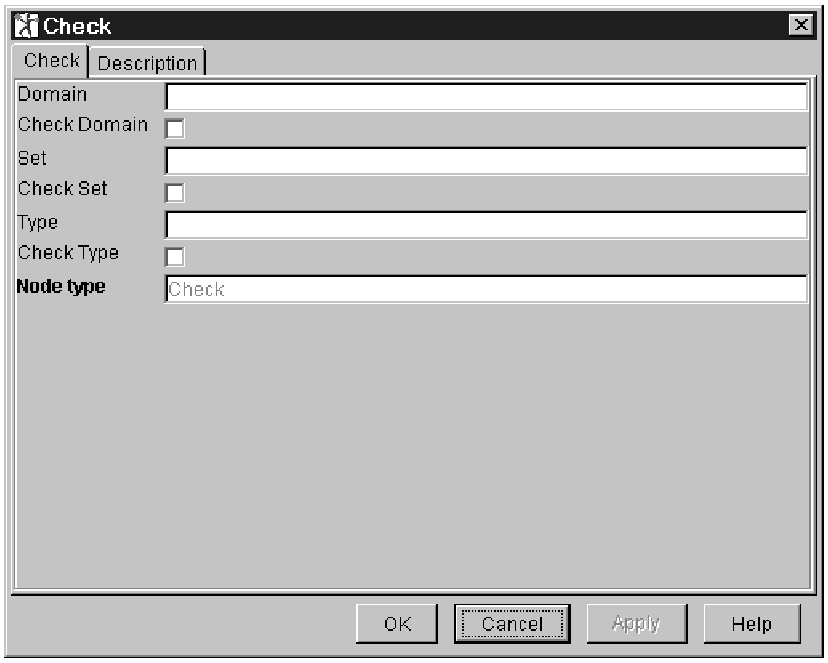
Configuring the check node

For a description of the properties of the Check node and their possible values, see "Check node properties" on page 106.

To configure a Check node:

1. In the Message Flow Definition pane, right click the symbol of the Check node you want to configure and click **Properties**.

   The **Check** dialog is displayed, as shown in Figure 25.



Domain
Check Domain
Set
Check Set
Type
Check Type
Node type: Check

Figure 25. The Check dialog

2. In the **Check** dialog, type values for those properties that you want to be considered part of the message-type specification. For each value you enter, select the relevant check box. For example, if you supply a Domain value, select the **Check Domain** check box.
3. If you want to provide a description of this instance of the Check node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Check** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this Check node.

Compute node

Description

The Compute node constructs a new message, or modifies elements (headers, header fields, and body data) within an existing message, or its associated destination or exception list, or both. These components of the message can be defined using an ESQL expression, and can be based on elements of both the input message and data from an external database. The expression can make use of arithmetic operators, text operators (for example, concatenation), logical operators, and other built-in functions.

Compute node terminals

Table 4 describes the terminals of the compute node.

Table 4. Compute node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal that outputs the transformed message.
failure	The output terminal to which the original message is routed if a failure is detected during the computation. For example, if an integer is divided by another integer that has a value of zero.

Compute node properties

These properties are displayed when you right click a Compute node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Data Source
If data from an external database is to be used in constructing the transformed message, Data Source identifies ODBC name for the database.

Transaction
The Transaction value, which is always automatic for a Compute node, specifies that the decision to commit or roll back the Compute node action depends on the success or failure of the message flow to which it belongs.

Compute Expression
The ESQL expression generated when you configure a Compute node.

Compute Mode
Identifies the components of the message with which you want to work in this Compute node. Any combination of Message (the default value), Exception, and Destination can be specified. The destination list structure represents the destinations to which the message will be sent. The exception list structure represents any exception conditions that have occurred during message processing.

The components of the message that are not included in your selection are propagated through the node unchanged (that is, the output value equals the input value). You can manipulate these components if you choose, but any modifications are valid within the node only, and are not passed to the next node.

If you are using the RouteToLabel node (described in “RouteToLabel node” on page 159) in the message flow for this Compute node, you can set up specific destinations in this compute node by selecting a Compute Mode that includes Destination.

For more information about the message, exception list, and destination list structures, see Appendix C, “ESQL reference” on page 293.

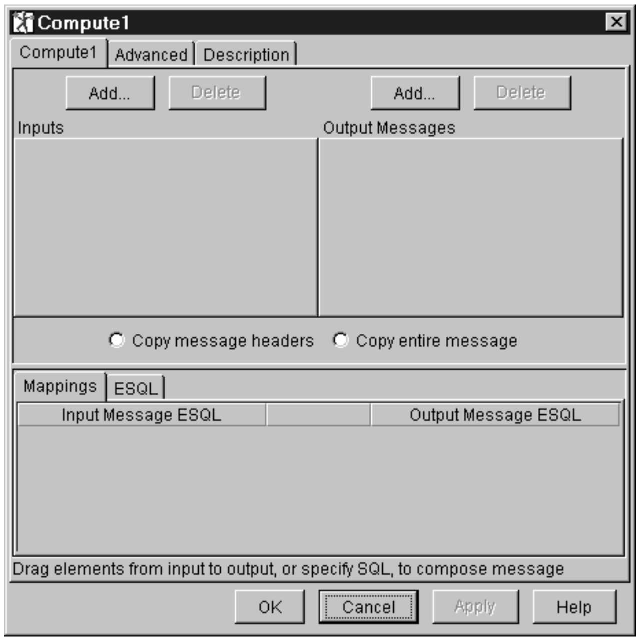
Configuring the Compute node

For a description of the properties of the Compute node and their possible values, see “Compute node properties” on page 108.

To configure a compute node:

1. In the Message Flow Definition pane, right click the symbol of the Compute node you want to configure and click **Properties**.

   The **Compute** dialog is displayed, as shown in Figure 26.



Drag elements from input to output, or specify SQL, to compose message

Figure 26. The Compute dialog

2. In the **Compute** dialog, click the left-hand **Add** button (or right-click on the left-hand pane and select **Add**) to define the Inputs.

   The **Add** dialog is displayed.

3. In the **Add** dialog, either:

   - Click **Message** and select the names of a message set and message from the drop-down lists.

   or

   - Click **Database table** and type values in the Data Source and Table Name fields. These two values identify the database and database table from which data will be taken.

   Click **OK**.

Depending on which you choose, the message or database tree structure appears in the Inputs pane of the **Compute** dialog. A tab is added to the Inputs pane for each input data source you specify. To delete any of these, click **Delete** when the relevant tab is to the fore.

When you select a database as a source of input, the Transaction property is displayed on the pane. This is always set to *Automatic*: you cannot change this value.

4. For any database you have added to the Inputs pane, you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Inputs pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Inputs pane.

   Repeat this process for each column you want to work with.

   Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

5. Repeat steps 3 and 4 for all input sources (messages or database tables) you require.

6. In the **Compute** dialog, click the right-hand **Add** button (or right-click on the right-hand pane and select **Add**) to define the Output Messages.

   The **Add** dialog is displayed, with **Message** preselected.

   Select the names of a message set and message from the drop-down lists.

   If you want this message to define the MessageSet and MessageType of the output message, select the **Use as message body** option that is located below the specification of the message and message set.

   Click **OK**.

7. To copy the entire message from the input message to the output message before you apply the computation, click **Copy entire message**.

   If you want to manipulate header information only, click **Copy message headers**.

8. Drag elements from the Inputs pane to the Output Messages pane to compose the output message.

   As you do this, ESQL statements are generated automatically in the Mappings section of the dialog.

   Click on the **ESQL** tab if you want to edit the generated ESQL directly, or if you want to add your own ESQL to customize the node. When you click on the **ESQL** tab, you will see a comment in the input window that reads:

   ```
   -- Enter SQL below this line. SQL above this line might be regenerated,
   causing any modifications to be lost.
   ```

The ESQL that is generated automatically by actions that you take (selecting Mappings or radio buttons) appears above this comment. You can make changes to this ESQL, but you must be aware that any changes you make above the line are discarded if the ESQL is regenerated. This will happen if you:

- Select **Use as message body** in an output message panel
- Select or deselect the *copy message headers* radio button
- Select or deselect the *copy entire message* radio button
- Add a mapping or drag-drop to the ESQL field

The following actions do **not** cause the ESQL to be regenerated:

- Modifying long or short description properties
- Modifying the *Compute mode* property on the Advanced tab
- Adding an input message
- Adding an input table
- Adding an output message if you do not select *use as message body*
- Adding an output table

If you want to add extra ESQL statements of your own, you are recommended to add these below the comment line. All data you add below the comment line is preserved when the generated ESQL is regenerated. If you have added your own statements above the comment, they are lost if the ESQL is regenerated.

If you make any changes to the comment line itself, these will be overwritten with the standard comment line if the ESQL is regenerated.

The comment is passed to the run-time code, but as it is only a comment, it is ignored.

For information about valid ESQL statements, see Appendix C, "ESQL reference" on page 293.

9. Click on the **Advanced** tab to select a Compute Mode value from the drop-down list.

   If you want the Compute node to act on the exception list structure or the destination list structure, you must remember to select the appropriate Compute Mode. If you fail to do this, only the Message structure (the default value) is used.

   The following values are supported:

   - Message
   - Destination
   - Destination and Message
   - Exception
   - Exception and Message
   - Exception and Destination
   - All

10. If you want to provide a description of this instance of the Compute node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Compute** dialog. Type a short description, or a long description, or both.

11. Click **OK** to finish configuring the Compute node.

For an example of configuring a Compute node within a message flow, see the retail scenario described in Appendix A, An example scenario (section “Stock flow” on page 273). This provides an example configuration for both an MRM message and an XML message.

Database node

Description

The Database node allows a database operation in the form of an ESQL statement to be applied to the specified ODBC data source. Data from the input message can be substituted into the ESQL expression, and transforms can be applied to the data as part of that assignment. A node property controls whether the update to the database is committed immediately, or deferred until the completion of processing of the message flow at which time the update is committed or rolled back according to the overall completion status of the message flow.

Database node terminals

Table 5 describes the terminals of the database node.

Table 5. Database node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed following the execution of the database statement.
failure	The output terminal to which the original message is routed if a failure is detected during execution of the database statement. For example, if the connection to the database fails, or if the table specified is invalid.

Database node properties

These properties are displayed when you right click a Database node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Data Source
Identifies the ODBC name of the external database to which the ESQL expression is to be applied.

Statement
Is the ESQL statement or expression, generated automatically from the values you specify when configuring the Database node, that performs the database operation.

Transaction
The Transaction value specifies whether the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the Database node action depends on the success or failure of the message flow to which it belongs. This is the default value.

Commit
The action of the Database node is to be committed, irrespective of the success or failure of the message flow as a whole.

Treat warnings as errors
Specifies whether warning messages generated during this node's processing are to be treated as errors, causing the message to be routed to the failure terminal.

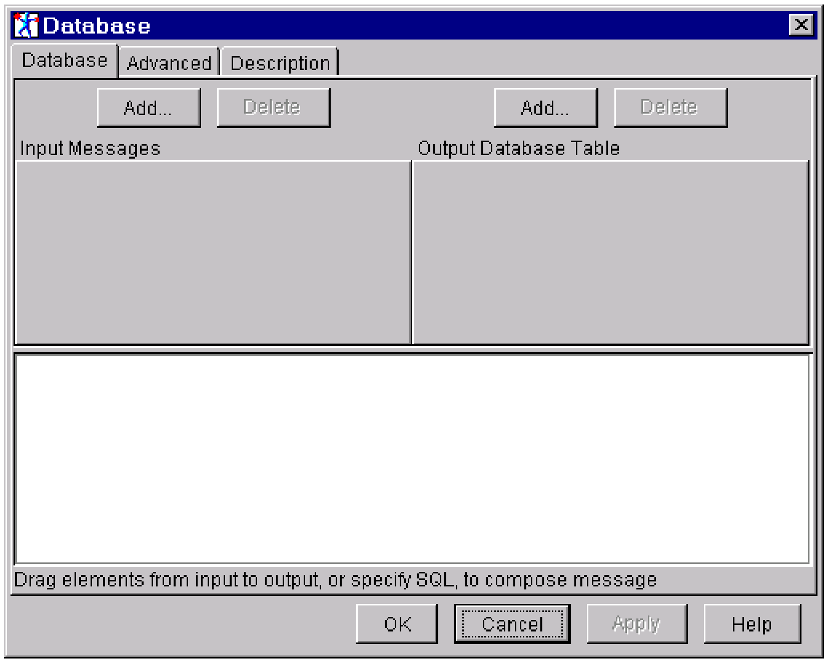
Configuring the Database node

For a description of the properties of the Database node and their possible values, see "Database node properties" on page 113.

To configure a Database node:

1. In the Message Flow Definition pane, right click the symbol of the Database node you want to configure and click **Properties**.

   The **Database** dialog is displayed, as shown in Figure 27.



Drag elements from input to output, or specify SQL, to compose message

Figure 27. The Database dialog

2. In the **Database** dialog, click **Add** to define the Input Message.

   The **Add** dialog is displayed.

3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message from the drop-down lists to define the Input Messages value. Click **OK**.

   The message tree structure appears in the Input Messages pane of the **Database** dialog, and a tab is added to the Input Messages pane showing the name of the message.

   Repeat this step if you want to identify additional messages. To delete any of the messages you have added to the Input Messages pane, click **Delete** when the relevant tab is to the fore.

4. Click **Add** to define the Output Database Table.

   The **Add** dialog is displayed.

5. In the **Add** dialog, **Database table** is preselected. Enter Data Source and Table Name values. Click **OK**.

   The database tree structure appears in the Output Database Table pane of the dialog.

6. Now you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Output Database Table pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Output Database Table pane.

   Repeat this process for each column you want to work with.

   Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Drag elements from the Input Messages pane to the database columns in Output Database Table pane to compose the output data. As you do this, ESQL statements are generated automatically.

   You can edit the mappings that you generate by dragging input to output. To edit the ESQL, double-click on it and enter your modifications. To indicate that the field can be edited, its border changes to yellow.

8. From the **Transaction Mode** drop-down list, select Automatic or Commit.

9. If you want warnings to be treated as errors, click the **Advanced** tab of the **Database** dialog, and select the **Treat warnings as errors** check box.

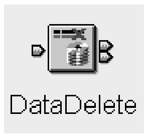
10. If you want to provide a description of this instance of the Database node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Database** dialog. Type a short description, or a long description, or both.

11. Click **OK** to finish configuring this Database node.

For an example of configuring a Database node within a message flow, see the retail scenario described in Appendix A, An example scenario (section “Updating the Multibuy database” on page 279).

DataDelete node

Description



The DataDelete node is a specialized form of the Database node that allows deletion of one or more rows from a table in the specified ODBC data source. Data from the input message can be substituted into the ESQL expression, and transforms can be applied to the data as part of that assignment. A property controls whether the update to the database is committed immediately, or deferred until the completion of processing of the message flow at which time the update is committed or rolled back according to the overall completion status of the message flow.

DataDelete node terminals

Table 6 describes the terminals of the DataDelete node.

Table 6. DataDelete node terminals

Terminal	**Description**
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed following the execution of the data delete statement. The message is identical to the input message.
failure	The output terminal to which the original message is routed if a failure is detected during execution of the data delete statement. For example, if the connection to the database fails, or if the table specified is invalid.

DataDelete node properties

These properties are displayed when you right click a DataDelete node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Table
Identifies the database table from which rows are to be deleted.

Transaction
The Transaction value specifies whether the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the DataDelete node action depends on the success or failure of the message flow to which it belongs. This is the default value.

Commit
The action of the DataDelete node is to be committed, irrespective of the success or failure of the message flow as a whole.

Statement
The ESQL statement or expression, generated automatically from the values you specify when configuring a DataDelete node, that performs the delete operation.

Data Source
It is the ODBC name of the database containing the table from which rows will be deleted.

Treat warnings as errors
Specifies whether warning messages generated during this node's processing are to be treated as errors, causing the message to be routed to the failure terminal.

Configuring a DataDelete node

For a description of the properties of the DataDelete node and their possible values, see "DataDelete node properties" on page 116.

To configure a DataDelete node:

1. In the Message Flow Definition pane, right click the DataDelete node you want to configure and click **Properties**.

   The **DataDelete** dialog is displayed, as shown in Figure 28.

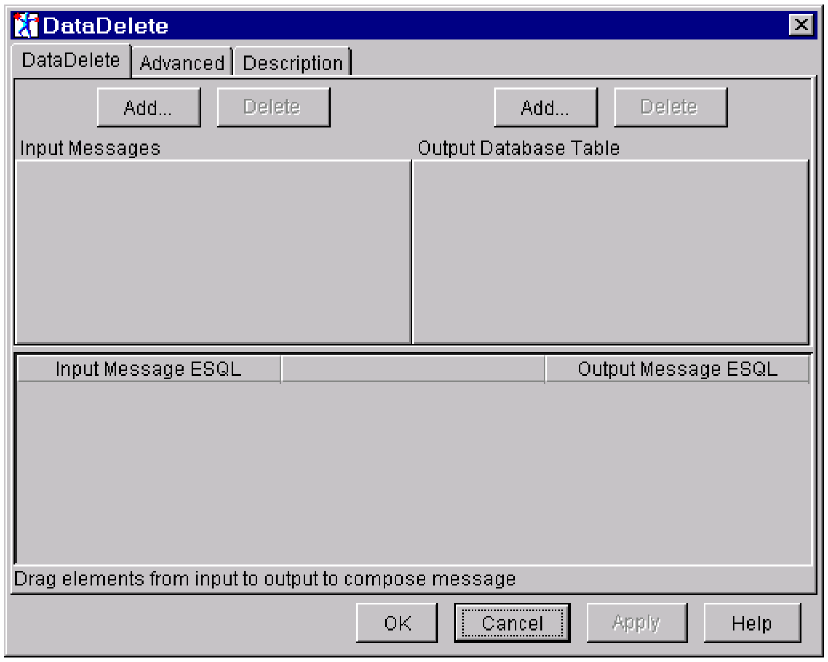


Figure 28. The DataDelete dialog

2. In the **DataDelete** dialog, click **Add** to define the Input Messages.

   The **Add** dialog is displayed.

3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message from the drop-down lists to define the Input Messages value. Click **OK**.

   The message tree structure appears in the Input Messages pane of the **DataDelete** dialog.

   Repeat this step for additional messages. A tab is added to the Input Messages pane for each message you add. To delete any message from the Input Messages pane, click **Delete** when the relevant tab is to the fore.

4. Click **Add** to define the Output Database Table.

   The **Add** dialog is displayed.

5. In the **Add** dialog, **Database table** is preselected. Enter Data Source and Table Name values. Click **OK**.

   The database tree structure is shown in the Output Database Table pane.

6. Now you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Output Database Table pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Output Database Table pane.

   Repeat this process for each column you want to work with.

   Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Drag elements from the Input Messages pane to the Output Database Table columns to compose the output. As you do this, ESQL mappings are generated automatically. WHERE clauses are generated to delete rows from the database.

   You can edit the mappings that you generate by dragging input to output. To edit the ESQL, double-click on it and enter your modifications. To indicate that the field can be edited, its border changes to yellow.

8. From the **Transaction Mode** drop-down list, select Automatic or Commit.

9. If you want warnings to be treated as errors, click the **Advanced** tab of the **DataDelete** dialog, and select the **Treat warnings as errors** check box.

10. If you want to provide a description of this instance of the DataDelete node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **DataDelete** dialog. Type a short description, or a long description, or both.

11. Click **OK** to finish configuring this DataDelete node.

DataInsert node

Description

The DataInsert node is a specialized form of the Database node that allows insertion of one or more rows into a table in the specified ODBC data source. Data from the input message can be substituted into the ESQL expression, and transforms can be applied to the data as part of that assignment. A property controls whether the update to the database is committed immediately, or deferred until the completion of processing of the message flow at which time the update is committed or rolled back according to the overall completion status of the message flow.

DataInsert terminals

Table 7 describes the terminals of the DataInsert node.

Table 7. DataInsert node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed following the execution of the data insert statement.
failure	The output terminal to which the message is routed if a failure is detected during execution of the data insert statement. For example, if the connection to the database fails, or if the table specified is invalid.

DataInsert node properties

These properties are displayed when you right click a DataInsert node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Table
Is the database table into which the new row is to be inserted.

Transaction
The Transaction value specifies whether the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the DataInsert node action depends on the success or failure of the message flow to which it belongs. This is the default value.

Commit
The action of the DataInsert node is to be committed, irrespective of the success or failure of the message flow as a whole.

Statement
Is the ESQL statement or expression, generated automatically from the values you specify when configuring the DataInsert node, that performs the insert operation.

Data Source
It is the ODBC datasource name of the database containing the table into which rows are to be inserted.

Treat warnings as errors
Specifies whether warning messages generated during this node's processing are to be treated as errors, causing the message to be routed to the failure terminal.

Configuring a DataInsert node

For a description of the properties of the DataInsert node and their possible values, see "DataInsert node properties" on page 119.

To configure a DataInsert node:

1. In the Message Flow Definition pane, right click the symbol of the DataInsert node you want to configure and click **Properties**.

   The **DataInsert** dialog is displayed, as shown in Figure 29.

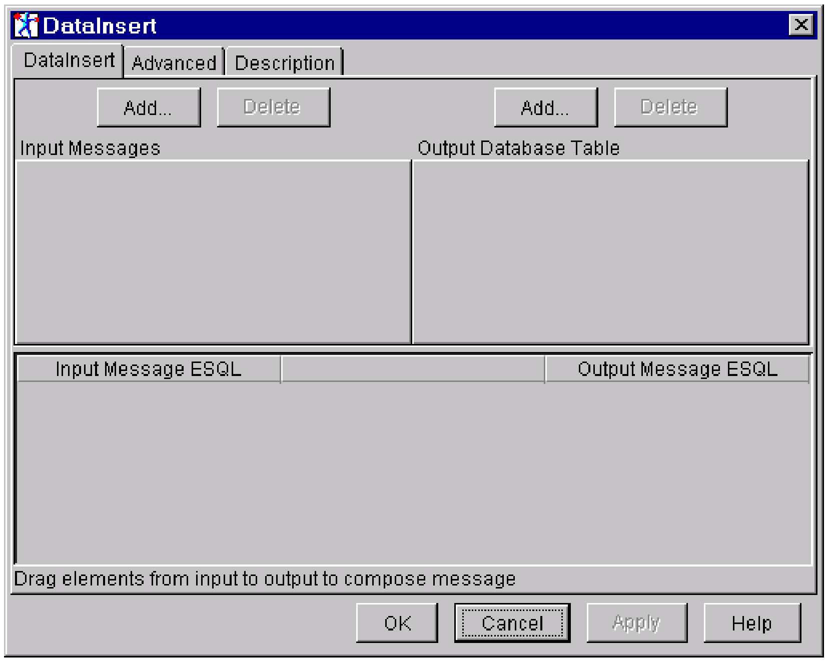


Figure 29. The DataInsert dialog

2. In the **DataInsert** dialog, click **Add** to define the Input Messages.

   The **Add** dialog is displayed.

3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message from the drop-down lists to define the Input Messages value. Click **OK**.

   The message tree structure appears in the Input Messages pane of the **DataInsert** dialog. A tab showing the name of the message is added to the Input Messages pane.

   Repeat this step for additional messages. To delete a message from the Input Messages pane, click **Delete** when the relevant tab is to the fore.

4. Click **Add** to define the Output Database Table.

   The **Add** dialog is displayed.

5. In the **Add** dialog, **Database Table** is preselected. Enter Data Source and Table Name values. Click **OK**.

   The database tree structure is added to the Output Database Table pane.

6. Now you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Output Database Table pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Output Database Table pane.

   Repeat this process for each column you want to work with.

   Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Drag elements from the Input Messages pane to the Output Database Table pane to compose the output. As you do this, ESQL mappings are generated automatically. A VALUE clause is generated to insert each new row into the database.

   You can edit the mappings that you generate by dragging input to output. To edit the ESQL, double-click on it and enter your modifications. To indicate that the field can be edited, its border changes to yellow.

8. From the **Transaction Mode** drop-down list, select Automatic or Commit.

9. If you want warnings to be treated as errors, click the **Advanced** tab of the **DataInsert** dialog, and select the **Treat warnings as errors** check box.

10. If you want to provide a description of this instance of the DataInsert node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **DataInsert** dialog. Type a short description, or a long description, or both.

11. Click **OK** to finish configuring this DataInsert node.

DataUpdate node

Description

The DataUpdate node is a specialized form of the Database node that allows the modification of one or more rows in a table in specified ODBC data source. Data from the input message can be substituted into the ESQL expression, and transforms can be applied to the data as part of that assignment. A property controls whether the update to the database is committed immediately, or deferred until the completion of processing of the message flow at which time the update is committed or rolled back according to the overall completion status of the message flow.

DataUpdate node terminals

Table 8 describes the terminals of the DataUpdate node.

Table 8. DataUpdate node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed following the execution of the data update statement.
failure	The output terminal to which the original message is routed if a failure is detected during execution of the data update statement. For example, if the connection to the database fails, or if the table specified is invalid.

DataUpdate node properties

These properties are displayed when you right click a DataUpdate node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Table
Is the database table in which rows are to be updated.

Transaction
The Transaction value specifies whether the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the DataUpdate node action depends on the success or failure of the message flow to which it belongs. This is the default value.

Commit
The action of the DataUpdate node is to be committed, irrespective of the success or failure of the message flow as a whole.

Statement
The ESQL statement or expression, generated automatically from the values you specify when configuring the DataUpdate node, that performs the update operation.

Data Source
The name ODBC datasource name of the database containing the table in which rows are to be updated.

Treat warnings as errors
Specifies whether warning messages generated during this node's processing are to be treated as errors, causing the message to be routed to the failure terminal.

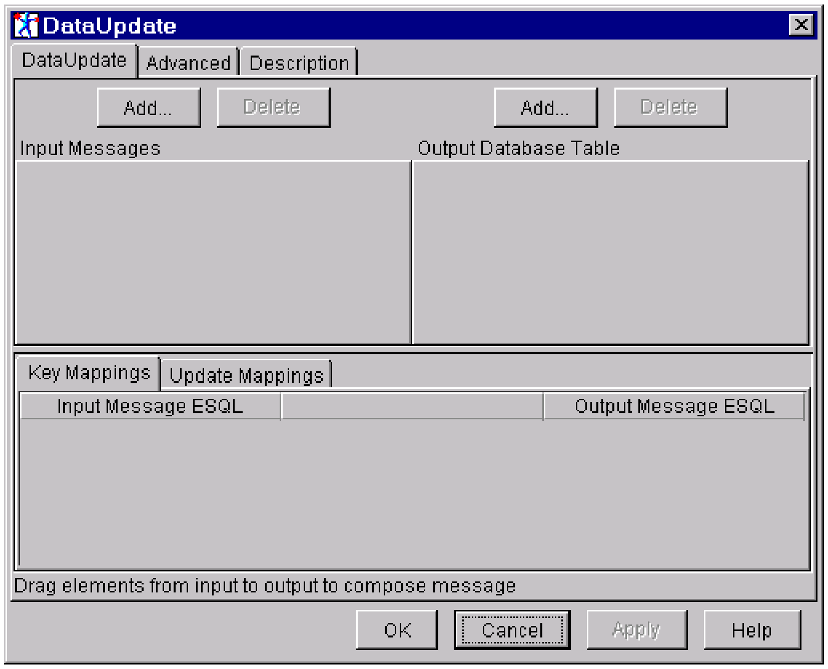
Configuring a DataUpdate node

For a description of the properties of the DataUpdate node and their possible values, see "DataUpdate node properties" on page 122.

To configure a DataUpdate node:

1. In the Message Flow Definition pane, right click the symbol of the DataUpdate node you want to configure and click **Properties**.

   The **DataUpdate** dialog is displayed, as shown in Figure 30.



Drag elements from input to output to compose message

Figure 30. The DataUpdate dialog

2. In the **DataUpdate** dialog, click **Add** to define the Input Messages.

   The **Add** dialog is displayed.

3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message from the drop-down lists to define the Input Messages value. Click **OK**.

   The message tree structure appears in the Input Messages pane of the **DataUpdate** dialog. A tab is added to the Input Messages pane showing the name of the message.

   Repeat this step for additional messages. To delete a message from the Input Messages pane, click **Delete** when the relevant tab is to the fore.

4. Click **Add** to define the Output Database Table.

   The **Add** dialog is displayed.

5. In the **Add** dialog, **Database table** is preselected. Enter Data Source and Table Name values. Click **OK**.

   The database tree structure appears in the Output Database Table pane. A tab is added to the Output Database Table pane showing the name of the database table.

6. Now you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Output Database Table pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Output Database Table pane.

   Repeat this process for each column you want to work with.

   Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Drag elements from the Input Messages pane to the columns in the Output Database Table pane to generate the Key Mappings and the Update Mappings.

   The Key Mappings specify the WHERE conditions in the generated ESQL statement. The Update Mappings specify the changes to be made to the selected columns in the database table.

   You can edit the mappings that you generate by dragging input to output. To edit the ESQL, double-click on it and enter your modifications. To indicate that the field can be edited, its border changes to yellow.

8. From the **Transaction Mode** drop-down list, select Automatic or Commit.

9. If you want warnings to be treated as errors, click the **Advanced** tab of the **DataUpdate** dialog, and select the **Treat warnings as errors** check box.

10. If you want to provide a description of this instance of the DataUpdate node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **DataUpdate** dialog. Type a short description, or a long description, or both.

11. Click **OK** to finish configuring this DataUpdate node.

Extract node

Description

The Extract node derives an output message from an input message. The output message comprises only those elements of the input message that you specify for inclusion when configuring the Extract node.

Extract node terminals

Table 9 describes the terminals of the Extract node.

Table 9. Extract node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the transformed message is routed.
failure	The output terminal to which the message is routed if an element specified for inclusion in the output message is not present in the input message. The message consists of only those elements that could be extracted from the input message.

Extract node properties

These properties are displayed when you right click an Extract node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Element List
Is the list of elements to be extracted.

Compute Expression
The expression to be evaluated. This is generated automatically when you configure an Extract node.

Configuring an Extract node

For a description of the properties of the Extract node and their possible values, see “Extract node properties.”

To configure an Extract node:

1. In the Message Flow Definition pane, right click the symbol of the Extract node you want to configure and click **Properties**.

   The **Extract** dialog is displayed, as shown in Figure 31 on page 126.

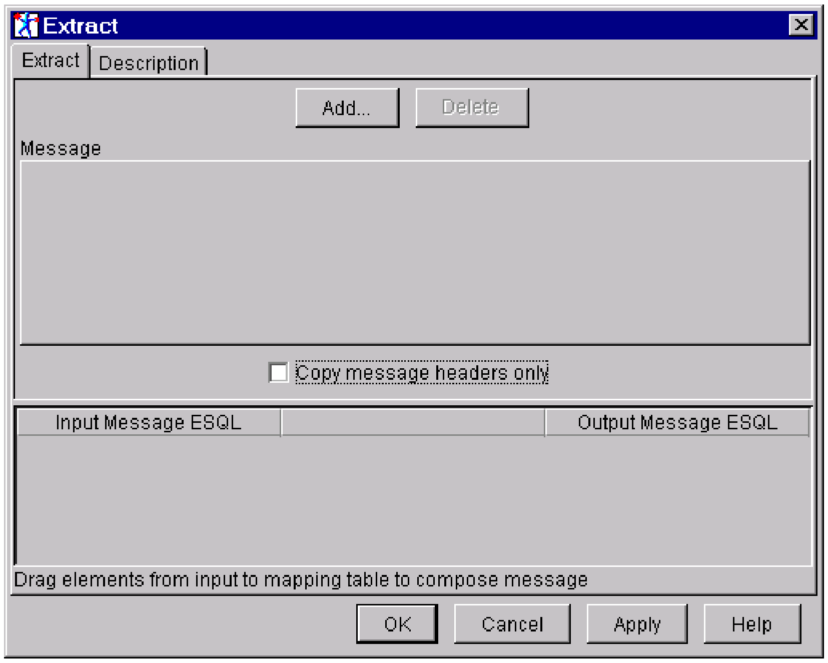


Figure 31. The Extract dialog

2. In the **Extract** dialog, click **Add** to define the input message.

   The **Add** dialog is displayed, with **Message** preselected.

3. In the **Add** dialog, select a message set and message from the drop-down lists. Click **OK**.

   The message tree structure appears in the Message pane. A tab showing the name of the message is added to the Message pane.

   Repeat this step for additional messages. To delete a message from the Message pane, click **Delete** when the relevant tab is to the fore.

4. Drag elements from the Message pane down to the ESQL pane (the mapping table) to compose the output message.

   For example, if you have a message consisting of the elements Name, Street, and Town, and you want to extract only the Town value, drag Town into the ESQL pane to create the output message.

5. If required, select Copy message headers.

6. If you want to provide a description of this instance of the Extract node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Extract** dialog. Type a short description, or a long description, or both.

7. Click **OK** to finish configuring the Extract node.

| For an example of configuring an Extract node within a message flow, see the retail
| scenario described in Appendix A, An example scenario (section "Extracting
| elements from the message" on page 271).

Filter node

Description

The Filter node routes a message according to message content using a filter expression specified in ESQL. The filter expression can include elements of the input message or message properties. It can also use data held in an external database. The output terminal to which the message is routed depends on whether the expression is evaluated to true, false, or unknown.

Filter node terminals

Table 10 describes the terminals of the Filter node.

Table 10. Filter node terminals

Terminal	**Description**
in	The input terminal that accepts a message for processing by the node.
true	The output terminal to which the message is routed if the specified filter expression evaluates to true.
false	The output terminal to which the message is routed if the specified filter expression evaluates to false.
unknown	The output terminal to which the message is routed if the specified filter expression evaluates to unknown. For example, if the field in the message that is being referenced does not exist.
failure	The output terminal to which the message is routed if a failure is detected during the filter operation. For example, if an integer is divided by another integer that has a value of zero.

Filter node properties

These properties are displayed when you right click a Filter node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Data Source
Is the name of the database.

Transaction
Specifies that the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the Filter node action depends on the success or failure of the message flow to which it belongs. This is the default value.

No
The action of the Filter node is not part of a larger transaction.

Filter Expression
This defines the expression, for example, `SQL WHERE`, against which the input data is to be evaluated.

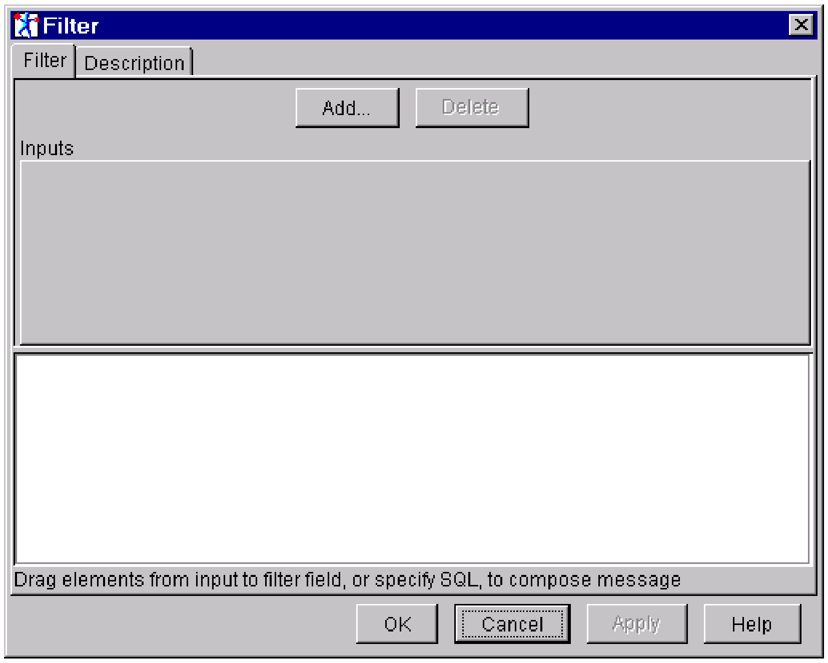
Configuring a filter node

For a description of the properties of the Filter node and their possible values, see "Filter node properties" on page 127.

To configure a Filter node:

1. In the Message Flow Definition pane, right click the symbol of the Filter node you want to configure and click **Properties**.

   The **Filter** dialog is displayed, as shown in Figure 32.



Drag elements from input to filter field, or specify SQL, to compose message

Figure 32. The Filter dialog

   In the **Filter** dialog, click **Add** to define the input message.

   The **Add** dialog is displayed.

2. In the **Add** dialog, either:

   - Click **Message** and select the names of a message set and message from the drop-down lists.

   or

   - Click **Database** and type the names of the database and database table from which data will be used.

   Click **OK**.

   Depending on which you choose, the message or database tree structure appears in the **Filter** dialog. A tab is added to the Inputs pane showing the name of the message or database table.

   To delete a message or database table from the Inputs pane, click **Delete** when the relevant tab is to the fore.

3. For any database you have added to the Inputs pane, you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Inputs pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Inputs pane.

   Repeat this process for each column you want to work with.

4. Drag elements from the Inputs pane to the filter field at the bottom of the dialog to generate the ESQL mappings. You can also edit the ESQL field directly. (For information about valid ESQL, see Appendix C, "ESQL reference" on page 293.)

5. If you want to provide a description of this instance of the Filter node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Filter** dialog. Type a short description, or a long description, or both.

6. Click **OK** to finish configuring this Filter node.

| For an example of configuring a Filter node within a message flow, see the retail
| scenario described in Appendix A, An example scenario (section "Filtering multibuy
| records" on page 276).

FlowOrder node

Description

The FlowOrder node enables you to control the order in which a message is processed by a message flow. You can use this node to specify the order in which each message is propagated to each (of two) output terminals. The message is only propagated to the second output terminal if propagation to the first output terminal is successful.

You can include this node in a message flow at any point where the order of execution of subsequent nodes is important. The FlowOrder node by definition propagates the input message through the first output terminal and its target node or nodes before propagating the input message to the second output terminal and its target node or nodes. The default behavior of all other nodes is for the order of propagation to any output terminal to be random and unpredictable.

If you connect multiple target nodes to the first terminal, or the second terminal or both, the order in which the multiple connections on each terminal are processed is random and unpredictable. However, the message is propagated to all target nodes connected to the first terminal, and all must complete successfully before it is propagated to any connection on the second output terminal.

FlowOrder node terminals

Table 11 describes the terminals of the FlowOrder node.

Table 11. FlowOrder node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
first	The output terminal to which the message is routed in the first instance.
second	The output terminal to which the message is routed in the second instance.
failure	The output terminal to which the message is routed if an error is encountered.

FlowOrder node properties

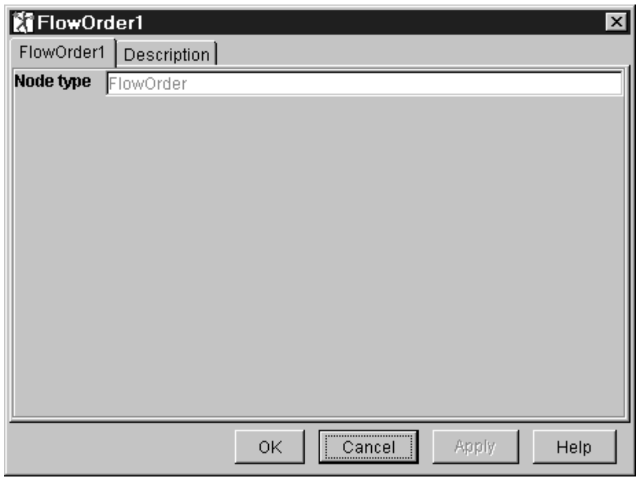
The FlowOrder node has no properties.

Configuring a FlowOrder node

The FlowOrder node has no configurable properties. However, you can provide a description of this instance of the node, as follows:

1. In the Message Flow Definition pane, right click the FlowOrder node you want to configure and click **Properties**.

   The **FlowOrder** dialog is displayed, as shown in Figure 7 on page 18.



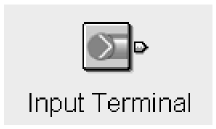
Node type FlowOrder

Figure 33. The FlowOrder dialog

2. Click the **Description** tab of the **FlowOrder** dialog. Type a short description, or a long description, or both.
3. Click **OK** to finish configuring this FlowOrder node.

Input Terminal

Description



The Input Terminal provides an in terminal for an embedded message flow.

From an Input Terminal, you can make a connection to any in terminal on any message flow node. The Input Terminal is the first node of a subflow that you can embed in another flow. An MQInput node is not required in a subflow. For more information about embedded flows, see “Including one message flow in another” on page 97.

Input Terminal node terminals

Table 12 describes the terminals of the Input Terminal.

Table 12. Input Terminal node terminals

Terminal	Description
out	The output terminal to which the message is routed.

Input Terminal properties

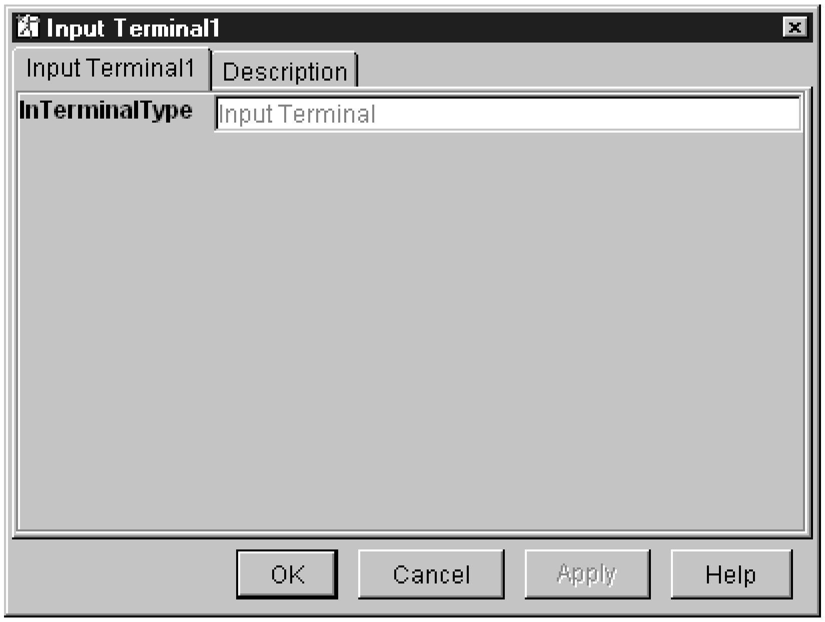
The Input Terminal has no properties.

Configuring the Input Terminal

The Input Terminal has no configurable properties. However, you can provide a description of this instance of the node, as follows:

1. In the Message Flow Definition pane, right click the Input Terminal for which you want to provide a description and click **Properties**.

   The **Input Terminal** dialog is displayed, as shown in Figure 34 on page 133.



InTerminalType: Input Terminal

| *Figure 34. The Input Terminal dialog*

| 2. In the **Input Terminal** dialog, click the **Description** tab. Type a short
| description, or a long description, or both.

| 3. Click **OK** to finish configuring this Input Terminal.

Label node

Description

The Label node is a named destination for a message processed by a RouteToLabel node. The Label node is identified by an entry in a destination list of the message when it is processed by a RouteToLabel node.

The combination of a RouteToLabel node with Label nodes provides a level of dynamic routing within a message flow, with the destination of the message following the RouteToLabel node being determined by the contents of the destination list within the message itself.

Typically, a label node connects to a subflow that processes each message in a specific way, and either ends in an output node or in another RouteToLabel node.

Figure 45 on page 161 illustrates a message flow that is made up of subflows that are associated with the main flow using RouteToLabel and Label nodes.

Label node terminals

Table 13 describes the terminals of the Label node.

Table 13. Label node terminals

Terminal	**Description**
out	The output terminal to which the message is routed.

Label node properties

These properties are displayed when you right click a Label node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Label Name

Is the unique name of the node to which the message is propagated. You can specify any value or characters in this field.

This value is not the same as the name of the instance of the node created. The name of the Label node might be enhanced by the Control Center when the message flow of which this node is a part is incorporated as a nested message flow within one or more other message flows. The Label Name is referenced by the RouteToLabel node and must therefore always retain the same value.

For example, you create a Label node in a message flow and name this instance of the node Test Route 1. You set the Label Name property of the node Test Route 1 to the value testroute1. The Label name value must be set to a value that will be set in a Destination list by a Compute node, and referenced in (and removed from) the Destination list by a RouteToLabel node. A message with a destination of testroute1 will be routed to node Test Route 1. A message with a destination of Test Route 1 will not.

The uniqueness of the name within the entire message flow is checked when that message flow is instantiated in the broker.

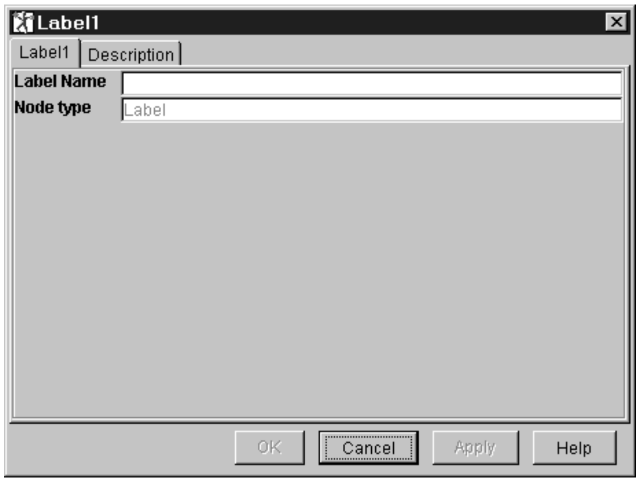
Configuring a Label node

For a description of the properties of the Label node and the possible values, see "Label node properties" on page 134.

To configure a Label node:

1. In the Message Flow Definition pane, right click the Label node you want to configure and click **Properties**.

   The **Label** dialog is displayed, as shown in Figure 35.



Label Name
Node type Label

Figure 35. The Label dialog

2. In the **Label** dialog, enter the string value that is the unique label for this node.
3. If you want to provide a description of this instance of the Label node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Label** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this Label node.

MQInput node

Description

The MQInput node uses MQGET to read a message from an MQSeries message queue defined on the broker's queue manager, and establishes the processing environment for the message.

MQInput routes messages to the out terminal. If this fails, the message is retried. If the retry out expires (as defined by the BackoutThreshold attribute of the input queue), the message is routed to the failure terminal. If this is not connected, the message is written to the backout queue.

If the message is caught by this node after an exception has been thrown further on in the message flow, the message is routed to the catch terminal.

You have to ensure that you define a backout requeue queue or a dead letter queue (DLQ) to prevent the message looping continuously through the node.

You must use the supplied MQInput node: you cannot replace it with a user-written equivalent.

MQInput node terminals

Table 14 describes the terminals of the MQInput node.

Table 14. MQInput node terminals

Terminal	Description
out	The output terminal to which the message is routed.
failure	The output terminal to which the message is routed if the backout count of the message is greater than or equal to the backout count threshold on the queue. Only failures internal to the MQInput node are routed to its failure terminal.
catch	The output terminal to which the message is propagated if an exception is thrown downstream and then caught by this node.

MQInput node properties

These properties are displayed when you right click an MQInput node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Queue Name
Is the name of the MQSeries queue from which the MQInput node reads messages. The queue manager associated with this queue is the broker's queue manager. The maximum length of a queue name is 48 characters.

Message Domain
Is the name of the message domain of the input message. If the message has an MQRFH2 header with an <mcd> folder, this property is derived from the header.

For further information about supported message domains, see "Message domains" on page 30.

Message Set
Is the name of the message set of the input message. For an MRM message, this is the message set identifier. If the message has an MQRFH2 header with an <mcd> folder, this property is derived from the header.

Message Type
Is the name of the message type of the input message. For an MRM message, this is the message identifier. If the message has an MQRFH2 header with an <mcd> folder, this property is derived from the header.

Message Format
Is the name of the message format of the input message. If the message has an MQRFH2 header with an <mcd> folder, this property is derived from the header.

Topic
Is the topic identifier. If the message has an MQRFH2 header with an <psc> folder, this property is derived from the header.

Transaction Mode
Defines whether this instance of a message flow is under transaction control. Valid values are:

Automatic
The message flow is under transaction control only if the incoming message is marked persistent. This remains true for messages derived from this input message and output by an MQOutput node, unless the MQOutput node explicitly overrides the transaction status.

Yes
The message flow is under transaction control. This remains true for messages derived from this input message and output by an MQOutput node, unless the MQOutput node explicitly overrides the transaction status. This is the default.

No
The message flow is not under transaction control. This remains true for messages derived from this input message and output by an MQOutput node, unless the MQOutput node explicitly overrides the transaction status.

Order Mode
Determines the order in which messages are retrieved from the queue, and processed. Valid values are:

Default
Messages are retrieved from the queue in the order defined by the queue attributes, but are processed in random order by the one or more instances of the message flow in which this node appears. If more than one instance of the message flow is active, the order in which message processing completes is unpredictable.

By UserId
Messages that are associated with a specific user ID are retrieved in the order defined by the queue attributes, and that order is guaranteed to be retained such that these messages reach the output node or nodes in the order they are retrieved by the input node. That is, only one message for a given user is processed concurrently.

By Queue Order
Messages are retrieved in the order defined by the queue attributes. and they are propagated to target nodes in the order in which they are retrieved: therefore message order is guaranteed.

Logical Order
Determines whether messages are received in logical order, as defined by MQSeries. Valid values are:

yes
Messages that are part of a message group are received in the correct order as assigned by the sending application. This is the default value.

no
Messages sent as part of a group are not received in a predetermined order. If a broker expects to receive messages in groups when Logical Order is set to no, either the order of the input messages must not be significant or the message flow must be designed to process them appropriately.

All Messages Available
Specifies whether all messages in a group need to be available before retrieval of a message is possible. By default, this value is not selected.

Match Message ID
Specifies whether the MQInput node receives only those messages with a matching message identifier value, as set in the MsgId field of the MQMD. If the supplied hexadecimal string is shorter than the MsgId field, it is assumed to be padded to the right with `X'00'` characters.

By default, this value is not selected.

Match Correlation ID
Specifies whether the MQInput node receives only those messages with a matching correlation identifier value, as set in the CorrelId field of the MQMD. If the supplied hexadecimal string is shorter than the CorrelId field, it is assumed to be padded to the right with `X'00'` characters.

By default, this value is not selected.

Convert
Determines whether MQSeries performs data conversion on the message, in conformance with the CodedCharSetId and Encoding values. If this value is selected, the values of the properties Convert Encoding and Convert Coded Character Set ID are used to update the MQMD to cause the required conversion to take place.

By default this value is not selected.

Convert Encoding
Specifies the representation used for numeric values in the message data, expressed as an integer value. This property is relevant only when the Convert value has been selected.

Convert Coded Character Set ID
Specifies the coded character set identifier of character data in the message data, expressed as an integer value. This property is relevant only when the Convert value has been selected.

By default, this value is not selected.

Commit By Message Group
Specifies at what point a transaction is committed when processing messages that are part of a message group. If this value is selected, a commit is performed only after the final message in the group has been received. Note that the Logical Order value must also be selected in this case. If this value is not selected, a commit is performed after each message has been routed completely through the message flow.

By default, this value is not selected.

Validate
Specifies whether the message parsers should use any validation capabilities they have.

By default, this value is not selected.

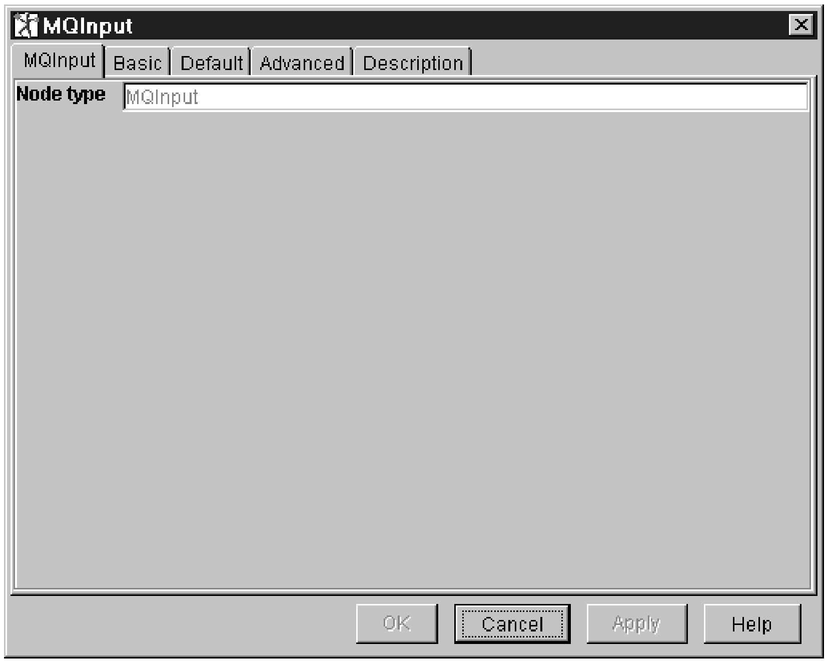
Configuring an MQInput node

For a description of the properties of the MQInput node and their possible values, see "MQInput node properties" on page 136.

To configure an MQInput node:

1. In the Message Flow Definition pane, right click the MQInput node you want to configure and click **Properties**.

   The **MQInput** dialog is displayed, as shown in Figure 36.



Node type MQInput

Figure 36. The MQInput dialog

2. In the **MQInput** dialog, click the **Basic** tab. Type a value in the **Queue Name** field.

3. In the **MQInput** dialog, click the **Default** tab.

   If you wish, type values in the fields:

   - Message Domain
   - Message Set
   - Message Type
   - Message Format
   - Topic

   If the incoming message does not contain an MQRFH or MQRFH2 header, these values constitute a default message template and enable MQSeries Integrator to parse the message.

4. In the **MQInput** dialog, click the **Advanced** tab.

   Supply values for the following fields to suit your processing requirements:

   - Transaction Mode
   - Order Mode
   - Logical Order
   - All Messages Available
   - Match Message Id
   - Match CorrelId
   - Convert
   - Convert Encoding
   - Convert Coded Character Set ID
   - Commit by Message Group
   - Validate

5. If you want to provide a description of this instance of the MQInput node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **MQInput** dialog. Type a short description, or a long description, or both.

6. Click **OK** to finish configuring this MQInput node.

| For an example of configuring an MQInput node within a message flow, see the
| retail scenario described in Appendix A, An example scenario (section “Getting the
| message” on page 267).

MQOutput node

Description

The MQOutput node uses MQPUT to write messages to an MQSeries message queue defined on any queue manager accessible by the broker's queue manager, or to the destinations identified in the destination list associated with the message.

MQOutput node terminals

Table 15 describes the terminals of the MQOutput node.

Table 15. MQOutput node terminals

Terminal	Description
in	The terminal that accepts a message for processing by the node.
failure	The output terminal to which the message is routed if an MQPUT call returns an error.

MQOutput node properties

These properties are displayed when you right click an MQOutput node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Queue Manager Name
Is the name of the MQSeries queue manager to which the output queue is defined. The maximum length of a queue manager name is 48 characters.

Queue Name
Is the name of the MQSeries queue to which the message is written if Destination Mode is fixed. The maximum length of a queue name is 48 characters.

Destination Mode
Identifies the queue to which the message will be sent. Valid values are:

Queue Name
The message is sent to the queue named in the Queue Name property. This is the default value.

replyToQueue
The message is sent to the queue named in the replyToQueue field in the MQMD.

Destination List
The message is sent to the queues named in the destination list.

Transaction Mode
Specifies whether the message will be put as part of a transaction.

Valid values are:

Automatic
The message transactionality is as it was specified at the MQInput node. This is the default value.

Yes
The message is put transactionally.

No
The message is not put transactionally.

Persistence Mode
Specifies if the message will be put persistently.

Valid values are:

Automatic
The persistence is as specified in the incoming message. This is the default value.

Yes
The message is put persistently.

No
The message is not put persistently.

As Defined for Queue
The message persistence is as defined for the MQSeries queue to which the message is written.

| **New Message ID**
Specifies whether MQSeries generates a new message identifier to replace the contents of the MsgId field in the MQMD.

By default, this value is not selected.

| **New Correlation ID**
Specifies whether MQSeries generates a new correlation identifier to replace the contents of the CorrelId field in the MQMD.

By default, this value is not selected.

Segmentation Allowed
Specifies whether MQSeries can, if appropriate, break the message into segments.

By default, this value is not selected.

Message Context
Valid values are:

| • Pass All
| • Pass Identity
| • Set All
| • Set Identity
| • Default
| • None

Its default value is passAll.

Alternate User Authority

By default, this value is not selected.

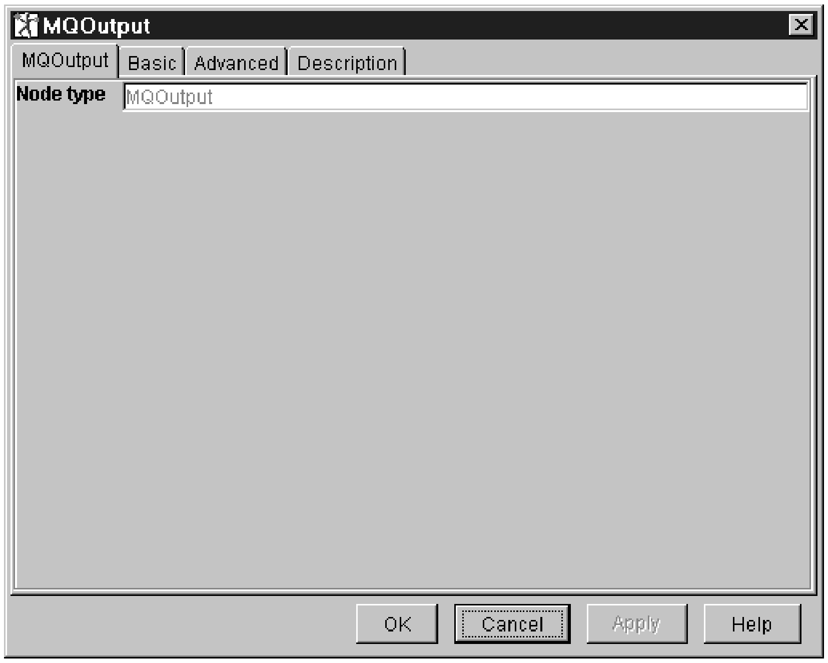
Configuring an MQOutput node

For a description of the properties of the MQOutput node and their possible values, see “MQOutput node properties” on page 141.

To configure an MQOutput node:

1. In the Message Flow Definition pane, right click the MQOutput node you want to configure and click **Properties**.

   The **MQOutput** dialog is displayed, as shown in Figure 37.



Node type MQOutput

Figure 37. The MQOutput dialog

2. In the **MQOutput** dialog, click the **Basic** tab. Type a value in the **Queue Manager Name** field and in the **Queue Name** field. These values are required if Destination Mode is fixed.
3. In the **MQOutput** dialog, click the **Advanced** tab.

   Supply values for the following fields to suit your processing requirements:

   - Destination Mode
   - Transaction Mode
   - Persistence Mode
   - New MessageID
   - New Correl ID
   - Segmentation Allowed
   - Message Context
   - Alternate User Authority
4. If you want to provide a description of this instance of the MQOutput node (which is recommended if you want other Control Center users to be able to

make use of it), click the **Description** tab of the **MQOutput** dialog. Type a short description, or a long description, or both.

5. Click **OK** to finish configuring this MQOutput node.

MQReply node

Description

The MQReply node is a specialized form of the MQOutput node that sends a response to the originator of the message by putting a message to the MQSeries queue identified by the ReplyToQ field of the message header.

MQReply node terminals

Table 16 describes the terminals of the MQReply node.

Table 16. MQReply node terminals

Terminal	Description
in	The terminal that accepts a message for processing by the node.
failure	The output terminal to which the message is routed if a failure is detected. For example, if the MQPUT operation to the reply-to queue fails.

MQReply node properties

These properties are displayed when you right click an MQReply node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Segmentation Allowed
Specifies whether MQSeries can, if appropriate, break the message into segments.

By default, this value is not selected.

Persistence Mode
Specifies whether the message will be put persistently.

Valid values are:

Automatic
Persistence is as specified in the incoming message. This is the default value.

Yes
The message is put persistently.

No
The message is not put persistently.

As Defined for Queue
The message persistence is as defined for the MQSeries queue to which the message is put.

Transaction Mode
Specifies whether the message will be put as part of a transaction.

Valid values are:

Automatic
The message transactionality is as it was specified at the MQInput node. This is the default value.

Yes
The message is put transactionally.

No
The message is not put transactionally.

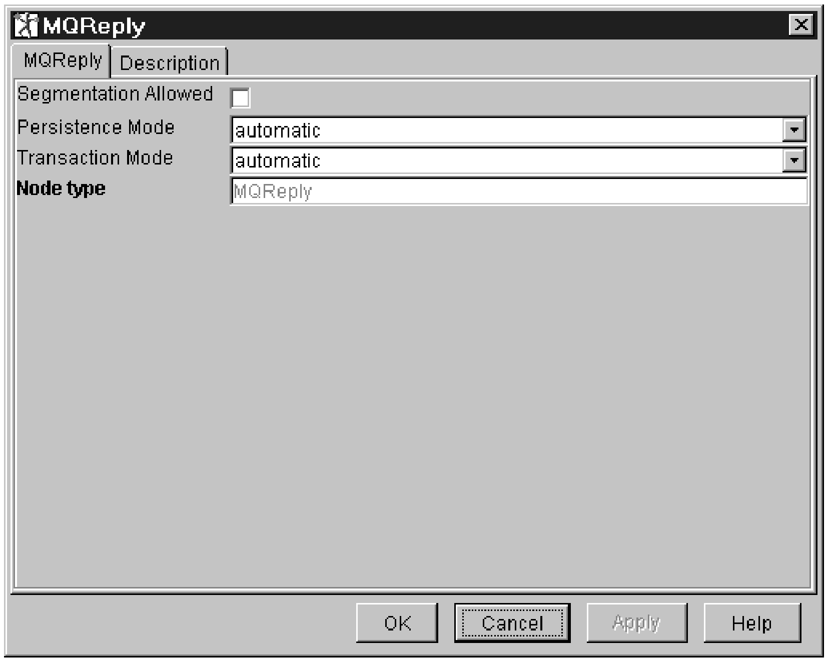
Configuring an MQReply node

For a description of the properties of the MQReply node and their possible values, see "MQReply node properties" on page 145.

To configure an MQReply node:

1. In the Message Flow Definition pane, right click the MQReply node you want to configure and click **Properties**.

   The **MQReply** dialog is displayed, as shown in Figure 38.



Segmentation Allowed ☐
Persistence Mode: automatic
Transaction Mode: automatic
Node type: MQReply

Figure 38. The MQReply dialog

2. In the **MQReply** dialog, supply values for the following fields to suit your processing requirements:
   - Segmentation Allowed
   - Persistence Mode
   - Transaction Mode

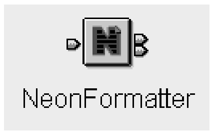
3. If you want to provide a description of this instance of the MQReply node (which is recommended if you want other Control Center users to be able to

make use of it), click the **Description** tab of the **MQReply** dialog. Type a short description, or a long description, or both.

4. Click **OK** to finish configuring this MQReply node.

NEONFormatter node

Description



The NEONFormatter node is used transform a message from a known input format to a specified output format. The message definition and transformations are defined using the NEON Formatter graphical utility, not the MQSeries Integrator Control Center.

NEON**Formatter node terminals**

Table 17 describes the terminals of the NEONFormatter node.

Table 17. NEONFormatter node terminals

Terminal	**Description**
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the transformed message is routed.
failure	The output terminal to which the untransformed message is routed if a failure is detected during the reformatting.

NEONFormatter node properties

These properties are displayed when you right click a NEONFormatter node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Target Format
Is the format to which the information being passed through the node will be transformed.

The remaining properties of the NEONFormatter node define the message properties used to parse the message after it has been transformed. They have no effect on the reformatting performed by this node.

Output Domain
Is the message domain.

Output Set
Is the message set.

Output Type
Is the message type.

Output Wire Format
Is the output wire format.

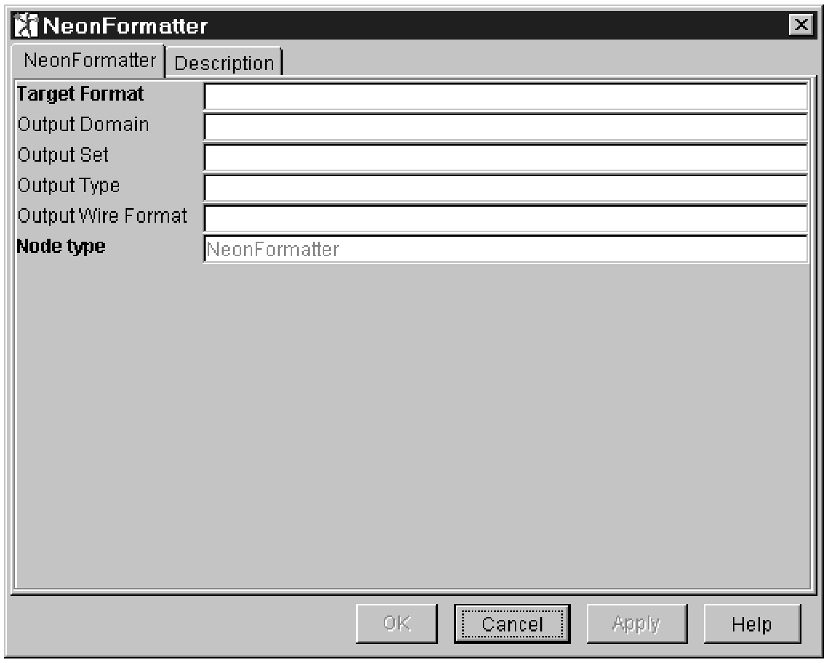
Configuring a NEONFormatter node

For a description of the properties of the NEONFormatter node and their possible values, see “NEONFormatter node properties” on page 148.

To configure a NEONFormatter node:

1. In the Message Flow Definition pane, right click the NEONFormatter node you want to configure and click **Properties**.

   The **NEONFormatter** dialog is displayed, as shown in Figure 39.



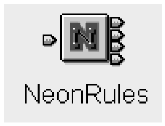
Target Format	
Output Domain	
Output Set	
Output Type	
Output Wire Format	
Node type	NeonFormatter

Figure 39. The NEONFormatter dialog

2. In the **NEONFormatter** dialog, supply values for the following fields to suit your processing requirements:
   - Target Format (a required value)
   - Output Domain
   - Output Set
   - Output Type
   - Output Wire Format
3. If you want to provide a description of this instance of the NEONFormatter node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **NEONFormatter** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this NEONFormatter node.

NEONRules node

Description



The NEON Rules node provides an encapsulation of the NEON Rules engine within the MQSeries Integrator Version 2 environment. The firing of the Propagate action propagates the output message to the propagate terminal. The firing of the PutQueue action attaches a queue name to the destination list associated with the message and routes the message to the putqueue terminal.

NEONRules node terminals

Table 18 describes the terminals of the NEONRules node.

Table 18. NEONRules node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
propagate	The output terminal to which the message is routed when the propagate action is executed in the rules engine.
failure	The output terminal to which the message is routed if a failure is detected during the extraction.
putqueue	The output terminal to which the message is routed if the queue name given under the Putqueue action is added to the message's destination list.
noHit	The output terminal to which the input message is routed if no rule is triggered.

NEONRules node properties

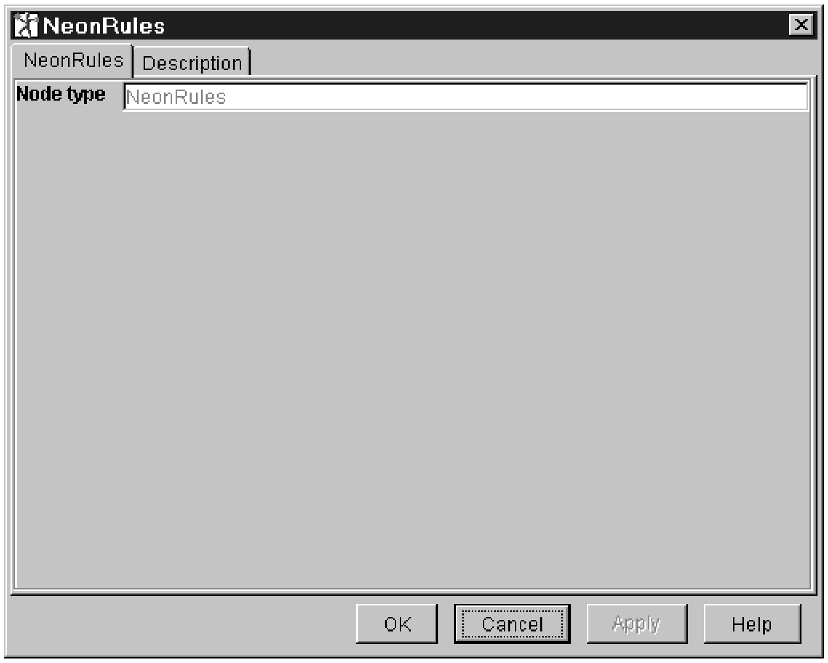
The NEONRules node has no properties.

Configuring a NEONRules node

The NEONRules node has no configurable properties. However, you can provide a description of this instance of the node, as follows:

1. In the Message Flow Definition pane, right click the NEONRules node for which you want to provide a description and click **Properties**.

   The **NEONRules** dialog is displayed, as shown in Figure 40 on page 151.



Node type NeonRules

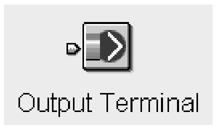
Figure 40. The NEONRules dialog

2. In the **NEONRules** dialog, click the **Description** tab. Type a short description, or a long description, or both.
3. Click **OK** to finish configuring this NEONRules node.

The database connection details for this node are obtained from the file named on the environment variable MQSI_PARAMETERS_FILE. For more information, see Appendix E, "NEON Rules and Formatter" on page 397.

Output Terminal

Description



The Output Terminal provides an out terminal for an embedded message flow.

An Output Terminal can only receive connections from a message flow node. The Output Terminal is the last node of a subflow that you can embed in another flow. A subflow can be set up with more than one Output Terminal. When the subflow is included in an embedding flow, the number of out terminals displayed represents the number of Output Terminals in the embedded subflow. Each out connector is labelled with the name you assigned to the Output Terminal node. For more information about embedded flows, see "Including one message flow in another" on page 97.

Output Terminal node terminals

Table 19 describes the terminals of the Output Terminal.

Table 19. Output Terminal node terminals

Terminal	Description
in	The terminal that accepts the message for processing by the node.

Output Terminal properties

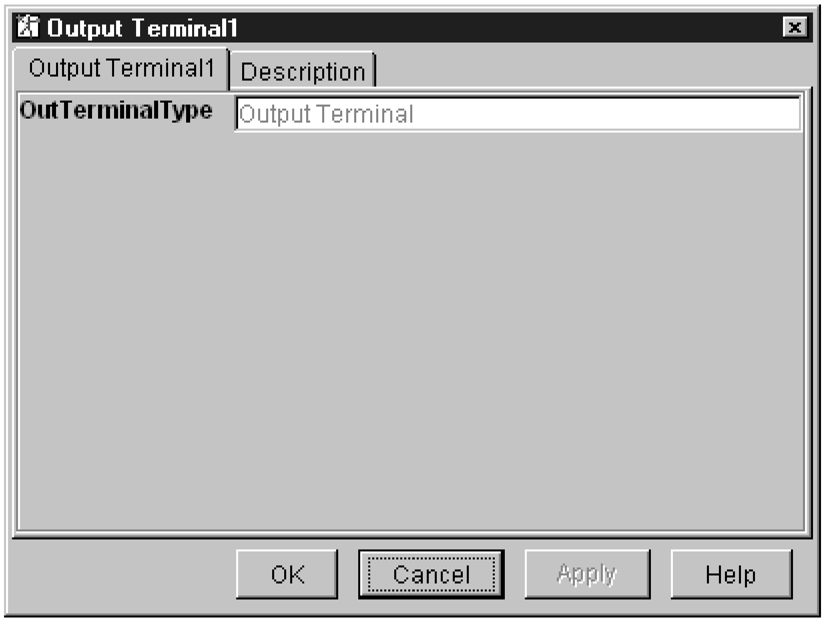
The Output Terminal has no properties.

Configuring the Output Terminal

The Output Terminal has no configurable properties. However, you can provide a description of this instance of the node, as follows:

1. In the Message Flow Definition pane, right click the Output Terminal for which you want to provide a description and click **Properties**.

   The **Output Terminal** dialog is displayed, as shown in Figure 41 on page 153.



OutTerminalType: Output Terminal

Figure 41. The Output Terminal dialog

2. In the **Output Terminal** dialog, click the **Description** tab. Type a short description, or a long description, or both.
3. Click **OK** to finish configuring this Output Terminal.

Publication node

Description

The Publication node filters and transmits the output from a message flow to subscribers who have registered an interest in a particular set of topics.

The Publication node must always be an output node of a message flow and has no output terminals of its own.

Publication node terminals

Table 20 describes the terminals of the Publication node.

Table 20. Publication node terminals

Terminal	Description
in	The terminal that accepts a message for processing by the node.

Publication node properties

These properties are displayed when you right click a Publication node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Implicit Stream Naming
If this value is selected, the name of the MQSeries queue on which the message is received is also the stream name. This feature is provided for compatibility with MQSeries base publish/subscribe function, and applies to messages with an MQRFH header when MQPSStream is not specified.

By default, this value is not selected.

Subscription Point
The subscription point value for the node. If the property is not specified, the default subscription point is assumed.

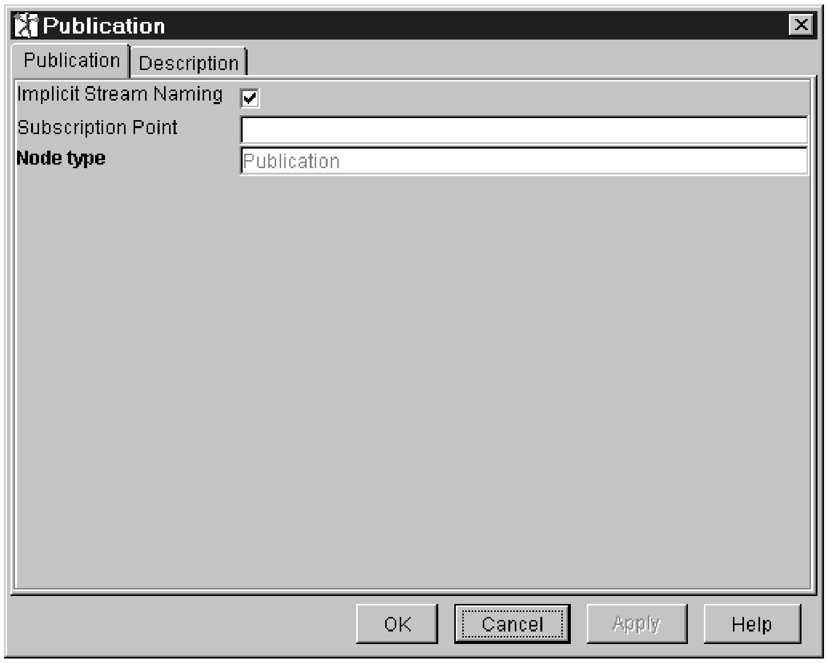
Configuring the Publication node

For a description of the properties of the Publication node and their possible values, see “Publication node properties.”

To configure a Publication node:

1. In the Message Flow Definition pane, right click the Publication node you want to configure and click **Properties**.

   The **Publication** dialog is displayed, as shown in Figure 42 on page 155.



Implicit Stream Naming ✓
Subscription Point
Node type Publication

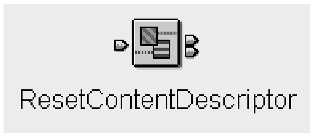
Figure 42. The Publication dialog

2. In the **Publication** dialog, supply values for the following fields to suit your processing requirements:
   - Implicit Stream Naming
   - Subscription Point
3. If you want to provide a description of this instance of the Publication node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Publication** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this Publication node.

See the *MQSeries Integrator Programming Guide* for information about publish/subscribe implementation.

ResetContentDescriptor node

Description



The ResetContentDescriptor node takes the bit stream of the input message and reparses it using a different message template from the same or a different message dictionary. The node can reset any combination of message domain, set, type, and format.

ResetContentDescriptor node terminals

Table 21 describes the terminals of the ResetContentDescriptor node.

Table 21. ResetContentDescriptor node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the message is routed when it is successfully reparsed by the specified parser.
failure	The output terminal to which the message is routed if it is not successfully reparsed.

ResetContentDescriptor node properties

These properties are displayed when you right click a ResetContentDescriptor node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Message Domain
Is the new message domain.

| For further information about supported message domains, see “Message
| domains” on page 30.

| **Reset Message Domain**
| If this value is selected, the message domain is reset.

| By default, this value is not selected.

Message Set
Is the new message set.

Reset Message Set
If this value is selected, the message set is reset.

By default, this value is not selected.

Message Type
Is the new message type.

Reset Message Type
If this value is selected, the message type is reset.

By default, this value is not selected.

Message Format
Is the new message format.

Reset Message Format
If this value is selected, the message format is reset.

By default, this value is not selected.

These properties set the domain, set, type, and format in the message header of the message passing through the ResetContentDescriptor node. However, this will only happen if suitable headers already exist. If the message does not have an MQRFH or MQRFH2 header, the node does not create one.

When you exit the ResetContentDescriptor node properties, the Standard Properties are set to reflect the new values specified by this node. The parse tree available to all the nodes placed further along the message flow is also made consistent with these values.

Configuring the ResetContentDescriptor node

For a description of the properties of the ResetContentDescriptor node and their possible values, see "ResetContentDescriptor node properties" on page 156.

To configure a ResetContentDescriptor node:

1. In the Message Flow Definition pane, right click the ResetContentDescriptor node you want to configure and click **Properties**.

   The **ResetContentDescriptor** dialog is displayed, as shown in Figure 43.

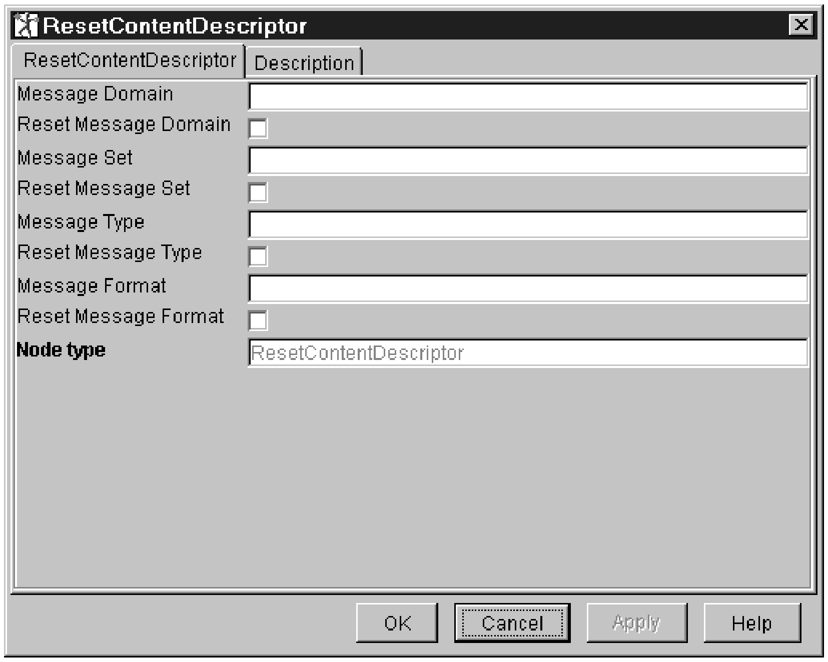


Figure 43. The ResetContentDescriptor dialog

2. In the **ResetContentDescriptor** dialog, supply values for the following fields to suit your processing requirements:

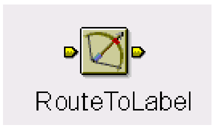
   - Message Domain
   - Reset Message Domain
   - Message Set
   - Reset Message Set
   - Message Type

- Reset Message Type
- Message Format
- Reset Message Format

3. If you want to provide a description of this instance of the ResetContentDescriptor node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **ResetContentDescriptor** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this ResetContentDescriptor node.

RouteToLabel node

Description



The RouteToLabel node provides a dynamic routing facility based on the contents of the destination list associated with the message. The destination list contains the identity of one or more target Label nodes, identified by their **Label Name** property (not the node name).

The destination, defined by the **Label Name** of a Label node, is resolved by the broker itself during message flow processing: you do not connect a terminal on the RouteToLabel node to the destination nodes.

RouteToLabel node terminals

Table 22 describes the terminals of the RouteToLabel node.

Table 22. RouteToLabel node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
failure	The terminal to which the message is routed if the destination label is not valid.

RouteToLabel node properties

These properties are displayed when you right click a RouteToLabel node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Mode
Specifies how the labels in the destination list within the message are processed.

Valid values are:

Route To First
The first entry in the destination list is removed, and processing is transferred to the Label node that has property Label Name set to the value of the entry that has been removed.

Route To Last
The last entry in the destination list is removed, and processing is transferred to the Label node that has property Label Name set to the value of the entry that has been removed. This is the default.

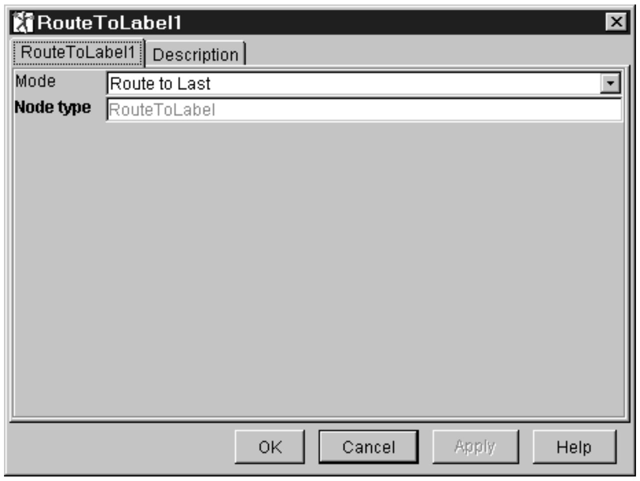
Configuring a RouteToLabel node

For a description of the properties of the RouteToLabel node and their possible values, see “RouteToLabel node properties.”

To configure a RouteToLabel node:

1. In the Message Flow Definition pane, right click the RouteToLabel node you want to configure and click **Properties**.

   The **RouteToLabel** dialog is displayed, as shown in Figure 44 on page 160.



Mode: Route to Last
Node type: RouteToLabel

Figure 44. The RouteToLabel dialog

2. In the **RouteToLabel** dialog, select the method of operation you require for this node.
3. If you want to provide a description of this instance of the RouteToLabel node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **RouteToLabel** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this RouteToLabel node.

Using a RouteToLabel node

A RouteToLabel node uses a destination list within a message to route the message to a target node of type Label that matches the label within the destination list item. Therefore the message must include a destination list to be acted on by the RouteToLabel node.

You must create the destination list, and include it in the message, in a compute node. You must select the *Advanced* tab on the Compute node properties dialog, and select an option that includes *Destination* from the drop-down list.

The destinations are set up as a list of label names. The label names can be any string value, and can be explicitly specified in the compute node or taken or cast from any field in the message or from a database. A label name in the destination list must, however, match the Label Name property of a corresponding Label Node.

Figure 45 on page 161 illustrates a message flow that uses these techniques to achieve dynamic routing based on message content. The flow is made up of subflows that are associated with the main flow using RouteToLabel and Label nodes.

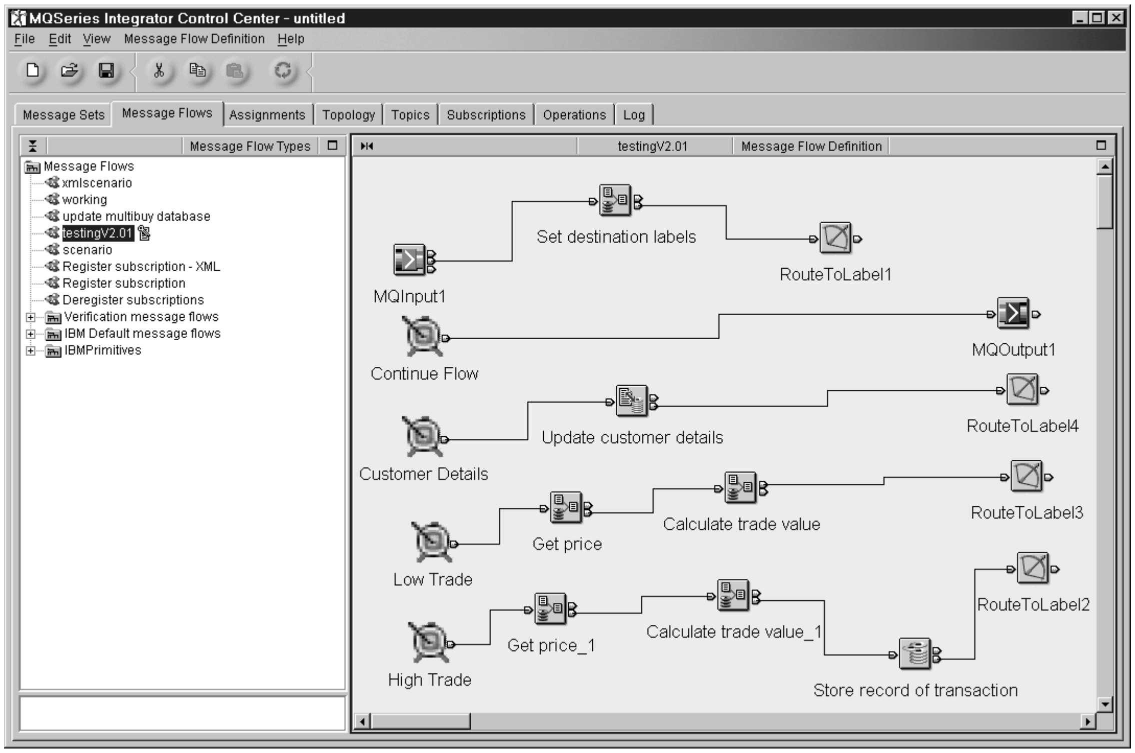


Figure 45. A message flow with RouteToLabel and Label nodes. This shows the nodes connected 'Manhattan Style' (you can select this style of connecting the nodes by right clicking on the pane background).

The message flow shows how you might handle brokerage transactions if you want to process high-value trading requests, low-value trading requests and requests to updates customer details differently.

The use of RouteToLabel and Label nodes makes a simpler message flow than the message flow you would need if you used a sequence of Filter nodes that identify and route the message for different processing, or a sequence of nodes that each performs an action on a subset of the total number of messages processed by the message flow.

Each message in this brokerage example has a request field that indicates whether the message contains "hightrade", "lowtrade", or "custdetails" information. Each type is routed to a different sequence of nodes before being completed by a common flow.

You configure the compute node "Set destination labels" to create a destination list in the message by entering the following ESQL:

```
SET OutputDestinationList.Destination.RouterList.DestinationData[1].labelname
 = 'continue';
SET OutputDestinationList.Destination.RouterList.DestinationData[2].labelname
 = "InputBody.MRM.trademsg.request";
```

If you set Route To Last on the RouteToLabel node, a message is routed to the last label in the destination list. In this example, that is the label that matches the value of the "request" field in the message. Therefore a message with a value of "hightrade" in the request field is routed to the Label node with a Label Name property of "hightrade".

If the message fragment performing the dynamically routed work itself ends in a RouteToLabel node, the message is passed to the next destination in the list. In the example above, the message is passed to the Label node with a Label Name property of "continue", and continues along the common part of the message flow.

There are four message flow fragments, beginning with a Label node:

1. Continue (Label Name property = continue). This fragment does not end in a RouteToLabel node. It might end in an MQOutput node to complete the message flow execution or it might continue with work that is not required to be dynamically routed. Typically, the Label Name of this fragment is the one at index [1] of the destination list. This means that, when all the dynamically routed work is complete, the flow either finishes or continues with common processing.
2. High Trade (Label Name property = hightrade). This fragment performs processing specific to high-value trading requests, for example storing records of the trade in a database and performing credit authorizations. This fragment ends in a RouteToLabel node to send the message on to the next destination in the list.
3. Low Trade (label name property = lowtrade). This fragment performs processing specific to low-value trading requests. It ends in a RouteToLabel node to send the message on to the next destination in the list.
4. Customer Details (Label Name property = custdetails). This performs processing specific to requests to update customer details, for example, updating a customer details database. This fragment ends in a RouteToLabel node to send the message on to the next destination in the list.

The flexibility of this dynamic routing facility enables an infinite number of variations on the above scenario.

You must ensure that you include all related Label nodes and the subflows that follow from the Label nodes within the Message Flow Definition pane of the embedding message flow, or its surrounding subflow. This ensures that all the Label nodes are included when you deploy the message flow. The Label node is not connected to a prior node: if you create a subflow that starts with any other type of node, the subflow defined in the Message Flow Definition pane is ignored when the message flow is deployed. Subflows that start with a Label node are not ignored.

For more information about destination lists, see “Exception and destination list structure” on page 373. If you intend to derive destination values from the message itself, or from a database, you might also need to cast values from one type to another. Casts are described in more detail in “CAST expressions” on page 342.

Throw node

Description

The Throw node provides a mechanism for throwing an exception within a message flow. The exception might be caught and processed by a preceding TryCatch node within the message flow, or handled by the MQInput node.

Throw node terminals

Table 23 describes the terminals of the Throw node.

Table 23. Throw node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.

Throw node properties

These properties are displayed when you right click a Throw node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Message Catalog
The name of the message catalog in which the error text that explains the error number of the exception is to be found.

Message Number
The error number of the exception being thrown. The numbers 3000-3019 are reserved in the MQSeries Integrator catalog for this use but in principle any number can be used.

Message Text
Text giving the cause of the error. This can be different from that associated with the message number in the message catalog.

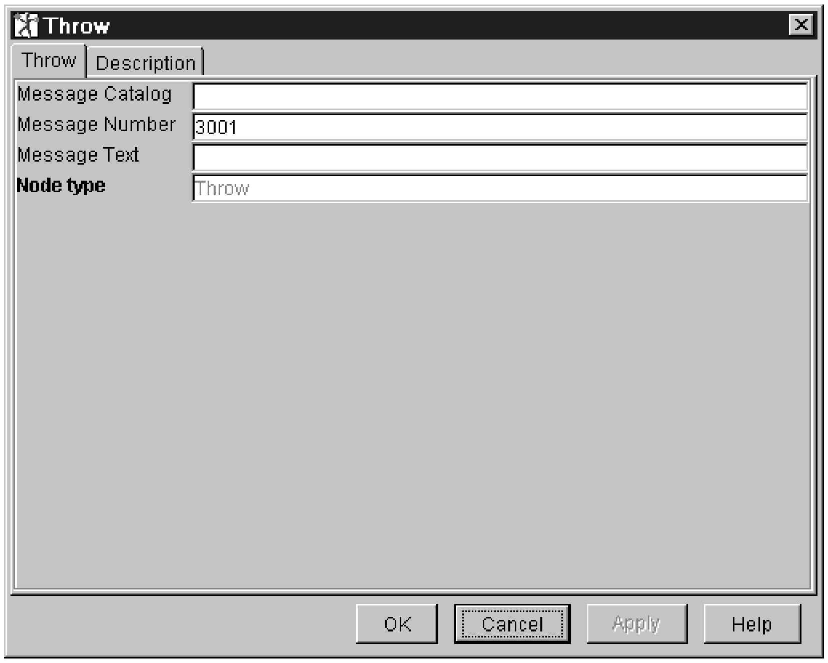
Configuring a Throw node

For a description of the properties of the Throw node and their possible values, see "Throw node properties."

To configure a Throw node:

1. In the Message Flow Definition pane, right click the Throw node you want to configure and click **Properties**.

   The **Throw** dialog is displayed, as shown in Figure 46 on page 164.



Message Catalog
Message Number 3001
Message Text
Node type Throw

Figure 46. The Throw dialog

2. In the **Throw** dialog, supply values for the following fields to suit your processing requirements. The values you specify define the contents of the exception thrown by the Throw node.
   - Message Catalog
   - Message Number
   - Message Text
3. If you want to provide a description of this instance of the Throw node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Throw** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this Throw node.

Trace node

Description

The Trace node generates trace records that can incorporate text, message content, and date and time information, to help you to monitor the behavior of the message flow.

| The operation of the Trace node is independent of the setting of user tracing for the message flow in which it is included: output from the trace node is written even if user tracing is set off. In particular, if user tracing is set to `None`, and a Trace node in that message flow has its Destination property set to `User Trace`, the entries written by the Trace node are recorded in the user trace log.

Trace node terminals

Table 24 describes the terminals of the Trace node.

Table 24. Trace node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed, even if a failure occurs while the message is in the trace node.

Trace node properties

These properties are displayed when you right click a Trace node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Destination
Specifies the destination of the trace record.

Valid values are:

None
No trace record is produced.

User Trace
The trace record is written to the userTrace log. This is the default value.

File
The trace record is written to the file specified in the File Path property.

File Path
Is the fully qualified name of the file to which trace records are to be written.

Pattern
Defines the format of the trace record to be produced. The pattern that you enter is passed to the trace destination, and any expression of the form ${...} is resolved by evaluating the ESQL expression between the braces. Any ESQL expression that is valid on the right-hand side of a compute statement can be used.

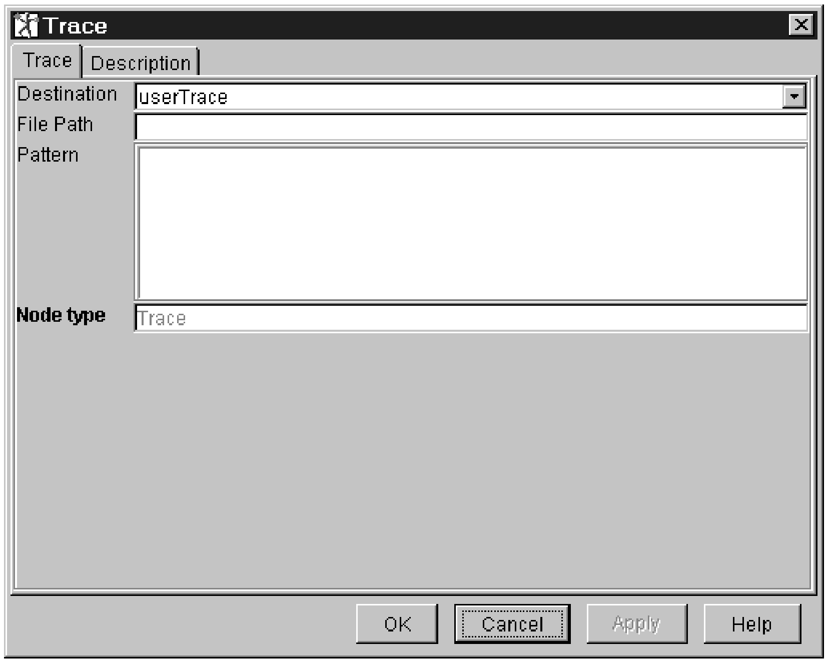
Configuring the Trace node

For a description of the properties of the Trace node and their possible values, see "Trace node properties" on page 165.

To configure a Trace node:

1. In the Message Flow Definition pane, right click the Trace node you want to configure and click **Properties**.

   The **Trace** dialog is displayed, as shown in Figure 47.



Destination: userTrace
File Path:
Pattern:
Node type: Trace

Figure 47. The Trace dialog

2. In the **Trace** dialog, supply values for the following fields to suit your processing requirements:
   - Destination
   - File Path
   - Pattern
3. If you want to provide a description of this instance of the Trace node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Trace** dialog. Type a short description, or a long description, or both.
4. Click **OK** to finish configuring this Trace node.

| For an example of configuring a Trace node within a message flow, see the retail
| scenario described in Appendix A, An example scenario (section "Writing a trace
| entry" on page 272).

Using a Trace node

An example of the trace output is below:

```
(0x1000010)XML       = (
  (0x5000018)XML       = (
    (0x6000011) = '1.0'
  )
  (0x1000000)MESSAGETAG = (
    (0x3000000)highlevel   = '1'
    (0x3000000)higherlevel = '21'
    (0x1000000)m_m1        = (
      (0x3000000)level2     = '2'
      (0x1000000)e_string01 = (
        (0x3000000)level3 = '3'
        (0x2000000)       = 'IAPMDI27 ddddddd'
```

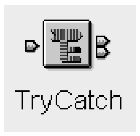
This can be interpreted as follows:

1. The numbers in brackets at the left hand end of each line are element types. These numbers, which do not display leading zeros, are defined as follows:
   - Bits 0 to 7 identify the specific element type. See the parser descriptions in Appendix D, “MQSeries message header parsers” on page 381 for details about element types.
   - Bits 8 to 23. These bits are not used.
   - Bits 24 to 31 identify the generic element type:
     – Bit 24 indicates that the element has a name.
     – Bit 25 indicates that the element has a value.
     – Bit 26 indicates that the element is meaningful only to its parser type (therefore it is not copied to others).
2. The names after the brackets are element names.
3. The values after the equal sign are the element values. These values are displayed as ESQL literals. Where a node contains other nodes this is indicated by the nesting.

For further usage information, see “Writing a trace entry” on page 272 in Appendix A, “An example scenario” on page 257.

TryCatch node

Description



The TryCatch node provides a special handler for exception processing. The input message is initially routed on the try terminal of this node. If an exception is subsequently thrown by a downstream node, it is caught by this node, which then routes the original message to its catch terminal. If the TryCatch node catch terminal is not connected, the message is discarded.

TryCatch node terminals

Table 25 describes the terminals of the TryCatch node.

Table 25. TryCatch node terminals

Terminal	**Description**
in	The input terminal that accepts a message for processing by the node.
try	The output terminal to which the original message is routed.
catch	The output terminal to which the message is routed if an exception is thrown downstream and then caught by the node (that is, it was not caught by another TryCatch node further downstream). If the catch terminal is unconnected, the message is lost.

TryCatch node properties

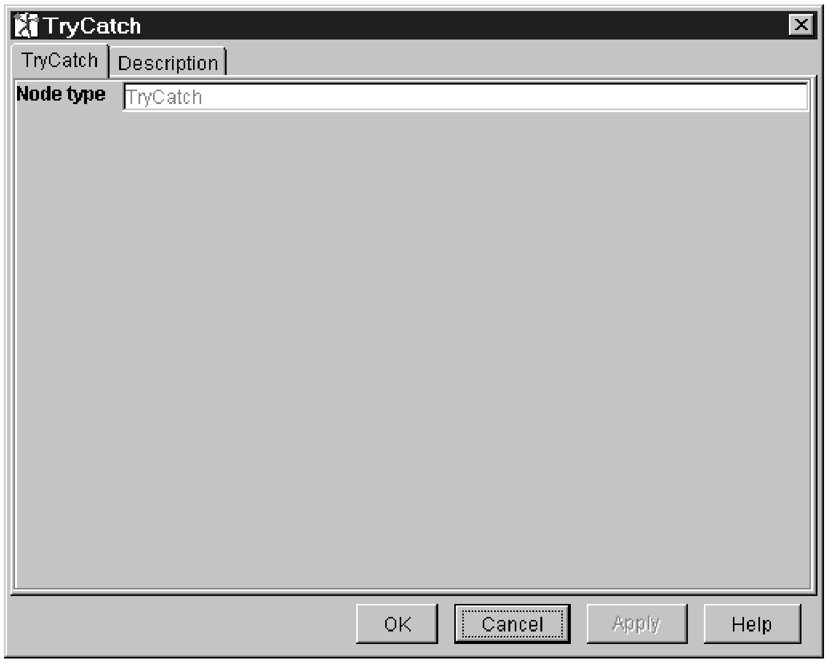
The TryCatch node has no properties.

Configuring the TryCatch node

The TryCatch node has no configurable properties. However, you can provide a description of this instance of the node, as follows:

1. In the Message Flow Definition pane, right click the TryCatch node for which you want to provide a description and click **Properties**.

   The **TryCatch** dialog is displayed, as shown in Figure 48 on page 169.



Node type TryCatch

Figure 48. The TryCatch dialog

2. In the **TryCatch** dialog, click the **Description** tab. Type a short description, or a long description, or both.
3. Click **OK** to finish configuring this TryCatch node.

Warehouse node

Description

The Warehouse node is a specialized form of the Database node that stores the entire message, or parts of the message, or both, to the specified ODBC data source.

Warehouse node terminals

Table 26 describes the terminals of the Warehouse node.

Table 26. Warehouse node terminals

Terminal	Description
in	The input terminal that accepts a message for processing by the node.
out	The output terminal to which the original message is routed when processing completes successfully.
failure	The output terminal to which the message is routed if a failure is detected during processing. For example, if the connection to the database fails, or the table specified is invalid.

You can use a message warehouse:

- To maintain an audit trail of messages
- For offline or batch processing of messages (a process sometimes referred to as *data mining*)
- To enable subsequent reprocessing of selected messages

Once stored in the message warehouse, messages can be retrieved using standard database query and data mining techniques. No explicit support for these functions is supplied by MQSeries Integrator.

You can choose to store in the message warehouse:

- The entire message
- Selected parts of the message

Storing the entire message

When you store the entire message in a message warehouse, it is stored as a binary object. You can choose to store a timestamp for the message, though this is optional. Any timestamp is stored in a separate column from the message itself.

The advantages of storing the entire message are:

- You do not have to have decided how you will use the data before you store it.
- You do not have to have defined a database schema for every type of message that could pass through the broker.

However, you could consider preceding each Warehouse node with a Compute node that would convert each message to a common schema before the Warehouse node stores it.

Storing parts of the message

If you store selected parts of a message, with a timestamp if required, you must define a database schema for that message type. The message is mapped to true type: for example, a character string in a message is stored as a character string.

Warehouse node properties

These properties are displayed when you right click a Warehouse node entry in the Message Flow Types pane, and click **Properties**. The values displayed are the default properties for this instance of the node. They cannot be edited when displayed from the Message Flow Types pane.

Transaction
The Transaction value specifies whether the action performed by this node is to be viewed as part of a larger transaction, or managed independently of the work performed by other nodes in the message flow.

Valid values are:

Automatic
The decision to commit or roll back the Warehouse node action depends on the success or failure of the message flow to which it belongs. This is the default value.

Commit
The action of the Warehouse node is to be committed, irrespective of the success or failure of the message flow as a whole.

Field Mapping
Is a list of assignment statements mapping message content into database fields.

Data Source
Is the name of the database to be used as the warehouse.

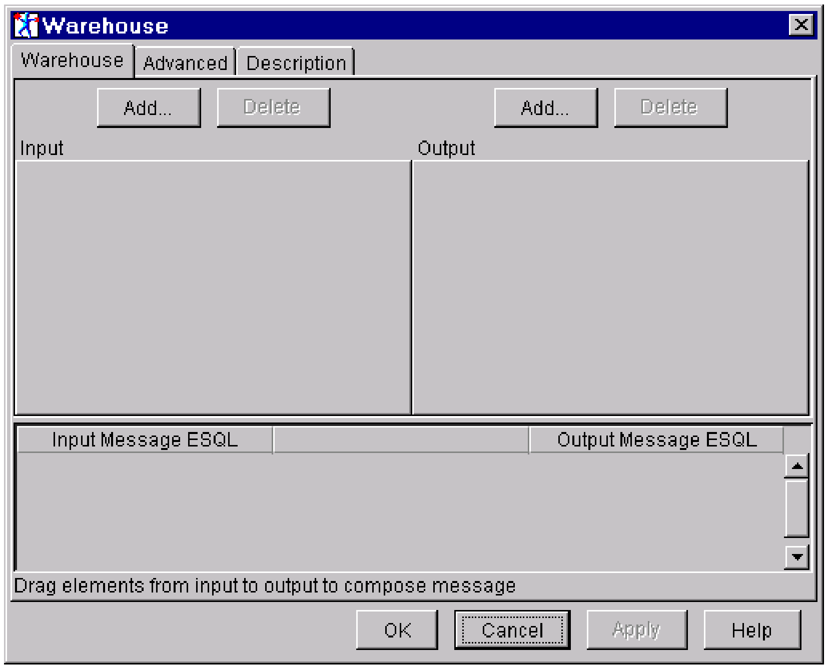
Treat warnings as errors
Specifies whether warning messages generated during this node's processing are to be treated as errors, causing the message to be routed the failure terminal.

Configuring the Warehouse node to store the entire message

For a description of the properties of the Warehouse node and their possible values, see "Warehouse node properties."

1. In the Message Flow Definition pane, right click the symbol of the Warehouse node you want to configure and click **Properties**.

   The **Warehouse** dialog is displayed, as shown in Figure 49 on page 172.



Drag elements from input to output to compose message

Figure 49. The Warehouse dialog

2. In the **Warehouse** dialog, click **Add** to define the input message.

   The **Add** dialog is displayed.

3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message from the drop-down lists. Click **OK**.

   The message tree structure appears in the Input pane. A tab is added to the Input pane showing the name of the message.

   Repeat this step for additional messages. To remove a message from the Input pane, click **Delete** when the relevant tab is to the fore.

4. Click **Add** to define the Output.

   The **Add** dialog is displayed. **Database table** is preselected.

5. In the **Add** dialog, enter Data Source and Table Name values. Click **OK**.

   The database tree structure is shown in the Output pane. You can name only one database in this pane. To delete table and database names, click **Delete**.

6. Now you must identify the columns you want to work with within the database table you identified. To do this:

   a. Right click anywhere in the white space around the database tree structure in the Output pane, and click **Add column**.

      The **Enter database column** dialog is displayed.

   b. Click in the Column field of the dialog, then enter the column identifier.

   c. Click **OK**.

      The column is added to the database tree structure in the Output pane.

   Repeat this process for each column you want to work with. (You need entries for only those columns you will be using, even if additional columns exist in the database.)

| Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Select the **Store Message** check box, and select the column in which you want to store the index record and attached binary object.
8. From the **Transaction Mode** drop-down list, select automatic or commit.
9. If you want to store a timestamp, select the **Store Timestamp** check box and select the column in which you want to store it.
10. If you want warnings to be treated as errors, click the **Advanced** tab of the **Warehouse** dialog, and select the **Treat warnings as errors** check box.
11. If you want to provide a description of this instance of the Warehouse node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Warehouse** dialog. Type a short description, or a long description, or both.
12. Click **OK** to finish configuring this Warehouse node.

Configuring the Warehouse node to store parts of a message

For a description of the properties of the Warehouse node and their possible values, see “Warehouse node properties” on page 171.

1. In the Message Flow Definition pane, right click the symbol of the Warehouse node you want to configure and click **Properties**.

   The **Warehouse** dialog is displayed, as shown in Figure 49 on page 172.
2. In the **Warehouse** dialog, click **Add** to define the input message.

   The **Add** dialog is displayed.
3. In the **Add** dialog, **Message** is preselected. Select the names of a message set and message template from the drop-down lists. Click **OK**.

   The message tree structure appears in the Input pane. A tab is added to the Input pane showing the name of the message.

   Repeat this step for additional messages. To remove a message from the Input pane, click **Delete** when the relevant tab is to the fore.
4. Click **Add** to define the Output.

   The **Add** dialog is displayed. **Database table** is preselected.
5. In the **Add** dialog, enter Data Source and Table Name values. Click **OK**.

   The database tree structure is shown in the Output pane. You can name only one database in this pane. To delete table and database names, click **Delete**.
6. Now you must identify the columns you want to work with within the database table you identified. To do this:
   a. Right click anywhere in the white space around the database tree structure in the Output pane, and click **Add column**.

      The **Enter database column** dialog is displayed.
   b. Click in the Column field of the dialog, then enter the column identifier.
   c. Click **OK**.

      The column is added to the database tree structure in the Output pane.

Repeat this process for each column you want to work with. (You need entries for only those columns you will be using, even if additional columns exist in the database.)

Note that there no validation is done on these values at this stage: the existence of the database, tables, and columns that you specify here cannot be determined until the message flow is deployed and executed within a broker.

7. Drag components of the input data from the Input pane to the target database column in the Output pane. This process is known as mapping, and represents the ESQL mappings that will be used in the processing of data through the node. The mappings are shown in the Input Message ESQL and Output Message ESQL pane. To delete mappings, right click on the expression to delete and click **Delete**. To delete all the expressions in the pane, click **Delete All**.
8. From the **Transaction Mode** drop-down list, select automatic or commit.
9. If you want to store a timestamp, select the **Store Timestamp** check box and select the column in which you want to store it.
10. If you want warnings to be treated as errors, click the **Advanced** tab of the **Warehouse** dialog, and select the **Treat warnings as errors** check box.
11. If you want to provide a description of this instance of the Warehouse node (which is recommended if you want other Control Center users to be able to make use of it), click the **Description** tab of the **Warehouse** dialog. Type a short description, or a long description, or both.
12. Click **OK** to finish configuring this Warehouse node.

For an example of configuring a Warehouse node within a message flow, see the retail scenario described in Appendix A, An example scenario (section "Configuring the Warehouse node" on page 269).

Using the IBM-supplied message flows

Some message flows are supplied with MQSeries Integrator V2.0.1. These are of two types:

1. The default message flows, which are:
   - A Neon message flow.

     This message flow provides function equivalent to an MQSeries Integrator Version 1.1 daemon. It is described in detail in "Version 1 Migration Compatibility message flow" on page 176.
   - A publish/subscribe message flow.

     This message flow provides a simple publish/subscribe service. It is described in detail in "The default publish/subscribe message flow" on page 178.
2. The verification message flows, which are:
   - The *ScribbleInversion* message flow.

     This message flow is required by the Scribble application, described in the *MQSeries Integrator Installation Guide*.
   - The *Soccer* message flow.

     This message flow is required by the Soccer Results Service, described in the *MQSeries Integrator Installation Guide*.
   - The *Postcard* message flow.

     This message flow is required by the Postcard application, described in the *MQSeries Integrator Installation Guide*.

   The definitions of these message flows are provided in the import file `SampleWorkspaceForImport`. The definition of the message set required by the Postcard IVP is provided in the file `PostcardMS.mrp`. The *MQSeries Integrator Installation Guide* describes the IVP message flows and message set in detail, and explains how to import the supplied definitions and save them for future use. It provides information on how to create the MQSeries resources required by the IVPs. It also describes how to deploy the message flows and message set to prepare for their use within a broker.

The following sections provide more information about the default message flows, and guidelines for using them.

Preparing to use the supplied message flows

Before you can use the supplied message flows, there are several actions you must complete. The message flows and the topology definitions in the import file make some assumptions For full details of these assumptions, and how to use the message flows, see the *MQSeries Integrator Installation Guide* for your operating system. A summary of actions is provided here.

1. Import the Postcard message set into the message repository using the command **mqsimrmimpexp**. You must restart the Configuration Manager to force it to pick up these changes.
2. Check out the Topology. Import the message flow and topology definitions from the supplied file and save them into the shared configuration repository.

3. If you want to run the Postcard verification program, you must assign the Postcard message set to the broker.
4. Deploy the message flows and the Postcard message set to the broker.
5. Check the success of the deployment: select the *Log* view and refresh the contents (you can click the green refresh icon or select **View —> Refresh**). It can take a few minutes for the deployment messages and responses flowing between the Configuration Manager and the broker to be displayed. Keep refreshing this view until you see the completion messages.

Version 1 Migration Compatibility message flow

This message flow can be deployed to any broker in your broker domain to provide equivalence to a NEON Rules and Formatter daemon. It incorporates the NEONRules and NEONFormatter nodes to process messages according to the Neon rules engine. An input node, to read messages from an input queue, and a set of output nodes, that provide failure, no-hit, and process action functions, are connected to the NEONRules node.

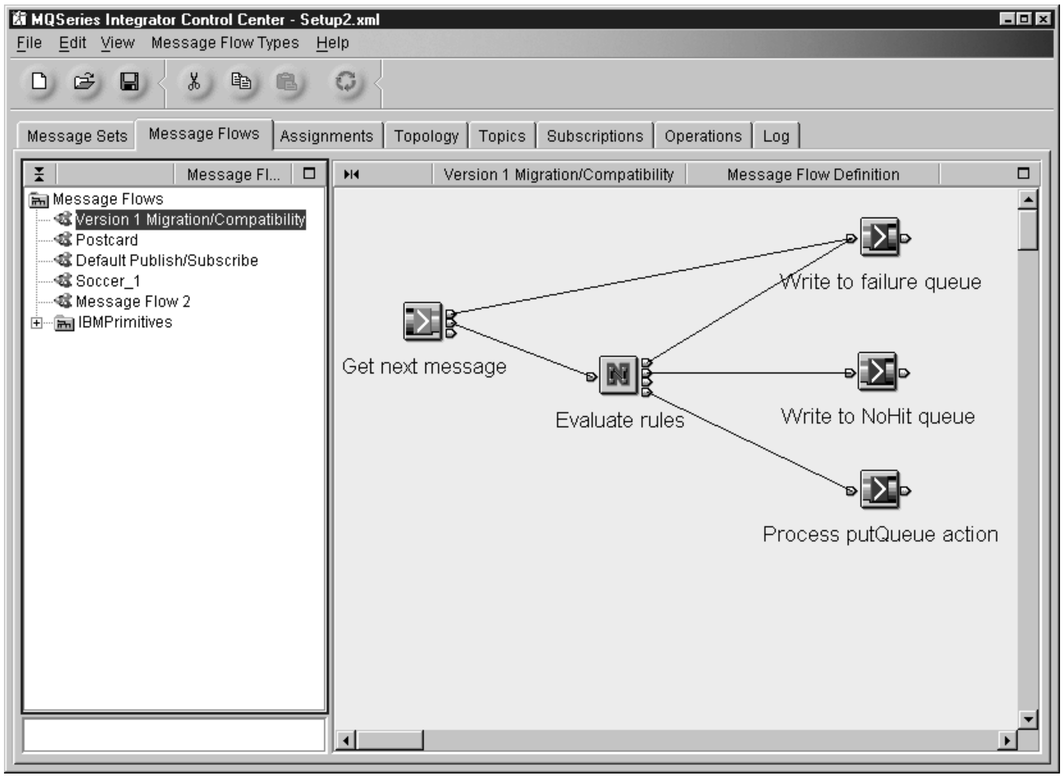


Figure 50. MQSeries Integrator V1 Migration Compatibility message flow

The operation of the five nodes within this message flow is:

1. The **Get next message** node (type MQInput) gets messages from a specified input queue and passes them to the NEONRules node.

   Before deploying this node you must configure it as follows:

   - Identify the input queue you want this message flow to use as the source of its messages. You specify the queue name in the **Queue Name** property of the **MQInput** properties dialog (Basic tab).

- You must ensure that the Message Domain on the Default tab of the properties dialog is set to NEON. You must specify the default NEON Application Group and Message Format attributes as the Message Set and Message Type properties respectively (on the Default tab) unless all messages that will be processed by the message flow contain an MQRFH header specifying the Application Group and Message Format. If one or more messages might not contain an MQRFH header, you must set these values.

  (For information about configuring an MQInput node, see “MQInput node” on page 136.)

2. The **Evaluate rules** node (type NEONRules) evaluates the message against the rules specified in the NEONRules engine.

   - If a putQueue action is triggered by the rules evaluation node, the message is routed through the putqueue terminal to the **Process putQueue action** node.
   - If no rule is triggered as a result of the evaluation is the NoHit condition, the message is routed to the **Write to NoHit queue** node.
   - If a failure is detected, the original message is routed to the **Write to failure queue** node.

   You do not have to configure this node before you deploy this message flow.

3. The **Write to NoHit queue** node (type MQOutput) puts the message to the output (NoHit) queue.

   Before you deploy this message flow, you must set the desired target queue and queue manager in the **Queue Name** and **Queue Manager Name** properties on the Basic tab of the properties dialog. The Destination Mode property on the Advanced tab is set to Queue Name: you must not change this property. (For information about configuring an MQOutput node, see “MQOutput node” on page 141.)

4. The **Write to failure queue** node (type MQOutput) puts the message to the output (failure) queue.

   Before you deploy this message flow, you must set the desired target queue and queue manager in the **Queue Name** and **Queue Manager Name** properties on the Basic tab of the properties dialog. The Destination Mode property on the Advanced tab is set to Queue Name: you must not change this property. (For information about configuring an MQOutput node, see “MQOutput node” on page 141.)

5. The **Process putQueue action** node (type MQOutput) puts the message to the output queue for further processing. The output queue is specified by the **Destination List** attached to the message by the EvaluateRules node.

   Before you deploy this message flow, you must set the desired target queue and queue manager in the **Queue Name** and **Queue Manager Name** properties on the Basic tab of the properties dialog. The Destination Mode property on the Advanced tab is set to Destination List: you must not change this property. (For information about configuring an MQOutput node, see “MQOutput node” on page 141.)

The default publish/subscribe message flow

This message flow provides a simple publish/subscribe service. It emulates exactly the basic publish/subscribe function supported by the Publish/Subscribe SDK, and is appropriate for all publish/subscribe services in which no additional processing of the message content is required.

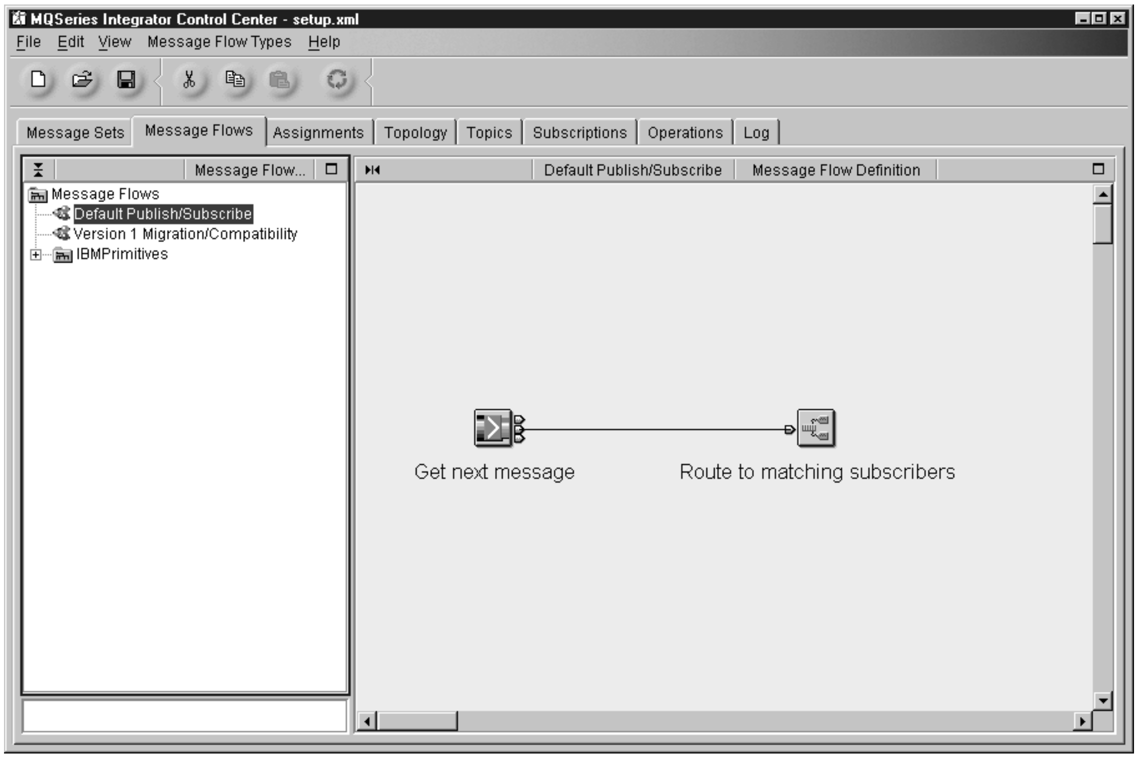

MQSeries Integrator Control Center - setup.xml
File Edit View Message Flow Types Help
Message Sets
Message Flows
Assignments
Topology
Topics
Subscriptions
Operations
Log
Message Flow...
Default Publish/Subscribe
Message Flow Definition
Message Flows
Default Publish/Subscribe
Version 1 Migration/Compatibility
IBMPrimitives
Get next message
Route to matching subscribers

Figure 51. MQSeries Integrator V1 Migration Compatibility message flow

The operation of the two nodes within this message flow is:

1. The **Get next message** node (type MQInput) gets the next available message from the input queue and passes it to the Publication node for matching against the table of subscription requests. The input queue is initially defined to be SYSTEM.BROKER.DEFAULT.STREAM, but you can change this according to your requirements.

   Failures in this node are not handled explicitly: the failing message is put to a backout queue or dead letter queue (if these queues have been defined). You can change this behavior by connecting other nodes to the failure terminal of this node, if you want to.

2. The **Route to matching subscribers** node (type Publication) matches the inbound publication against its internal subscription table (created and maintained in response to client subscription requests).

   For each matching subscription, the message is delivered to the subscriber by putting it to the queue on the queue manager specified in the subscription.

| For a more comprehensive example of a publish/subscribe message flow using
| MQSeries Integrator Version 2 functions, refer to Appendix A, "An example
| scenario" on page 257.

Copying the default message flows

If you want to deploy either of the default message flows, you are recommended to make a copy of it. This preserves the default message flow in your configuration repository for future reuse.

To make a copy of a default message flow:

1. Click the **Message Flows** tab.
2. Select the message flow you want to use from the folder IBM Default Message Flows, and click **Duplicate**.
3. A copy of the default message flow is created as a new message flow (with a new icon beside it). Click on the copy and select **Rename** to give it a unique name.
4. Make the changes you need to tailor your new copy of the supplied message flow.
5. Save your new message flow in the configuration repository using either **Check In** or one of the **File —> Check In** menu options.

Chapter 5. Defining the broker topology

This chapter describes the following tasks:

Authorization to work with Topology

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Operational domain controller**
  - **All roles**

  For information about setting your user role, see “Setting user roles” on page 13.
- Be a member of the MQSeries Integrator group **mqbrops**

The Topology view

To display the **Topology** view, click the **Topology** tab in the Control Center. Figure 52 on page 182 shows an example of the **Topology** view.

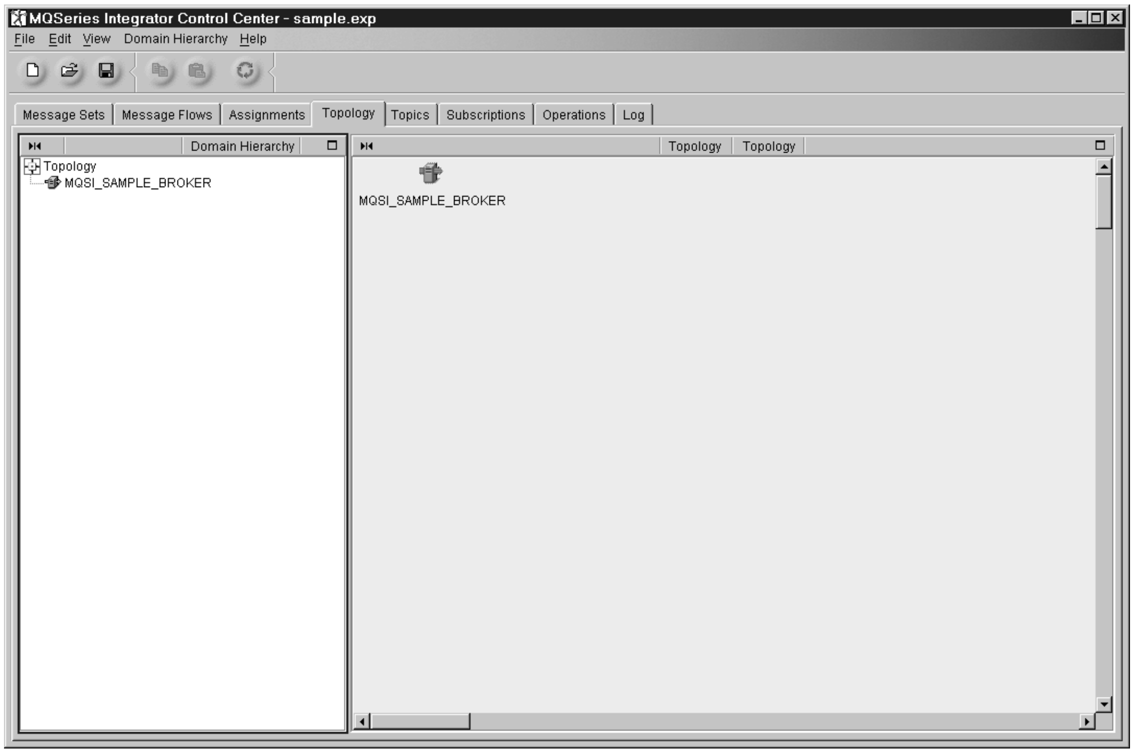


Figure 52. The Topology view. The left-hand pane, the Domain Hierarchy pane, shows a tree view of the topology of this broker domain. The right-hand pane, the Topology pane, contains an arrangement of graphical symbols that represent the current topology.

Controlling the appearance of the Topology pane

When you populate the broker domain in your workspace, graphical symbols representing collectives and brokers are added to the Topology pane. You can control the appearance and arrangement of these symbols by right-clicking in the Topology pane to display the **Topology** list, and selecting from the following actions:

Layout graph	Arranges the connected brokers and collectives in the Topology pane from left to right, right to left, top to bottom, or bottom to top.
Zoom	Alters the size of all broker and collective symbols in the Topology pane.
Manhattan style	Shows connections between brokers as lines at right angles.
Snap to grid	Aligns the symbols in the Topology pane on an invisible grid.

Checking out the Topology

The remainder of this chapter describes tasks that alter the topology of the broker domain. You cannot perform any of these tasks unless you have exclusive access to the Topology document, which you obtain by checking the Topology out of the configuration repository.

To check out the Topology:

1. In the Domain Hierarchy pane of the **Topology** view, right click the root of the Topology tree.[5]
2. Click **Check Out**.

The **Key** icon appears to the right of the root of the Topology tree to confirm that the Topology document is checked out. You can now update the Topology. Other users with access to this broker domain via another instance of the Control Center cannot make changes to the Topology while it remains checked out to you.

The collectives that you define in the Topology are considered part of the Topology itself for check out and check in purposes. Therefore if you want to update a collective, for example to add a broker to it, you must check out the Topology. You cannot check out the collective as a separate resource. See "Adding an existing broker to a collective" on page 188 and "Removing a broker from a collective" on page 191 for further details about updating collectives.

[5] Alternatively, you can right click anywhere on the background of the Topology pane, or you can highlight the root of the Topology tree and click on the **Domain Hierarchy** menu in the taskbar.

Creating a broker

To create a broker in the configuration repository:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. In the Domain Hierarchy pane of the **Topology** view, right click the root of the Topology tree. (Alternatively, you can right click anywhere on the background of the Topology pane, or you can highlight the root of the Topology tree and click the Topology menu in the Control Center taskbar.)
3. Click **Create —> Broker**.

   The **Create a new Broker** dialog, shown in Figure 53, is displayed.

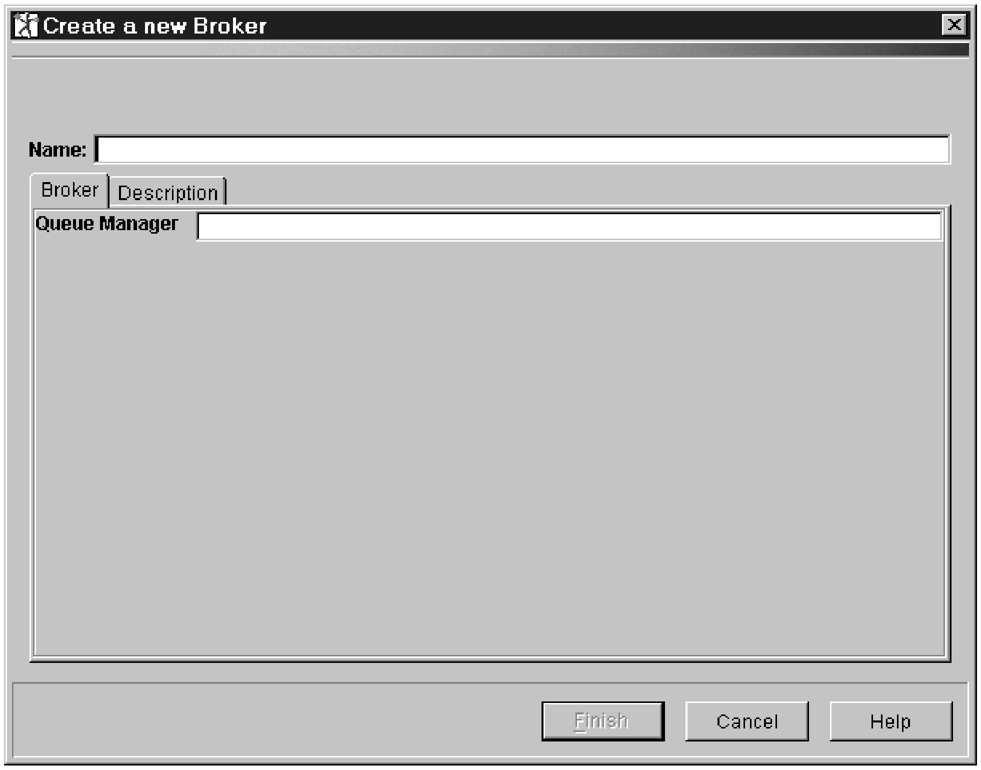


Figure 53. Create a new Broker dialog

4. In the **Name** field, type the name of your broker.

   This must be exactly the name specified when the broker was created (that is, the broker name specified on the **mqsicreatebroker** command). You must specify the name using the same case (lower, upper, or mixed). This value is required and must be unique.
5. In the **Queue Manager** field, type the name of the broker's queue manager.

   This must be exactly the name specified for the broker's queue manager when the broker was created (that is, the queue-manager name specified on the **mqsicreatebroker** command). You must specify the name using the same case (lower, upper, or mixed). This value is required and must be unique in your MQSeries network.

6. For documentation purposes, you can provide either a short description, or a long description, or both, of your broker, though a description is not required.

   If you want to provide a description, click the **Description** tab in the **Create a new Broker** dialog, and type some text.

7. Click **Finish** in the **Create a new Broker** dialog to complete creation of this broker.

Confirmation that your new broker has been created appears in two places in the **Topology** view:

- An entry representing the broker appears under the root of the Topology tree in the Domain Hierarchy pane. The **New** icon next to the broker entry indicates that this new definition has not yet been checked into the shared configuration.
- A graphical symbol of the broker appears in the Topology pane.

| It also appears in the Domain Topology pane of the Assignments view.

If you have no further topology changes to make, check in the Topology as described in "Checking in the Topology" on page 196.

Creating a collective

To create a collective in the configuration repository:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. In the Domain Hierarchy pane of the **Topology** view, right click the root of the Topology tree. (Alternatively, you can right click anywhere on the background of the Topology pane, or you can highlight the root of the Topology tree and click the Topology menu in the Control Center taskbar.)
3. Click **Create —**> **Collective**.

   The **Create a new Collective** dialog, shown in Figure 54, is displayed.

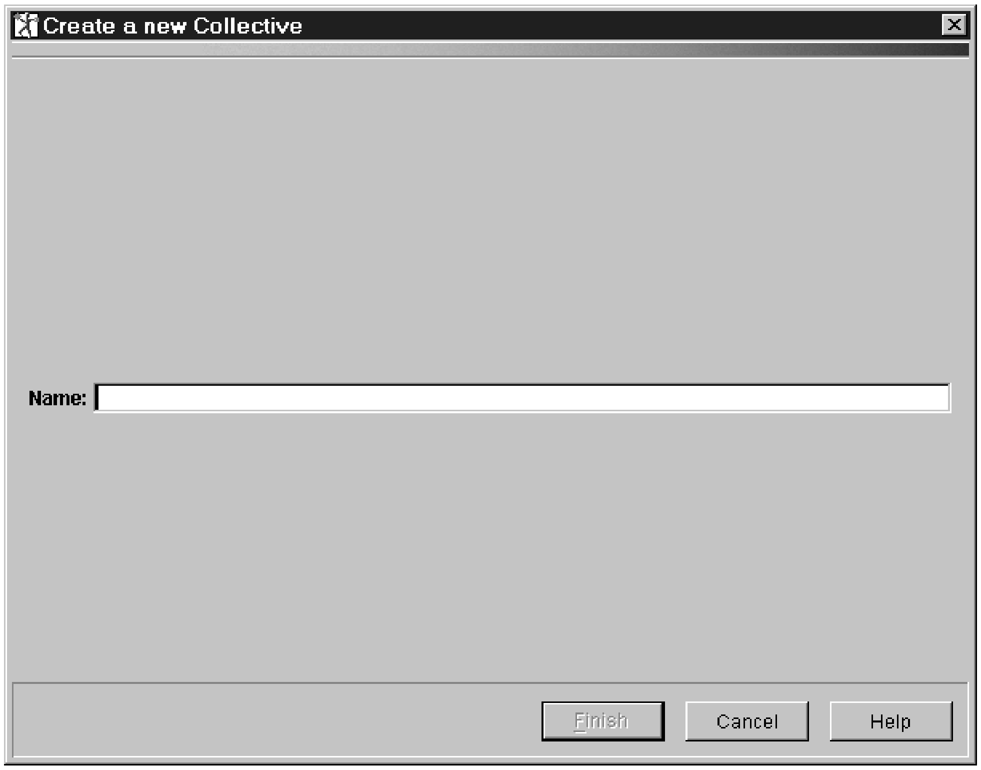


Figure 54. Create a new Collective dialog

| 4. In the **Name** field, type the name of your collective. This must follow the
| naming rules described in "Naming Control Center resources" on page 20 and
| must be unique within your broker domain.

5. Click **Finish** in the **Create a new Collective** dialog to complete creation of this collective.

Confirmation that your new collective has been created appears in two places in the **Topology** view:

- A folder representing the collective appears under the root of the Topology tree in the Domain Hierarchy pane. The **New** icon next to the collective entry indicates that this new definition has not yet been checked into the shared configuration.

- A graphical symbol representing the empty collective appears in the Topology pane.

If you have no further topology changes to make, check in the Topology as described in “Checking in the Topology” on page 196.

Note that collectives are checked in as part of the Topology check in, not as separate resources, as they exist only in the Topology document.

Adding an existing broker to a collective

There are several ways of adding an existing broker to a collective using the Control Center. This section describes one of these methods in detail, then mentions others briefly.

When you add brokers to the collective, the collective symbol in the Topology pane can appear crowded. To increase the size of the collective symbol, drag the double-headed arrow at the bottom-right corner of the symbol downward.

To add an existing broker to a collective:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183. (You cannot check out the collective.)
2. Right click the collective folder in the Topology tree.
3. Click **Add —> Broker**.

   The **Add an existing Broker** dialog, shown in Figure 55, is displayed.

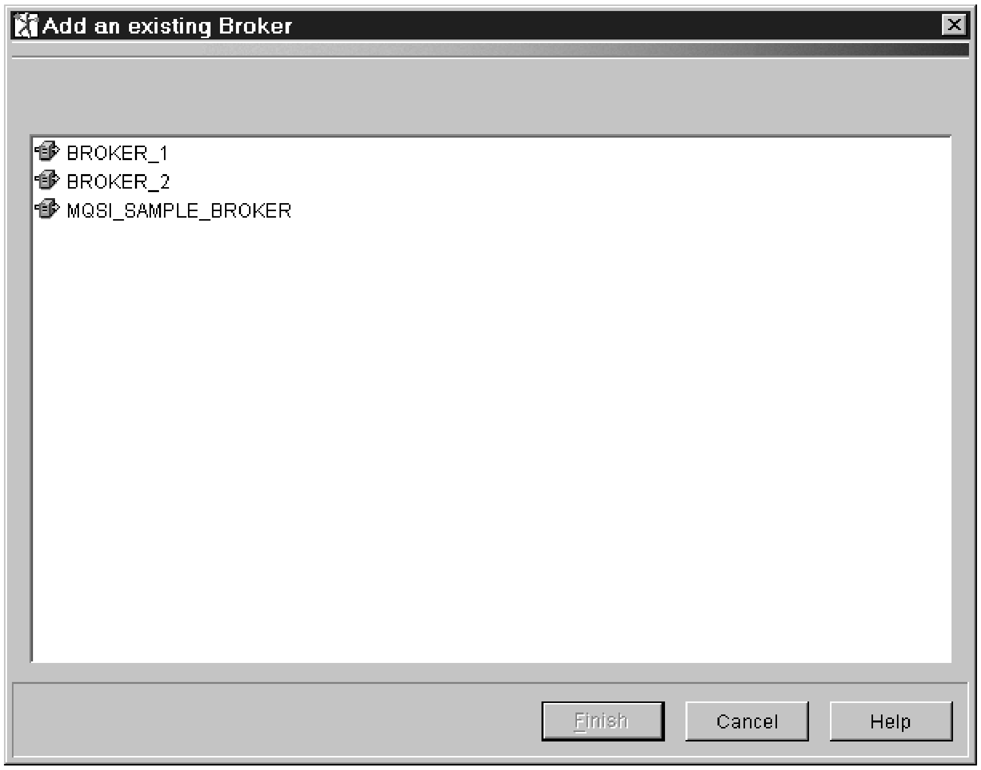


Figure 55. Add an existing Broker dialog. This dialog lists all brokers that you have created and those that exist in the shared configuration that are eligible for inclusion in this collective (that is, that are not already assigned to another collective).

- To select a single broker from this list, click the broker name.
- To select multiple brokers that appear sequentially in the list, click on the first broker you want, press and hold the Shift key, then click on the last broker you want. This action selects the two brokers you highlighted, plus any that appear between these two in the list.
- To select multiple brokers that do not appear in a sequence in the list, hold down Ctrl and click each broker you want.

4. When you have selected the brokers you want to add to the collective from this list, click **Finish**.

Confirmation that the selected brokers have been added to the collective appears in two places in the **Topology** view:

- In the Domain Hierarchy pane, the brokers are now shown under the collective folder.
- In the Topology pane, the broker symbols now appear inside the collective symbol.

Alternatively, you can invoke the **Add an existing Broker** dialog from the **Topology** menu in the Control Center taskbar.[6]

You can also add an existing broker to a collective simply by:

- Dragging the broker symbol in the Topology pane into the symbol of the collective

  or
- Dragging the broker entry in the Domain Hierarchy pane into the symbol of the collective in the Topology pane.

If you have no further topology changes to make, check in the Topology as described in “Checking in the Topology” on page 196.

[6] When the Topology pane has the focus, the **Topology** menu appears in the Control Center taskbar. When the Domain Hierarchy pane has the focus, the **Domain Hierarchy** menu appears in the Control Center taskbar. The menu items of the **Topology** and **Domain Hierarchy** menus are identical.

Creating a broker to add to a collective

To create a broker to add to a collective:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. In the Domain Hierarchy pane, right click on the collective folder in the Topology tree.
3. Click **Create —> Broker**.

   The **Create a new Broker** dialog, shown in Figure 53 on page 184, is displayed.[7]
4. In the **Name** field, type the name of your broker.

   This must be exactly the name specified when the broker was created (that is, the broker name specified on the **mqsicreatebroker** command). You must specify the name using the same case (lower, upper, or mixed). This value is required and must be unique.
5. In the **Queue Manager** field, type the name of the broker's queue manager.

   This must be exactly the name specified for the broker's queue manager when the broker was created (that is, the queue-manager name specified on the **mqsicreatebroker** command). This value is required.
6. For documentation purposes, you can provide either a short description, or a long description, or both, of your broker, though a description is not required.

   If you want to provide a description, click the **Description** tab in the **Create a new Broker** dialog, and type some text.
7. Click **Finish** in the **Create a new Broker** dialog to complete creation of this broker.

Confirmation that your new broker has been created appears in two places in the **Topology** view:

- An entry representing the broker appears under the appropriate collective folder in the Domain Hierarchy pane. The **New** icon next to the broker entry indicates that this new definition has not yet been checked into the shared configuration.
- A graphical symbol of the broker appears inside the symbol of the appropriate collective in the Topology pane.

The broker has been both created and added to the collective.

If you have no further topology changes to make, check in the Topology as described in "Checking in the Topology" on page 196.

[7] You can also select the **Create —> Broker** action by highlighting the collective symbol in the Topology pane, then clicking the **Topology** menu in the Control Center taskbar.

Removing a broker from a collective

To remove a broker from a collective:

1. Ensure that you have checked out the Topology, as described in “Checking out the Topology” on page 183. (You cannot check out the collective.)
2. In the Topology pane, right click the symbol of the broker inside the collective symbol.
3. Click **Remove**.

Confirmation that the broker has been removed from the collective appears in two places in the **Topology** view:

- In the Domain Hierarchy pane, the broker is no longer shown under the collective folder.
- In the Topology pane, the broker symbol now appears outside the collective symbol.

Alternatively, you can simply drag the broker symbol out of the symbol of the collective in the Topology pane. You can also right click the broker entry under the relevant collective in the Domain Hierarchy pane, and click **Remove**.

If the removed broker was connected to a broker outside the collective, you might need to remove the connection also. For more information, see “Deleting the connection between brokers” on page 193.

If you have no further topology changes to make, check in the Topology as described in “Checking in the Topology” on page 196.

Connecting brokers

To connect one broker to another:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. In the Topology pane, right click the symbol of one of the two brokers you want to connect.
3. Click **Connect —> port**.

   The cursor becomes a cross-hair attached by a red line to the broker you selected initially.
4. Move the cross-hair to the symbol of the broker you want to connect to, and click.

The brokers are now connected. In the Topology pane, a line connects the symbols of the two brokers.

Note that:

- You cannot connect a single broker outside a collective to more than one of the brokers in a single collective.
- You can connect a single broker outside a collective to multiple collectives (that is, to one broker per collective).
- You can connect a broker in one collective to a broker in another collective.
- You can connect two brokers outside a collective.
- A connection is created only if a cycle of connections would not result. If the addition of a connection would cause a cycle, an error message is issued.

If you have no further topology changes to make, check in the Topology as described in "Checking in the Topology" on page 196.

Deleting the connection between brokers

To delete the connection between two brokers:

1. Ensure that you have checked out the Topology, as described in “Checking out the Topology” on page 183.
2. In the Topology pane, right click on the line between the two brokers you want to disconnect.
3. Click **Delete**.

The line between the two brokers disappears. The brokers are now disconnected.

If you have no further topology changes to make, check in the Topology as described in “Checking in the Topology” on page 196.

Deleting a broker from the topology

This procedure describes how to delete a broker reference from the configuration repository. This procedure does not delete the broker from your system: it simply marks the broker as logically deleted from the configuration repository. For a full description of the process required to delete a broker from your broker domain, see "Deleting a broker from the broker domain" on page 217.

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. Ensure that the broker you want to delete is checked in. If it is not (that is, if the **Key** icon is displayed next to its entry in the Domain Hierarchy pane), right click the broker entry and click **Check In**. All execution groups assigned to the broker must also be checked in before you can delete the broker.
3. In the Topology pane, right click the broker you want to delete.
4. Click **Delete**.
5. A confirmation message is displayed. If you want to proceed with the deletion, click **Yes**.

Confirmation that the broker has been deleted appears in two places in the **Topology** view:

- The broker entry no longer appears in the Domain Hierarchy pane.
- The broker symbol no longer appears in the Topology pane.

If the broker was connected to another, the connection is also deleted.

If you have no further topology changes to make, check in the Topology as described in "Checking in the Topology" on page 196.

Renaming a broker

You might need to rename a broker if your original attempt at creating a broker reference contained an error: renaming the broker is simpler than deleting and recreating it.

To rename a broker:

1. Ensure that you have checked out the Topology, as described in "Checking out the Topology" on page 183.
2. Ensure that the broker you want to rename is checked out. If it is not (that is, if neither the **Key** icon nor the **New** icon is displayed next to its entry in the Domain Hierarchy pane), right click the broker entry in the Topology tree and click **Check Out**.
3. In the Topology pane, right click the broker you want to rename.
4. Click **Rename**.

   The **Rename Broker** dialog is displayed.
5. In the New name field, type the new name of the broker. This must be exactly the name specified on the **mqsicreatebroker** command. Click **Finish**.

Confirmation that the broker has been renamed appears in two places in the **Topology** view:

- The broker entry in the Domain Hierarchy pane shows the new name.
- The broker symbol in the Topology pane shows the new name.

If you need also to specify a different queue manager name for the renamed broker:

1. In the Topology pane, right click the broker you want to rename.
2. Click **Properties**.
3. In the broker's properties panel, type the new queue manager name, and correct the description if necessary. The name you specify must be exactly the name specified for this broker's queue manager on the **mqsicreatebroker** command. Click **Finish**.

If you have no further topology changes to make:

1. Check in the broker:
   a. In the Topology pane, right click the broker you want to check in.
   b. Click **Check In**.

   The **Key** icon against the broker entry in the Topology tree disappears.
2. Check in the Topology as described in "Checking in the Topology" on page 196.

Checking in the Topology

When you have finished making changes to the Topology, you *must* check it in. Until you check in the Topology, no one else is able to make changes to the topology of this broker domain, nor can you deploy the changes you have made.

You can check in Topology changes only, or all changes.

Checking in Topology changes

To check in the Topology:

1. Right click the root of the Topology tree.
2. Click **Check In** to store the Topology document in the Configuration Manager database.

To confirm that the Topology has been checked in:

- The **Key** icon disappears from the root of the Topology tree in the Domain Hierarchy pane.
- The **New** icon against any new brokers and collectives in the Topology tree disappears, indicating that they have also been checked into the shared configuration. Newly created resources are checked in automatically to ensure that the configuration remains consistent.

Note that any brokers with the **Key** icon against them must be checked in separately; they are not checked in as part of the general Topology check in.

Checking in multiple changes

The **File —> Check In** menu option allows you to check in multiple changes. You can use this instead of checking in individual objects such as the Topology. The options are:

- **File —> Check In —> List**
- **File —> Check In —> All in Current Workspace**
- **File —> Check In —> All (Save to Shared)**

These options are more efficient when you have many different resources checked out. The List option also allows you to check which resources are checked out in your current workspace before you decide which resources to check in.

For more information about check in options, see “Saving the workspace to the shared repository” on page 17.

Making changes operational

In checking in resources that are new or that you have altered, you make them visible in the shared configuration. However, the changes you have made have no operational effect until you *deploy* them in the broker domain. For information about deploying resources, see Chapter 7, “Deploying configuration data” on page 213.

Chapter 6. Assigning resources to a broker

This chapter describes the following tasks:

Authorization to assign resources to a broker

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Message flow and message set assigner**
  - **All roles**

  For information about setting your use role, see “Setting user roles” on page 13.
- Be a member of the MQSeries Integrator group **mqbrasgn**

The Assignments view

To display the **Assignments** view, click the **Assignments** tab in the Control Center. Figure 56 on page 200 shows an example of the **Assignments** view.

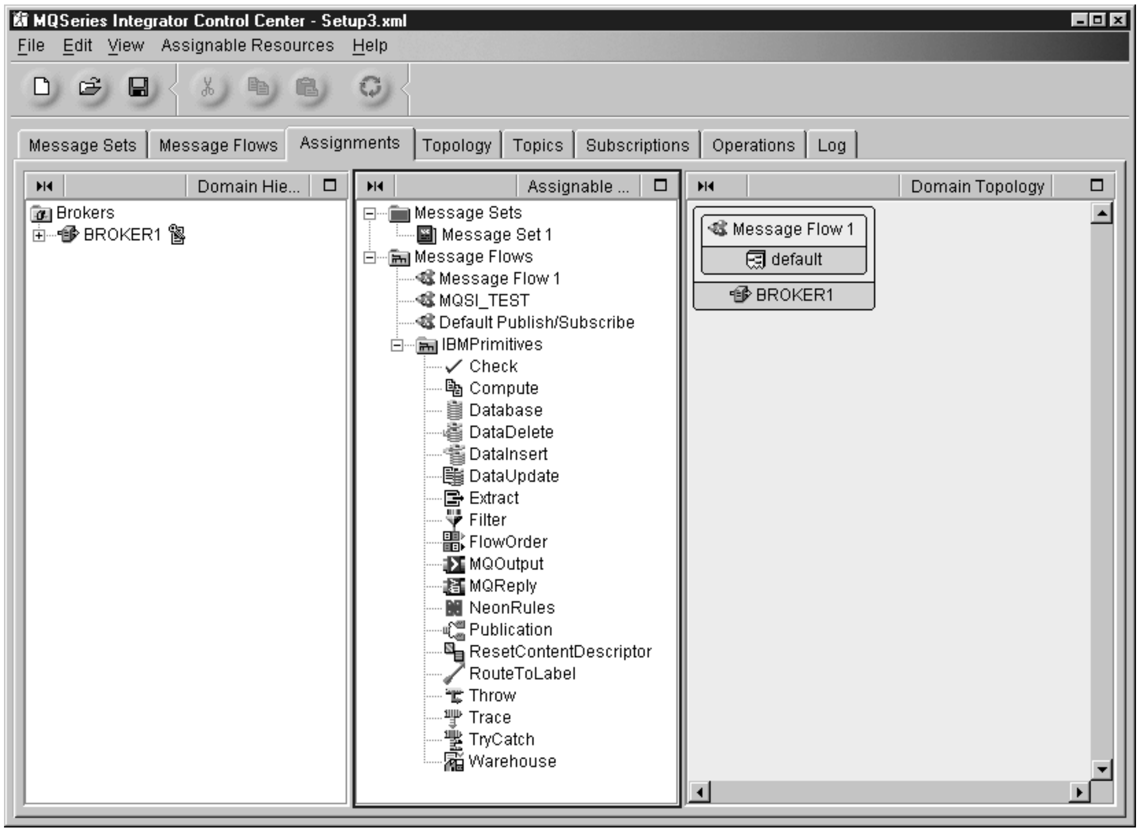


Figure 56. The Assignments view. The left-hand pane, the Domain Hierarchy pane, shows the current hierarchy of brokers, execution groups, message flows, and message sets in your workspace. The center pane, the Assignable Resources pane, shows the message sets and message flows in your workspace. The right-hand pane, the Domain Topology pane, shows in a graphical form the current assignment of execution groups to brokers; of message flows to execution groups; and of message sets to brokers in your workspace.

Creating an execution group

When you create a broker, it has a default execution group. If you want additional execution groups, you must create them explicitly.

To create an execution group:

1. Ensure that the broker to which you want to assign the new execution group is checked out of the shared configuration.

   If the broker entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the broker entry, and click **Check out**.

2. In the Domain Hierarchy pane, right click the entry for the broker.
3. Click **Create —**> **Execution Group**.

   The **Create a new Execution Group** dialog is displayed.

4. In the **Name** field, type the name of the execution group. This must follow the naming rules described in "Naming Control Center resources" on page 20. Click **Finish**.

The new execution group appears:

- Inside the broker symbol in the Domain Topology pane, alongside the symbols for other execution groups assigned to this broker
- Beneath the broker folder in the Domain Hierarchy pane, with a **New** icon against it

Assigning message flows to execution groups

To assign a message flow to an execution group:

1. Ensure that the execution group to which you want to assign the message flow is checked out of the shared configuration.

   If the execution group entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the execution group entry, and click **Check out**.

2. Drag the message flow symbol from the Assignable Resources pane into the symbol of the execution group in the Domain Topology pane. The Assignable Resources pane lists all message flows in your workspace.

3. Check in the execution group.

An alternative approach, and one that is useful when you have a large number of message flows to assign to a single execution group, is as follows:

1. In the Domain Hierarchy pane, right click the entry for the checked-out execution group to which you want to assign a message flow.

2. Click **Add —>** **Message Flow**.

   The **Add an existing Message Flow** dialog is displayed, showing all message flows in this workspace. Figure 57 shows an example of the **Add an existing Message Flow** dialog.

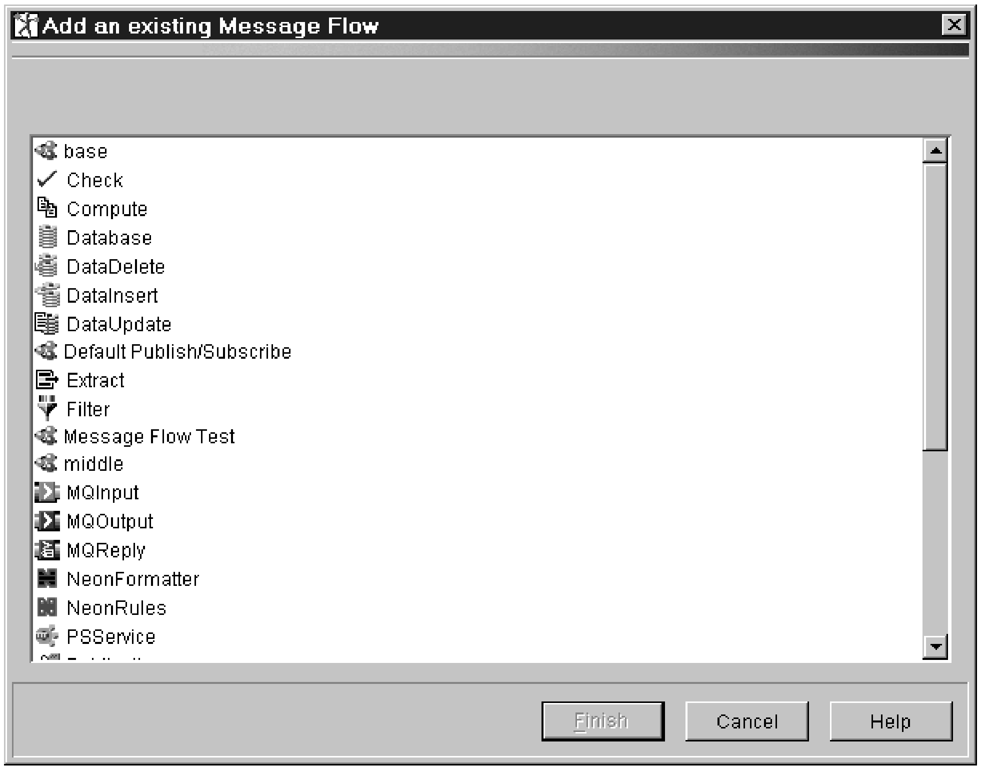


Figure 57. The Add an existing Message Flow dialog. This dialog lists all message flows in your workspace.

- To select a single message flow from this list, click the message flow name.
- To select multiple message flows that appear sequentially in the list, click the first message flow you want, press and hold the Shift key, then click the last message flow you want. This action selects the two message flows you highlighted, plus any that appear between these two in the list.
- To select multiple message flows that do not appear in a sequence in the list, hold down Ctrl and click each message flow you want.

3. When you have selected the message flows you want to assign to the execution group from this list, click **Finish**.

When you assign message flows:

- You cannot add a single message flow more than once to any execution group.
- Subflows are included in the **Add an existing Message Flow** dialog. If you are using a subflow in a higher-level flow, you must assign only the higher-level flow to the execution group: this action includes the subflow.

The message flows you selected appear:

- Inside the execution group symbol in the Domain Topology pane.
- Beneath the execution group entry in the Domain Hierarchy pane.

When you deloy an assigned message flow, each node in the message flow is allocated a label that consists of the name you gave the node qualified by its containing flow, if any. This guarantees the uniqueness of each label within the message flow.

For example, if you deploy a message flow called *Top*, that contains an instance of subflow Base that you have called *myBase*, and your subflow contains a compute node called *myCompute*, the label allocated to this compute node is *Top.myBase.myCompute*.

A few of the primitive nodes are known as compound nodes (that is, they are based on another node). The inclusion of compound nodes in the message flow affects the labels allocated. For example, the extract node is based on the compute node. Therefore if your node in the message flow *myBase* is an extract node, the label will be *Top.myBase.myExtract.Compute1*.

Setting the properties of an assigned message flow

You can change some of the properties of a message flow after you have assigned it to an execution group. To change the properties of a message flow, right click the entry for the message flow under the appropriate execution group in the Domain Hierarchy pane, and click **Properties**. The properties whose values you can change are:

Additional Instances
Specifies the number of threads that the broker should start in order to read messages from the input queue named on the MQInput node of the message flow and process them concurrently. You can have up to 256 threads.

Having additional threads can increase the throughput of a message flow. However, you should consider the impact on message order and set the Order Mode property on the MQInput node (**Advanced** tab) accordingly. You must also ensure that the input queue has been defined with the SHARE attribute to enable multiple threads to read the same queue.

Its default value is 0.

Commit Count
Specifies how many input messages are processed by a message flow before a syncpoint is taken (by issuing an MQCMIT).

This attribute should be used only if the Additional Instances property is set to 0.

The default value of 1 is also the minimum permitted value. Change this attribute if you want to avoid frequent MQCMIT calls when messages are being processed quickly and the lack of an immediate commit can be tolerated by the receiving application.

Use the Commit Interval to ensure that a commit is performed periodically when not enough messages are received to fulfill the Commit Count.

Commit Interval
Specifies a time interval at which a commit is taken when the Commit Count property is greater than 1 (that is, where the message flow is batching messages) but the number of messages processed has not reached the value of the Commit Count property. It ensures that a commit is performed periodically when not enough messages are received to fulfill the Commit Count.

The time interval is specified in seconds and must be in the range 0 through 60.

This attribute should be used only if the Additional Instances property is set to 0.

Its default value is 0.

Coordinated transaction
Controls whether the message flow is processed as a global transaction, coordinated by MQSeries. Such a message flow is said to be fully globally-coordinated.

Use coordinated transactions only where you need the message and any database updates performed by the message flow to be processed in a single unit-of-work, using a two-phase commit protocol. This means that both the message is read and the database updates are performed, or neither is done.

If you change this value, you **must** ensure that the broker's queue manager is configured correctly. If you do not set up the queue manager correctly, a message is generated by the broker when a message is received by the message flow to indicate that although the message flow is to be globally coordinated, the queue manager configuration does not support this.

See the *MQSeries Integrator Administration Guide* for information about which databases are supported as participants in a global transaction, and the *MQSeries System Administration* book for how to configure MQSeries and the database managers.

The default value is no.

Assigning message sets to brokers

To assign a message set to a broker:

1. Ensure that the broker to which you want to assign the message set is checked out of the shared configuration.

   If the broker entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the broker entry, and click **Check out**.

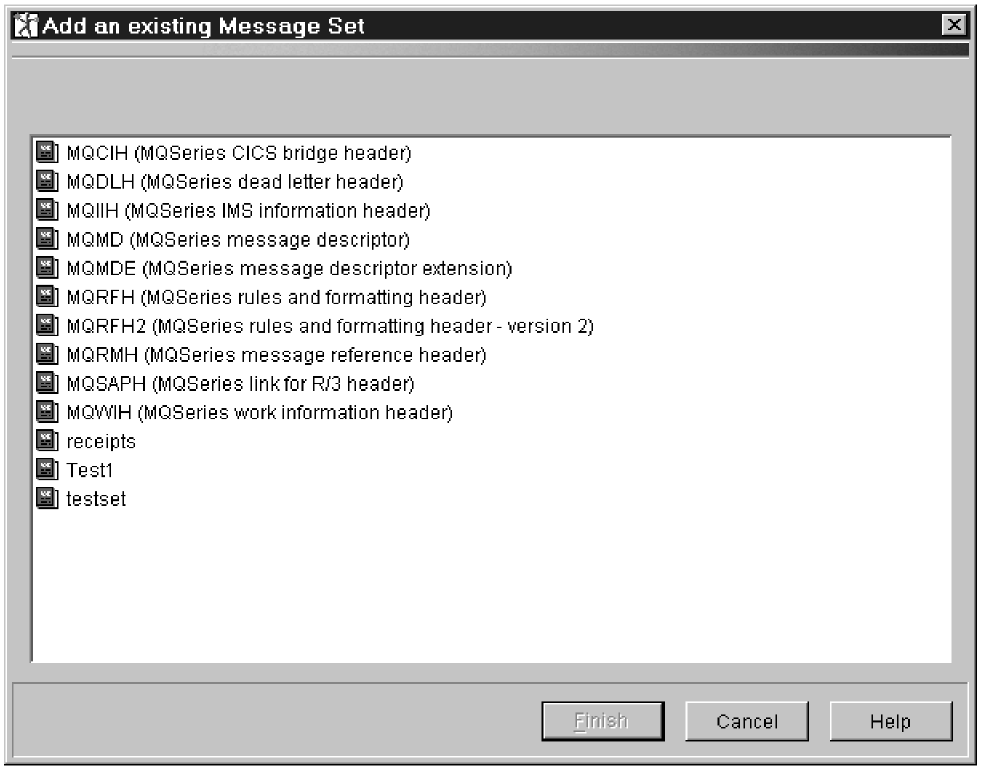
2. Drag the message set symbol from the Assignable Resources pane into the symbol of the broker (but not into any execution group contained in the broker) in the Domain Topology pane. The Assignable Resources pane lists all message sets in your workspace.

An alternative approach, and one that is useful when you have a large number of message sets to assign to a single broker, is as follows:

1. In the Domain Hierarchy pane, right click the entry for the checked-out broker to which you want to assign a message set.

2. Click **Add —> Message Set**.

   The **Add an existing Message Set** dialog is displayed, showing all message sets in this workspace. Figure 58 shows an example of the **Add an existing Message Set** dialog.



MQCIH (MQSeries CICS bridge header)
MQDLH (MQSeries dead letter header)
MQIIH (MQSeries IMS information header)
MQMD (MQSeries message descriptor)
MQMDE (MQSeries message descriptor extension)
MQRFH (MQSeries rules and formatting header)
MQRFH2 (MQSeries rules and formatting header - version 2)
MQRMH (MQSeries message reference header)
MQSAPH (MQSeries link for R/3 header)
MQWIH (MQSeries work information header)
receipts
Test1
testset

Figure 58. The Add an existing Message Set dialog. This dialog lists all message sets in your workspace.

- To select a single message set from this list, click the message set name.
- To select multiple message sets that appear sequentially in the list, click the first message set you want, press and hold the Shift key, then click the last message set you want. This action selects the two message sets you highlighted, plus any that appear between these two in the list.
- To select multiple message sets that do not appear in a sequence in the list, hold down Ctrl and click each message set you want.

   Note that you cannot assign a single message set more than once to any broker.

3. When you have selected the message sets you want to assign to the broker from this list, click **Finish**.

The message sets you selected appear:

- Inside the broker symbol in the Domain Topology pane.
- Beneath the broker symbol in the Domain Hierarchy pane.

Removing resources from a broker

You can remove message sets from the broker to which they have been assigned, you can remove message flows from the execution groups to which they have been assigned, and you can delete execution groups from their owning broker.

Deleting an execution group from a broker

To delete an execution group from a broker:

1. Ensure that the broker from which you want to delete the execution group is checked out of the shared configuration.

   If the broker entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the broker entry, and click **Check out**.
2. Ensure that the execution group you want to delete is *not* checked out.

   If the execution group entry in the Domain Hierarchy pane of the **Assignments** view has the **Key** icon against it, right click the execution group entry and click **Check in**.
3. Right click the execution group entry under the broker in the Domain Hierarchy pane, or right click the execution group symbol in the Domain Topology pane, and click **Delete**.

The execution group and any message flow assignments it contains are deleted:

- From the broker symbol in the Domain Topology pane
- From the relevant broker entry in the Domain Hierarchy pane

The message flows themselves are not deleted or removed from your workspace, and remain in the Assignable Resources pane to be assigned to other execution groups.

Removing a message set from a broker

To remove a message set from a broker:

1. Ensure that the broker from which you want to remove the message set is checked out of the shared configuration.

   If the broker entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the broker entry, and click **Check out**.
2. Right click the message set symbol in the Domain Topology pane, or right click its entry in the Domain Hierarchy pane, and click **Remove**.

The message set assignment disappears from:

- The broker symbol in the Domain Topology pane
- The broker entry in the Domain Hierarchy pane

The message set is not deleted or removed from your workspace, and is still available in the Assignable Resources pane to be assigned to other brokers.

Removing a message flow from an execution group

To remove a message flow from an execution group:

1. Ensure that the execution group from which you want to remove the message flow is checked out of the shared configuration.

   If the execution group entry in the Domain Hierarchy pane of the **Assignments** view has neither the **Key** icon nor the **New** icon against it, right click the execution group entry, and click **Check out**.

2. Right click the message flow symbol inside the execution group symbol in the Domain Topology pane, or right click the message flow entry in the Domain Hierarchy pane, and click **Remove**.

The message flow assignment disappears from:

- The execution group symbol in the Domain Topology pane
- The execution group entry in the Domain Hierarchy pane

The message flow is not deleted or removed from your workspace, and is still available in the Assignable Resources pane to be assigned to other execution groups.

Checking in the Assignments

When you have finished assigning resources to a broker, you *must* check in any brokers and execution groups that are checked out. Until you check in brokers and execution groups, no one else is able to make changes to them, nor can you deploy the assignments you have made.

When a newly created broker or execution group is checked in, all related resources are also checked in automatically. For example, when you check in a new broker, its default execution group and the Topology document are also checked in, to ensure consistency of configuration data. MQSeries Integrator does this to prevent you from accidentally stranding important information in a way that cannot easily be corrected. After a new resource has been checked in for the first time, you can check individual resources out, modify them, and check them in individually.

You can check in brokers and execution groups only, or all changes.

Checking in assignments

To check in a broker:

1. Right click the broker entry in the Domain Hierarchy pane.
2. Click **Check in** to store the broker in the shared configuration.

To confirm that the broker assignments have been checked in, the **Key** icon disappears from the broker entry in the Domain Hierarchy pane.

To check in an execution group:

1. Right click the execution group entry in the Domain Hierarchy pane.
2. Click **Check in** to store the execution group in the shared configuration.

To confirm that the execution group has been checked in, the **Key** icon disappears from the execution group in the Domain Hierarchy pane.

Checking in multiple changes

The **File —**> **Check In** menu option allows you to check in multiple changes. You can use this instead of checking in individual objects such as Assignments data. The options are:

- **File —**> **Check In —**> **List**
- **File —**> **Check In —**> **All in Current Workspace**
- **File —**> **Check In —**> **All (Save to Shared)**

These options are more efficient when you have many different resources checked out. The List option also allows you to check which resources are checked out in your current workspace before you decide which resources to check in.

For more information about check in options, see “Saving the workspace to the shared repository” on page 17.

Refreshing the Assignments view

You can update the Assignments view with the most recent contents of the configuration repository at any time. Select **View —>** **Refresh from Shared**. This shows changes to resources that are not checked out: if you have a resource checked out, the version you have in your current workspace is not overwritten with the version from the shared configuration.

Making changes operational

In checking in resources that are new or that you have altered, you make them visible in the shared configuration. However, the changes you have made have no operational effect until you *deploy* them in the broker domain. For information about deploying resources, see Chapter 7, “Deploying configuration data” on page 213.

Chapter 7. Deploying configuration data

The following types of configuration data need to be *deployed* before they can take effect in the broker domain:

Assignments data	Execution groups to brokers; message flows to execution groups; and message sets to brokers.
Topics data	Topics and associated Access Control Lists (ACLs) for the broker domain
Topology data	Broker and collective data for the broker domain

When you request deployment of any type of configuration data, the Configuration Manager copies the relevant configuration data from the shared configuration and transmits it to the relevant brokers. When the deployment is successful, the brokers are able to act in accordance with the newly deployed data.

This chapter begins with a discussion of the deployment function, then provides instructions for deploying the various types of configuration data.

Three types of deployment

You can deploy assignments data, topics data, topology data, or all three types of data at once. For each of these types of configuration data, you can request:

- A complete deployment
- A delta deployment

In addition, you can request a forced deployment. This type of deployment is valid only when all configuration data of all types is being deployed.

Complete deployment

A complete deployment:

1. Deletes all configuration data of that type that is currently deployed on the target brokers
2. Creates new configuration data from the shared configuration

For example, if you request a complete deployment of topics data, the Configuration Manager deploys instructions to all brokers to delete *all* currently deployed topics data and create a new set of topics data from those in the shared configuration.

Delta deployment

When you request a delta deployment, the Configuration Manager compares the configuration data of that type that is currently deployed on the target brokers with the shared configuration, and deploys only the differences between the two versions. Therefore, the delta deployment is better for performance, especially when you have a large amount of configuration data in the shared configuration.

Forced deployment

The forced deployment, which overrides any outstanding deployment request, is used typically to correct error situations. Therefore, to maintain consistency of the configuration data throughout the broker domain, a forced deployment is allowed only when deploying all types of configuration data. A forced deployment is always a complete deployment.

A summary of deployment actions

Table 27 summarizes the available deployment actions, showing:

- The type of deployment supported for each type of configuration data
- The Control Center view from which the deployment can be requested
- The brokers to which the deployment can be targeted

Table 27. Deployment summary

Data deployed	Complete	Delta	Forced	From Control Center view	Target
Assignments	Yes	Yes	No	Assignments	Single broker Single execution group All brokers
Topics	Yes	Yes	No	Topics	All brokers
Topology	Yes	Yes	No	Topology	All brokers
All types	Yes	Yes	Yes	Topology	All brokers

Note: The Topics, Topology, and All types deployments must apply to *all* brokers to maintain consistent configuration data throughout the broker domain.

The stages of the deployment process

Deployment of configuration data takes place in two stages.

Stage one of deployment

During stage one of deployment, which is synchronous, the Configuration Manager sends a configuration data stream to the SYSTEM.BROKER.ADMIN.QUEUE of each target broker. When the configuration data has been sent to all relevant brokers, control is returned to you.

If the first stage is successful, message BIP1520I is displayed identifying the brokers to which the data was deployed.

However, if an error is detected during the first stage of deployment, the deployment is abandoned: no configuration data is sent to any broker, and an appropriate error message is displayed in a Control Center dialog box.

Stage two of deployment

During stage two of the deployment process, which is asynchronous, the target brokers process the received configuration data and return a response on the Configuration Manager's SYSTEM.BROKER.ADMIN.REPLY queue. The Configuration Manager then updates its record of the deployed configuration.

Deployment of data to a target broker might be only partially successful. This is because the unit of deployment on a broker is the execution group: the deployment of one execution group to a broker might succeed, but the deployment of another to the same broker might fail. A unit of deployment is transactional, however, so either all changes are made to a given execution group or no change is made.

For deployment purposes, topics and topology data are considered to belong to a separate unit of deployment, so either all changes are made to both topics and topology, or no change is made.

Which data is deployed?

When a deployment of any type of configuration data takes place, the data of that type that has been checked into the shared configuration by all Control Center users in the broker domain is that which is deployed to the configuration repository. Data that has not been checked in is not deployed. Note also that descriptive text that you can supply when defining Control Center resources is not deployed.

If some data has not been checked in

If the fact that some data has not been checked in leaves the shared configuration in an inconsistent state, the deployment is likely to fail. If the Configuration Manager detects an inconsistency, you receive a message indicating that some Control Center resources are not checked in.

To help avoid this situation occurring, you can request a list of all resources in your workspace that have not been checked in (using the **File —>** **Check In List** action) before you deploy. You can also check in all checked-out configuration data in your workspace using the **File —>** **Save to Shared** action. Of course, if multiple users are creating shared configuration data, that activity must cease while a deployment takes place, and all users must check in any checked-out resources before the deployment is requested.

Finding out whether deployment has worked

You can find out whether stage two of a deployment has succeeded by refreshing the **Log** view: click the green refresh button on the taskbar, or select **View —>** **Refresh**. It might take a while for the response to arrive. The refreshed **Log** view displays a group of messages for each broker to which configuration data has been deployed. Typical messages are:

Message	**Meaning**
BIP2056	Indicates that a deployment was completely successful for the broker.
BIP2086	Indicates that a deployment was partially successful for the broker.
BIP2087	Indicates that a deployment was completely unsuccessful for the broker.

If a deployment fails completely or partially succeeds, and message BIP4046 also appears in the **Log** view, Topics or Topology data was not processed. In this case, the broker in question is out of step with the rest of the broker domain, and so you *must* correct the problem that caused the failure and deploy again to restore consistency of data throughout the broker domain. This might occur, for example, if you have deleted and recreated a broker. See "Deleting a broker from the broker domain" on page 217 for further details.

Refresh the **Operations** view of the Control Center to display the status of each broker after the deployment.

If deployment times out

It is possible for the deployment of an execution group to time out while it is being processed by the target broker. This effectively leaves the status of the execution group in doubt. This status is shown in the **Operations** view by the appearance of a yellow question mark over the traffic light status icon. A message in the **Log** view confirms the problem. The in-doubt status of the execution group can be resolved only by a subsequent deployment of *all* assignments data. (Note that a subsequent delta deployment is automatically converted to a complete deployment if any execution group is in the in-doubt state).

If the broker is not running

If a broker is not running when a deployment takes place, or an MQSeries queue manager on the route to the broker is not running, the deployment message is not processed immediately. Note, however, that the deployment message does not expire, so it will be processed eventually. You cannot perform a complete or delta deployment to a broker when a deployment of any type is outstanding to that broker: an attempt to do so returns an error message in a Control Center dialog box. Stage two of the deploy must complete before a further deploy is allowed, unless a forced deployment is requested.

Deleting a broker from the broker domain

When you delete a broker using the Control Center, the broker symbol is no longer visible in the **Assignments** view or the **Topology** view.

- If the broker has never been deployed to, the broker and all execution groups assigned to it are deleted immediately from the shared configuration repository.
- If the broker has been deployed to, its definition remains in the shared deployed configuration after the topology (with the broker deleted) has been checked in, until the updated configuration is deployed.
  - When configuration data of any type is next deployed after the broker is deleted, the Configuration Manager sends a configuration data stream requesting deletion of all data of the type relevant to that deployment request (which can be topology, topics, assignments, or all types) to the deleted broker.

    For example, if you request a delta deployment of topics data after having deleted a broker using the Control Center, the Configuration Manager constructs a configuration data stream to delete all topics data deployed to the deleted broker.
  - When all configuration data of all types has been successfully deleted in this way, which might take several deployment requests, the deleted broker is finally removed from both the shared and the deployed configurations.

  You can, of course, force early completion of this stage by requesting a complete or delta deployment of all types of data.

  When the broker has been removed from both the shared *and* the deployed configurations, it is removed from the **Operations** view when you next refresh that view.

  You are strongly recommended to perform the actions for deleting a broker that has been deployed to in the following order:
  - Delete the broker reference in the Control Center.
  - Deploy this change by selecting **File—**>**Deploy—**> **Complete Configuration (all types)—**>**Normal**. You cannot deploy just the topology data in this situation.
  - Delete the physical broker on its local system using the **mqsideletebroker** command.

  This order ensures that the deletion messages described above are processed by the broker on its local system before it is deleted and avoids potential inconsistency of deployed data.
- If you subsequently recreate a broker with the same name as a deleted broker, you **must** complete the necessary steps in the recommended order above, followed by these steps:
  - Recreate the physical broker on its local system using the **mqsicreatebroker** command, and recreate the broker reference in the Control Center.
  - Deploy your changes. Because the previous instance of this broker has been completely deleted, you can select either **File—**>**Deploy—**> **Complete Configuration (all types)—**>**Normal**. or **Deploy—**>**Complete Topology Configuration** or **Deploy—**>**Delta Topology Configuration** in

the Topology view (described in “Deploying delta topology” on page 227 and “Deploying complete topology” on page 228).

The act of deployment, recreation, and redeployment in the Control Center is required to reset the configuration repository and update internal identifiers for the broker that are generated by the create command. If you do not delete and recreate the broker, and redeploy all data to that broker, the identifiers will not match and check in or deployment will fail.

Authorization to deploy configuration data

To perform any of the tasks described in the remainder of this chapter, you must:

- Have the correct Control Center user role, as follows:
  - To deploy assignments only, you must have the user role **Message flow and message set assigner**, **Operational domain controller**, or **All roles**.
  - To deploy topics only, you must have the user role **Topic security administrator**, **Operational domain controller**, or **All roles**.
  - To deploy topology only, you must have the user role **Operational domain controller** or **All roles**.
  - To deploy all types of data, you must have the user role **Operational domain controller** or **All roles**.

  For information about setting your user role, see “Setting user roles” on page 13.
- Be a member of the appropriate MQSeries Integrator group, as follows:
  - To deploy assignments data, you must be a member of group **mqbrops**.
  - To deploy topics data, you must be a member of group **mqbrops** or group **mqbrtpic**.
  - To deploy topology data, you must be a member of group **mqbrops**.

Deploying delta data of all types

You can deploy delta data of all types from all views on the Control Center.

1. Ensure that the assignments, topics, and topology data you want to deploy has been checked into the shared configuration, as described in “Checking in the Assignments” on page 209, “Checking in topics data” on page 234, and “Checking in the Topology” on page 196.
2. Click **File —>** **Deploy —>** **Delta configuration (all types)**.

The Configuration Manager compares data of all types for all brokers in the shared configuration with the currently deployed data for all brokers, and deploys only the differences between the two versions.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying complete data of all types

You can deploy complete data of all types from all views on the Control Center.

1. Ensure that the assignments, topics, and topology data you want to deploy has been checked into the shared configuration, as described in “Checking in the Assignments” on page 209, “Checking in topics data” on page 234, and “Checking in the Topology” on page 196.
2. Click **File —>** **Deploy —>** **Complete configuration (all types) —>** **Normal**.

The Configuration Manager creates a request consisting of instructions to delete *all* deployed data of all types, followed by instructions to create a new set of data, based on the shared configuration, and deploys it to the target brokers.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Forcing deployment of all data

You can force deployment of complete data of all types from all views on the Control Center.

1. Ensure that the assignments, topics, and topology data you want to deploy has been checked into the shared configuration, as described in “Checking in the Assignments” on page 209, “Checking in topics data” on page 234, and “Checking in the Topology” on page 196.
2. Click **File —>** **Deploy —>** **Complete configuration (all types) —>** **Forced**.

The Configuration Manager creates a request consisting of instructions to delete *all* deployed data of all types, followed by instructions to create a new set of data, based on the shared configuration, and deploys it to the target brokers. Any outstanding deployment request, of any type, is overridden by this forced deployment of configuration data.

If you have deleted and recreated a broker, and have not followed the order of actions indicated in “Deleting a broker from the broker domain” on page 217, you might find it necessary to use a forced deployment to reset your broker domain configuration.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying delta assignments

You must be in the Assignments view to deploy only assignments data.

1. Ensure that the assignments data you want to deploy has been checked into the shared configuration, as described in “Checking in the Assignments” on page 209.
2. Select the objects to which you want to deploy the assignments data.

   If you are deploying to all brokers in the broker domain:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the root of the Broker tree. Select **Deploy—**>**Delta Assignments Configuration**.

   If you are deploying to a single broker:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the entry of the broker to which you want to deploy assignments data. Select **Deploy—**>**Delta Assignments Configuration**.

     Alternatively, you can right click the broker symbol in the Domain Topology pane and select **Deploy—**>**Delta Assignments Configuration**.

   If you are deploying to a single execution group:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the execution group to which you want to deploy assignments data. Select **Deploy—**>**Delta Assignments Configuration**.

     Alternatively, you can right click the execution group symbol in the Domain Topology pane and select **Deploy—**>**Delta Assignments Configuration**.
3. You can also invoke the **Deploy —**> **Delta Assignments Configuration** action from the Domain Hierarchy menu on the Control Center taskbar.

The Configuration Manager compares assignments data for the target brokers or execution groups in the shared configuration with the currently deployed assignments data for the same brokers, and deploys only the differences between the two versions.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying complete assignments

You must be in the Assignments view to deploy only assignments data.

1. Ensure that the assignments data you want to deploy has been checked into the shared configuration, as described in “Checking in the Assignments” on page 209.
2. Select the objects to which you want to deploy assignments data.

   If you are deploying to all brokers in the broker domain:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the root of the Broker tree. Select **Deploy—>Complete Assignments Configuration**.

   If you are deploying to a single broker:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the entry of the broker to which you want to deploy assignments data. Select **Deploy—>Complete Assignments Configuration**.

     Alternatively, you can right click the broker symbol in the Domain Topology pane and select **Deploy—>Complete Assignments Configuration**.

   If you are deploying to a single execution group:

   - In the Domain Hierarchy pane of the **Assignments** view, right click the execution group to which you want to deploy assignments data. Select **Deploy—>Complete Assignments Configuration**.

     Alternatively, you can right click the execution group symbol in the Domain Topology pane and select **Deploy—>Complete Assignments Configuration**.
3. You can also invoke the **Deploy —> Complete Assignments Configuration** action from the Domain Hierarchy menu on the Control Center taskbar.

The Configuration Manager creates a request consisting of instructions to delete *all* deployed assignments data, followed by instructions to create a new set of assignments data, based on the shared configuration, and deploys it to the target brokers.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying delta topics

You must be in the Topics view to deploy only topics data.

1. Ensure that the topics data you want to deploy has been checked into the shared configuration, as described in “Checking in topics data” on page 234.
2. In the Topics pane of the **Topics** view, right click TopicRoot.

   Select **Deploy —**> **Delta Topics Configuration**.
3. You can also invoke the **Deploy —**> **Delta Topics Configuration** action from the **Topics** menu on the Control Center taskbar.

The Configuration Manager compares topics data for all brokers in the shared configuration with the currently deployed topics data for all brokers, and deploys only the differences between the two versions.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying complete topics

You must be in the Topics view to deploy only topics data.

1. Ensure that the topics data you want to deploy has been checked into the shared configuration, as described in “Checking in topics data” on page 234.
2. In the Topics pane of the **Topics** view, right click TopicRoot.

   Select **Deploy —>** **Complete Topics Configuration**.
3. You can also invoke the **Deploy —>** **Complete Topics Configuration** action from the **Topics** menu on the Control Center taskbar.

The Configuration Manager creates a request consisting of instructions to delete *all* deployed topics data, followed by instructions to create a new set of topics data, based on the shared configuration, and deploys it to the target brokers.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying delta topology

You must be in the Topology view to deploy only topology data.

1. Ensure that the topology data you want to deploy has been checked into the shared configuration, as described in “Checking in the Topology” on page 196.
2. Choose one of the following options:
   - In the Domain Hierarchy pane of the **Topology** view, right click the root of the Topology tree. Select **Deploy —>** **Delta Topology Configuration**.

     You can also select **Deploy —>** **Delta Topology Configuration**. from the **Domain Hierarchy** menu on the Control Center taskbar.
   - In the Topology pane of the **Topology** view, right click the background and select **Deploy —>** **Delta Topology Configuration**.

     You can also select **Deploy —>** **Delta Topology Configuration** from the **Topology** menu on the Control Center taskbar.

The Configuration Manager compares topology data for all brokers in the shared configuration with the currently deployed topology data for all brokers, and deploys only the differences between the two versions.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Deploying complete topology

You must be in the Topology view to deploy only topology data.

1. Ensure that the topology data you want to deploy has been checked into the shared configuration, as described in “Checking in the Topology” on page 196.
2. Choose one of the following options:
   - In the Domain Hierarchy pane of the **Topology** view, right click the root of the Topology tree. Select **Deploy —>** **Complete Topology Configuration**.

     You can also select **Deploy —>** **Complete Topology Configuration**. from the **Domain Hierarchy** menu on the Control Center taskbar.
   - In the Topology pane of the **Topology** view, right click the background and select **Deploy —>** **Complete Topology Configuration**.

     You can also select **Deploy —>** **Complete Topology Configuration** from the **Topology** menu on the Control Center taskbar.

The Configuration Manager creates a request consisting of instructions to delete *all* deployed topology data, followed by instructions to create a new set of topology data, based on the shared configuration, and deploys it to the target brokers.

For information about checking the progress of this deployment request, see “Finding out whether deployment has worked” on page 215.

Chapter 8. Setting up publish/subscribe access control

This chapter describes how to create a new publish/subscribe topic, and how to update access control lists (ACLs). ACLs allow you to restrict user permission to publish messages, subscribe to topics, and request persistent delivery of messages.

Authorization to set up publish/subscribe access control

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Topic security administrator**
  - **All roles**

  For information about setting your use role, see "Setting user roles" on page 13.
- Be a member of the MQSeries Integrator group **mqbrtpic**

The Topics view

To display the **Topics** view, click the **Topics** tab in the Control Center. Figure 59 shows an example of the **Topics** view.

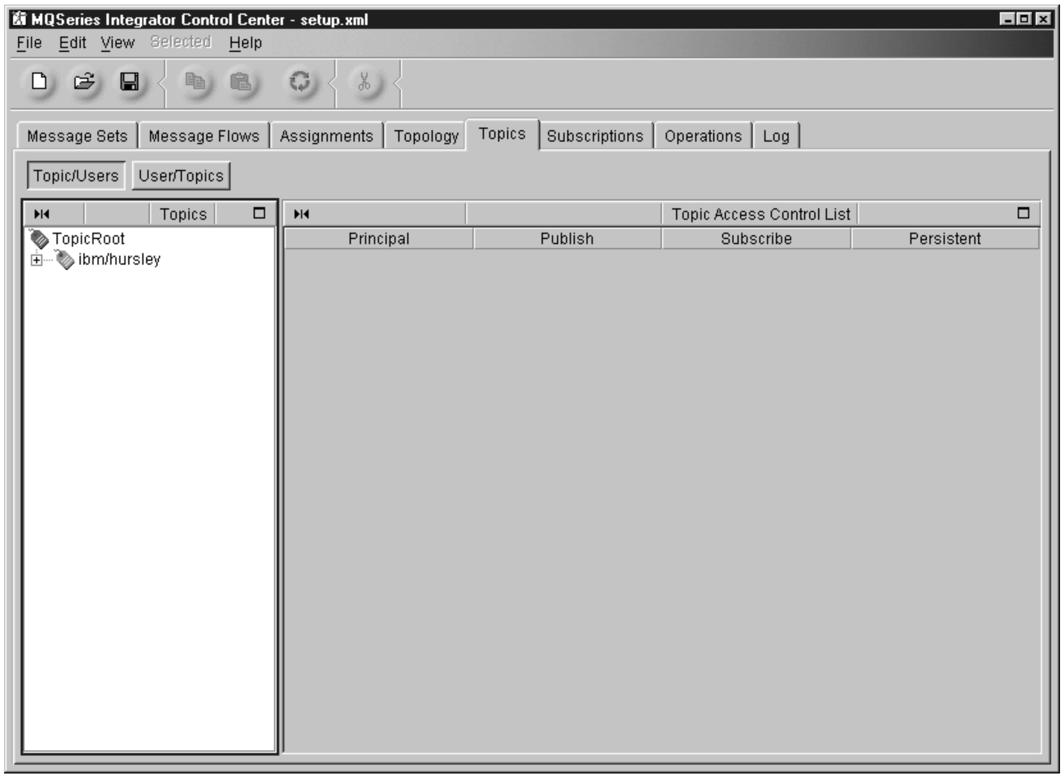


Figure 59. The Topics view.

In the **Topics** view, you can create the topics under which messages can be published. In addition, you can give users or groups permission to publish messages, or to subscribe to messages published under these topics. You can also deny users or groups these access rights. You would do this to ensure that privileged information was not being viewed by unauthorized users or groups, for example.

The information in the **Topics** view can be viewed in two ways:

- The hierarchy of topics is shown in the **Topics/Users** view, where the Access Control List (ACL) for the selected topic is shown.
- The list of users and groups is shown in the **Users/Topics** view, and the access to each topic is shown for the selected user or group.

In addition to the **Topics** view, you can use the **Subscriptions** view to see currently registered subscriptions if you are a member of MQSeries Integrator group **mqbrops**.

Creating topics

You create new topics beneath the TopicRoot,[8] which is always displayed in the Topics pane, or beneath any topic already defined. Any topic can have any number of children, and each of these can have different ACL settings.

To create a new topic:

1. Click the **Topics/Users** button in the **Topics** view.
2. Ensure that the topic under which you want to create a new one, which can be TopicRoot or any topic already defined, is checked out. If it is not checked out, right click the topic and click **Check Out**.
3. Right click the parent topic and click **Create —**> **Topic**.

   The **Create a new Topic** dialog is displayed.
4. In the **Create a new Topic** dialog, type the name of the topic in the Name field.
5. Select the users and groups that are to have explicit access defined for this topic. Note that this list contains users and groups (principals) only if you have a User Name Server installed and running, and the Configuration Manager is configured to communicate with it.

   If you do not select users or groups from the list on this dialog, access for this topic defaults to the ACL setting for PublicGroup for the topic root. You can update access to this topic for explicit users at a later time, if you do not do so now.

   To specify explicit users or groups access now, expand the Groups and Users folders and select the users or groups:

   - To select a single user or group from the list, click the user or group name.
   - To select multiple users or groups that appear sequentially in the list, click the first user or group you want, press and hold the Shift key, then click the last user or group you want. This action selects the two users or groups you highlighted, plus any that appear between these two in the list.
   - To select multiple users or groups that do not appear in a sequence in the list, hold down Ctrl and click each principal you want.
6. Select the required access setting for this topic. The values that you set apply to all users and groups that you selected in the create topic dialog (step 4).

   - For the Publish field, select one of

Allow	Publications are allowed.
Deny	Publications are not permitted.
Inherit	Permission to publish is inherited.

   - For the Subscribe field, select one of

Allow	Subscriptions are allowed.
Deny	Subscriptions are not permitted
Inherit	Permission to subscribe is inherited.

[8] TopicRoot is a special topic that cannot be deleted or renamed. It always has the PublicGroup in its ACL.

- For the Persistent field, select one of

Yes	Persistent delivery of messages is allowed.
No	Persistent delivery of messages is not allowed.
Inherit	Permission to request persistent delivery of messages is inherited.

7. Click **Finish**.

The new topic appears beneath its parent topic.

After you create a topic, you can add more users or groups to the ACL using the Properties dialog as described "Adding a principal to an ACL" on page 233.

If you do not select any users or groups when you create the topic, the ACL is empty, and the Topics Access Control List pane is left blank. In this case, each user or group inherits the same access to this topic as it has to the parent topic.

If you have selected users or groups, they appear in the Topic Access Control List pane. Beside the users or groups, you see the permissions they have to publish messages, subscribe to messages, and request persistent delivery of messages. You can change these permissions by selecting them. A drop down list is shown, allowing you to select a different permission.

Renaming, duplicating, and deleting topics

Topics can be renamed, duplicated, or deleted by right clicking the appropriate topic and selecting the desired action from the pop-up menu. When you duplicate a topic, a sibling topic with a unique name is created. Note that the parent topic must be checked out before you can perform any of these actions.

Adding a principal to an ACL

To add a principal[9] to an ACL:

1. In the **Topics** view, click the **Topics/Users** button.
2. Ensure that the topic for which you would like to edit the ACL is checked out. If it is not checked out, right click the topic and click **Check Out**.
3. Right click the topic and click **Properties**.
4. Expand the Groups or Users folders in the Available Principals.

   You can add principals that are not yet listed in the ACL; principals that are already in the ACL are not shown. You can grant permissions to a principal, or revoke permissions for a principal. You can specify that the principal inherit the same level of access to a permission as it has to the parent topic. Setting the access level of a principal in the ACL of the TopicRoot to Inherit is not allowed, since the TopicRoot does not have a parent topic. Each principal can be assigned the following permissions:

   Publish
   Permits or denies the principal permission to publish messages on this topic.

   Subscribe
   Permits or denies the principal permission to subscribe to messages on this topic.

   Persistent
   Permits or denies the principal permission to request persistent delivery of a publication when the principal subscribes to the topic.

If a user subscribes to a topic, and the user requests persistent delivery of the messages, the user must be granted permission both to subscribe to that topic and to request persistent delivery of messages for that topic. If the user does not request persistent delivery, only permission to subscribe to that topic is required.

Permission for persistent delivery does not affect the publishing of messages. You need only to be granted publish permissions to be able to publish messages on a topic.

To remove a entry from an ACL, right click the entry and click **Remove**.

Resolving permissions

Many factors play a part in determining whether the user has permission to publish messages on a topic, subscribe to messages under a topic, and to request persistent delivery of messages being subscribed to. The user can be explicitly listed in the topic's ACL. Groups to which the user belongs can also be listed, and their permissions may differ from each other and with the user's ACL entry. Users can also inherit permissions from parent topics. Determining whether the user has a permission might not always be straightforward.

For a complete description of how permissions are resolved, see the *MQSeries Introduction and Planning Guide*.

[9] A principal is a user or a group.

Checking in topics data

To check in topics data:

1. Right click the topic entry in the **Topics** view.
2. Click **Check in** to store the topics data in the shared configuration.

To confirm that the topics data has been checked in, the **New** icon or the **Key** icon disappears from the topic entry.

When you check in a new topic, its parent is also checked in. When you check in a parent topic, all new child topics are also checked in.

Checking in multiple changes

The **File —>** **Check In** menu option allows you to check in multiple changes that you have made in this or any other view. You can use this instead of checking in individual objects in the Topics view. The options are:

- **File —> Check In —> List**
- **File —> Check In —> All in Current Workspace**
- **File —> Check In —> All (Save to Shared)**

These options are more efficient when you have many different resources checked out. The List option also allows you to check which resources are checked out in your current workspace before you decide which resources to check in.

For more information about check in options, see “Saving the workspace to the shared repository” on page 17.

Making changes operational

When you check in resources that are new, or that you have altered, you make them visible in the shared configuration. However, the changes you have made have no operational effect until you *deploy* them in the broker domain. For information about deploying resources, see Chapter 7, “Deploying configuration data” on page 213.

Chapter 9. Running the broker domain

This chapter describes the **Operations** view and Control Center tasks that are related to running the operational broker domain. These are:

- "Monitoring the operational state of the broker domain" on page 239
- "Starting message flows" on page 240
- "Stopping message flows" on page 242
- "Starting user tracing" on page 244
- "Stopping user tracing" on page 245
- "Deleting subscriptions" on page 247

It also describes:

- The **Subscriptions** view, and how you can monitor and manage subscriptions, in "The Subscriptions view" on page 246.
- The **Log** view, and how you can manage the log contents, in "The Log view" on page 248.

Authorization to run the broker domain

To perform any of the tasks described in this chapter, you must:

- Have the correct Control Center user role, which can be one of:
  - **Operational domain controller**
  - **All roles**

  For information about setting your user role, see "Setting user roles" on page 13.
- Be a member of the MQSeries Integrator group **mqbrops**

The Operations view

To display the **Operations** view, click the **Operations** tab in the Control Center. Figure 60 on page 238 shows an example of the **Operations** view.

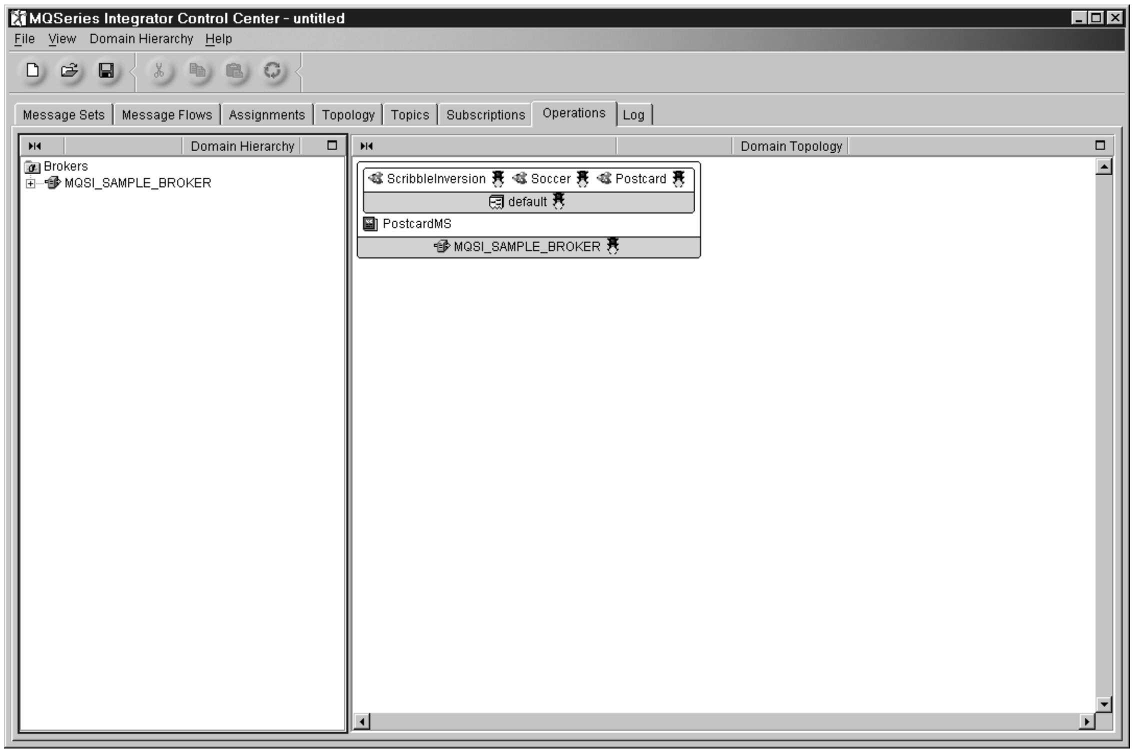


Figure 60. The Operations view. The left-hand pane, the Domain Hierarchy pane, shows a tree view of the brokers in your broker domain. The execution groups and message sets assigned to a broker are displayed when you expand the broker. The message flows assigned to an execution group are displayed when you expand the execution group. The right-hand pane, the Domain Topology pane, contains an arrangement of graphical symbols that represent the current broker domain. Execution groups and message sets appear inside the brokers to which they have been assigned. Message flows appear inside the execution groups to which they have been assigned. The brokers shown in the Operations view are those to which configuration data has been deployed.

Monitoring the operational state of the broker domain

To display the current status of the broker domain, you must refresh the view. You can either click the green circular refresh icon below the taskbar, or select **View —> Refresh**. This causes the Configuration Manager to update the information displayed in the Domain Topology pane from its deployed configuration.

Any resource shown in the Domain Topology pane of the **Operations** view can be in one of three states:

Started	Indicated by a green traffic light next to the resource.
Stopped	Indicated by a red traffic light next to the resource.
Unknown	Indicated by a yellow question mark next to the resource.

If you initiate a complete deployment, the operational state of all resources is reset. Therefore, if, for example, you have stopped an individual message flow, or you have started user trace, you will have to reissue this requests.

Starting message flows

You can start:

- All message flows in all execution groups assigned to a specified broker
- All message flows in a specified execution group
- A single message flow

Starting all message flows for a broker

To start all message flows in all execution groups assigned to a specified broker:

1. Right click the broker symbol in the Domain Topology pane or the broker entry in the Domain Hierarchy pane. (Alternatively, you can highlight the broker in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Start Message Flows**.

   The Configuration Manager sends a configuration message to the broker requesting that all message flows be started.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, all message flows within the broker have a green status light against them.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Starting all message flows within an execution group

To start all message flows in an execution group:

1. Right click the execution group symbol in the Domain Topology pane or the execution group entry in the Domain Hierarchy pane. (Alternatively, you can highlight the execution group in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Start Message Flows**.

   The Configuration Manager sends a configuration message to the broker requesting that all message flows within the specified execution group be started.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, all message flows within the execution group have a green status light against them.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Starting a single message flow

To start a single message flow:

1. Right click the message flow symbol in the Domain Topology pane or the message flow entry in the Domain Hierarchy pane. (Alternatively, you can highlight the message flow in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Start**.

   The Configuration Manager sends a configuration message to the broker requesting that the specified message flow be started.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, the message flow has a green status light against it.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Stopping message flows

You can stop:

- All message flows in all execution groups assigned to a specified broker
- All message flows in a specified execution group
- A single message flow

Stopping all message flows for a broker

To stop all message flows in all execution groups assigned to a specified broker:

1. Right click the broker symbol in the Domain Topology pane or the broker entry in the Domain Hierarchy pane. (Alternatively, you can highlight the broker in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Stop Message Flows**.

   The Configuration Manager sends a configuration message to the broker requesting that all message flows be stopped.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, all message flows within the broker have a red status light against them.

   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Stopping all message flows within an execution group

To stop all message flows in an execution group:

1. Right click the execution group symbol in the Domain Topology pane or the execution group entry in the Domain Hierarchy pane. (Alternatively, you can highlight the execution group in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Stop Message Flows**.

   The Configuration Manager sends a configuration message to the broker requesting that all message flows within the specified execution group be stopped.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, all message flows within the execution group have a red status light against them.

   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Stopping a single message flow

To stop a single message flow:

1. Right click the message flow symbol in the Domain Topology pane or the message flow entry in the Domain Hierarchy pane. (Alternatively, you can highlight the message flow in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **Stop**.

   The Configuration Manager sends a configuration message to the broker requesting that the specified message flow be stopped.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, the message flow has a red status light against it.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Starting user tracing

You can start user tracing:

- For all message flows in a specified execution group
- For a single message flow

The user tracing function of MQSeries Integrator is described in the *MQSeries Integrator Administration Guide*. Refer to this for information about the levels of tracing that can be started (normal and debug), and for information on how to format and read the output.

Starting user tracing for an execution group

To start user tracing of all message flows in an execution group:

1. Right click the execution group symbol in the Domain Topology pane or the execution group entry in the Domain Hierarchy pane. (Alternatively, you can highlight the execution group in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **User Trace —**> **Normal** or **User Trace —**> **Debug**.

   The Configuration Manager sends a configuration message to the broker requesting that user tracing be started for all message flows within the specified execution group.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, the execution group has an icon against it indicating that user tracing is active.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Starting user tracing for a single message flow

To start user tracing for a single message flow:

1. Right click the message flow symbol in the Domain Topology pane or the message flow entry in the Domain Hierarchy pane. (Alternatively, you can highlight the message flow in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **User Trace —**> **Normal** or **User Trace —**> **Debug**.

   The Configuration Manager sends a configuration message to the broker requesting that user tracing be started for the specified message flow.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in “Monitoring the operational state of the broker domain” on page 239. If the request was successful, the message flow has an icon against it indicating that user tracing is active.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Stopping user tracing

The user tracing function of MQSeries Integrator is described in the *MQSeries Integrator Administration Guide*. You can stop user tracing:

- For all message flows in a specified execution group
- For a single message flow

Stopping user tracing for an execution group

To stop user tracing of all message flows in an execution group:

1. Right click the execution group symbol in the Domain Topology pane or the execution group entry in the Domain Hierarchy pane. (Alternatively, you can highlight the execution group in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **User Trace —**> **None**.

   The Configuration Manager sends a configuration message to the broker requesting that user tracing be stopped for all message flows within the specified execution group.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in "Monitoring the operational state of the broker domain" on page 239. If the request was successful, any user tracing icon against the execution group has disappeared.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

Stopping user tracing for a single message flow

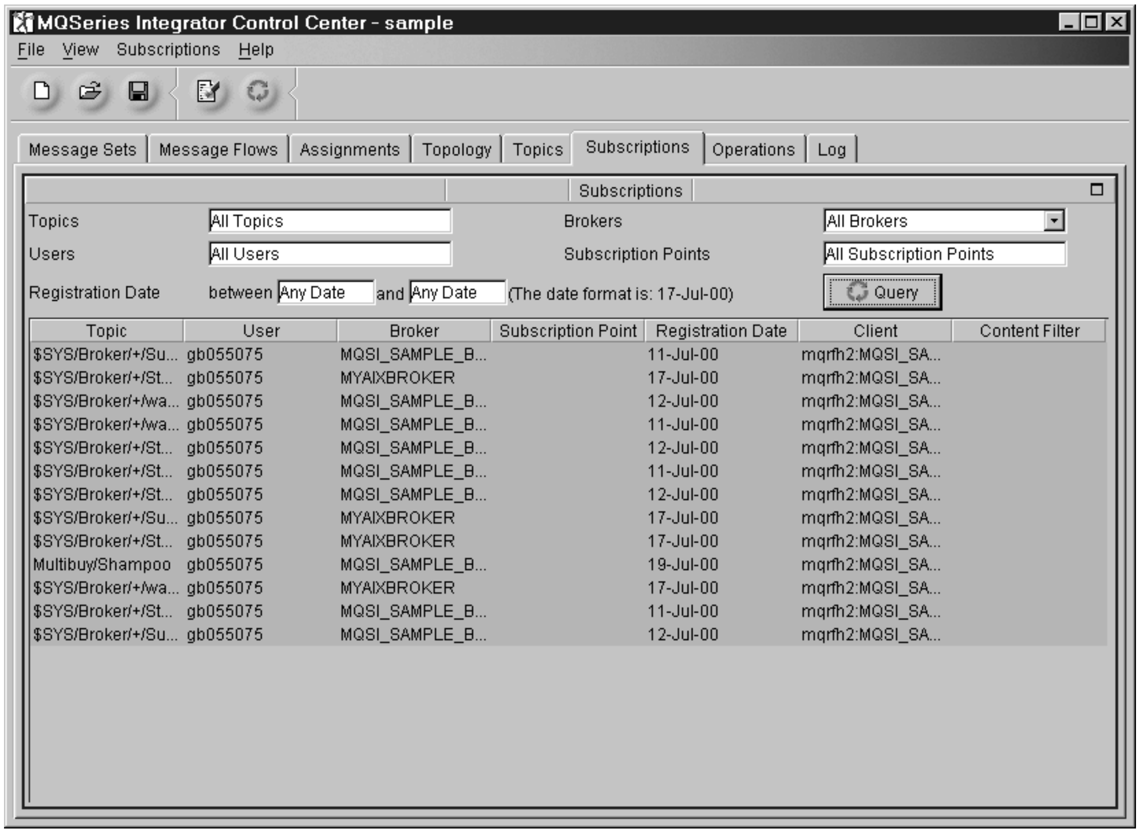
To stop user tracing for a single message flow:

1. Right click the message flow symbol in the Domain Topology pane or the message flow entry in the Domain Hierarchy pane. (Alternatively, you can highlight the message flow in either pane, then click the Domain Hierarchy or Domain Topology menu in the taskbar.)
2. Click **User Trace —**> **None**.

   The Configuration Manager sends a configuration message to the broker requesting that user tracing be stopped for the specified message flow.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Operations** view, as described in "Monitoring the operational state of the broker domain" on page 239. If the request was successful, any user tracing icon against the message flow has disappeared.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this request are displayed here.

The Subscriptions view

You use the **Subscriptions** view to monitor subscriptions to topics taken out by the applications running in your broker domain. Figure 61 shows an example of the **Subscriptions** view.



Subscriptions

Topics	All Topics	Brokers	All Brokers
Users	All Users	Subscription Points	All Subscription Points
Registration Date	between Any Date and Any Date (The date format is: 17-Jul-00)		

Topic	User	Broker	Subscription Point	Registration Date	Client	Content Filter
$SYS/Broker/+/Su...	gb055075	MQSI_SAMPLE_B...		11-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MYAIXBROKER		17-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/wa...	gb055075	MQSI_SAMPLE_B...		12-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/wa...	gb055075	MQSI_SAMPLE_B...		11-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MQSI_SAMPLE_B...		12-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MQSI_SAMPLE_B...		11-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MQSI_SAMPLE_B...		12-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/Su...	gb055075	MYAIXBROKER		17-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MYAIXBROKER		17-Jul-00	mqrfh2:MQSI_SA...	
Multibuy/Shampoo	gb055075	MQSI_SAMPLE_B...		19-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/wa...	gb055075	MYAIXBROKER		17-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/St...	gb055075	MQSI_SAMPLE_B...		11-Jul-00	mqrfh2:MQSI_SA...	
$SYS/Broker/+/Su...	gb055075	MQSI_SAMPLE_B...		12-Jul-00	mqrfh2:MQSI_SA...	

| *Figure 61. The Subscriptions view. Subscriptions owned by the brokers in this broker domain are shown in this view*
| *in a tabular form. Each subscription occupies one row in the table. For each subscription, the Topic, User, Broker,*
| *Subscription Point, Registration Date, Client, and Content Filter are displayed. Fields at the top of the view support*
| *filtering of information. The entries in the list of subscriptions are not timestamped, and are not ordered.*

Filtering the information in the Subscriptions view

Within any broker domain there can be many hundreds of active subscriptions. You are unlikely to want to view information relevant to all of these subscriptions at any one time. Therefore, the **Subscriptions** view allows you to select the information you are interested in by specifying a filter. You can filter the information displayed in the **Subscriptions** view by specifying any combination of:

- Brokers
- Topics
- Users
- Registration date
- Subscription points

For example, you can restrict the information displayed to particular topics within a single broker.

To filter the information by broker:

1. Click the **Brokers** drop-down list and click the broker name.
2. Refresh the **Subscriptions** view by clicking **Query**, clicking the green refresh icon below the taskbar, or selecting **View —>** **Refresh**.

The **Subscriptions** view is refreshed to display information for the selected broker.

To filter information by any other value, simply enter data in the appropriate field in the view. For example, to filter by Topic, enter the topic name in the **Topics** field, and refresh the **Subscriptions** view as described above. The wildcard character (%) can be used to represent any number of characters in the topic, user, and subscription point values.

To clear all data from the table, click the clear table icon next to the refresh icon on the taskbar. This action does not delete subscriptions; it simply clears the data from the **Subscriptions** view.

Refreshing the Subscriptions view

The **Subscriptions** view displays a snapshot of all current subscriptions in the broker domain, filtered by the current filter. The Configuration Manager updates its record of the deployed configuration whenever a subscription is created, changed, deleted, or expires. However, the **Subscriptions** view is not updated automatically to reflect these changes. You have to request that the **Subscriptions** view is refreshed by clicking **Query**, or by clicking the green refresh icon on the taskbar, or by selecting **View —>** **Refresh**.

Deleting subscriptions

To delete (deregister) a subscription from the deployed configuration:

1. In the **Subscriptions** view, select the subscriptions that you want to delete:
   a. To select a single subscription, click the row pertaining to that subscription.
   b. To select multiple rows that appear in a sequence in the table, click the first row you want to delete, press and hold the Shift key, then click the last row you want. This action selects the two rows you highlighted, plus any that appear between these two in the table.
   c. To select multiple rows that do not appear in a sequence in the table, hold down Ctrl and click each row you want.
2. From the **Subscriptions** menu in the taskbar, click **Delete**.
3. To monitor the outcome of this request, after a suitable delay:
   a. Refresh the **Subscriptions** view, as described in “Refreshing the Subscriptions view.” If the subscription has been successfully deleted, its entry is no longer included in the **Subscriptions** view.
   b. Refresh the **Log** view. Any messages returned by the broker in response to this deletion request are displayed here.

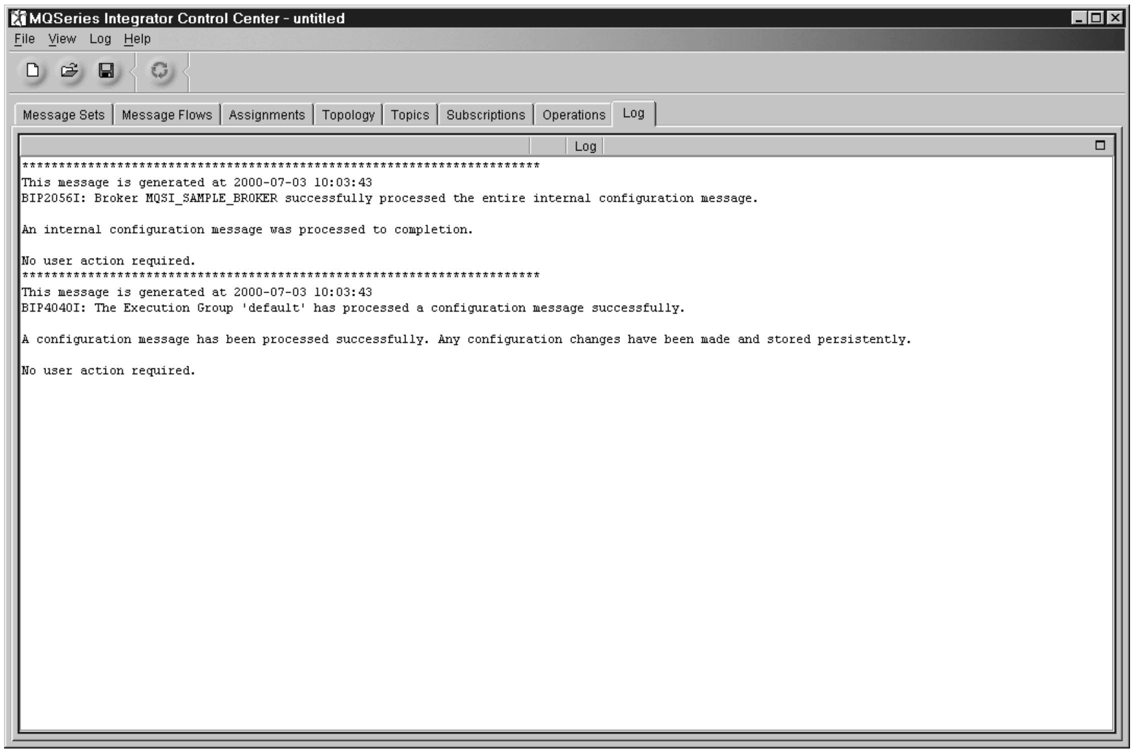
   Note that some subscriptions (specifically those used internally by the broker and the Configuration Manager) cannot be deleted. Any request to delete such a subscription fails.

The Log view

To display the contents of the **Log** view, click the **Log** tab in the Control Center. Figure 62 shows an example of the **Log** view.

The Log view is accessible unless you suppress it using the options on the Control Center Preferences dialog (see “Setting user roles” on page 13). The Log view shows messages associated with your user ID, and those that have no associated user ID.

When you change to the Log view, you must refresh the view to see the latest information available. You can request a refresh using the refresh button on the taskbar, the **View** menu, or the popup menu within the message pane.



```
**********************************************************************
This message is generated at 2000-07-03 10:03:43
BIP2056I: Broker MQSI_SAMPLE_BROKER successfully processed the entire internal configuration message.

An internal configuration message was processed to completion.

No user action required.
**********************************************************************
This message is generated at 2000-07-03 10:03:43
BIP4040I: The Execution Group 'default' has processed a configuration message successfully.

A configuration message has been processed successfully. Any configuration changes have been made and stored persistently.

No user action required.
```

Figure 62. The Log view. The Log view displays messages returned to you by the Configuration Manager in response to requests that update the broker domain configuration. It also displays messages relating to deployment requests and to requests to delete subscriptions.

You can perform the following tasks from the **Log** view, by right-clicking anywhere within the Log pane to display the pop-up menu:

- **Save Log As**, which saves the Log view in a file

  When you have saved the log to a filename and location of your choice, the saved records remain in the view. They are not removed unless you select the **Clear Log** action. You can view the saved log file, and print its contents if you want.

  You can use a text editor to open and work with a saved log file: you cannot open it in the Control Center Log view.

- **Clear Log**, which removes messages

  This menu option implements two actions:

  1. All messages are removed from the Log view.
  2. All messages specific to your user ID are removed from the Configuration Manager's database.

  If you select this option, you cannot retrieve any of the messages cleared from the Log. You are recommended to save the log if you want to preserve some or all of its content.

  If you do not clear messages from the Configuration Manager's database using the Clear function, they are automatically cleared after 72 hours to ensure that the database is not filled.

- **Refresh**, which adds any new messages to the Log view

  The Refresh action does not remove or overwrite existing messages in the Log view. Therefore if you frequently Refresh without clearing or saving the Log, the messages will build up in the local view and within the Configuration Manager's database.

Part 3. Appendixes

Appendix A. An example scenario

This appendix describes one way in which you can implement the retail scenario that is described in Chapter 3 of the *MQSeries Integrator Introduction and Planning* book.

You should use the information in this appendix in conjunction with the information in the rest of this book.

The whole message flow is shown in Figure 63:

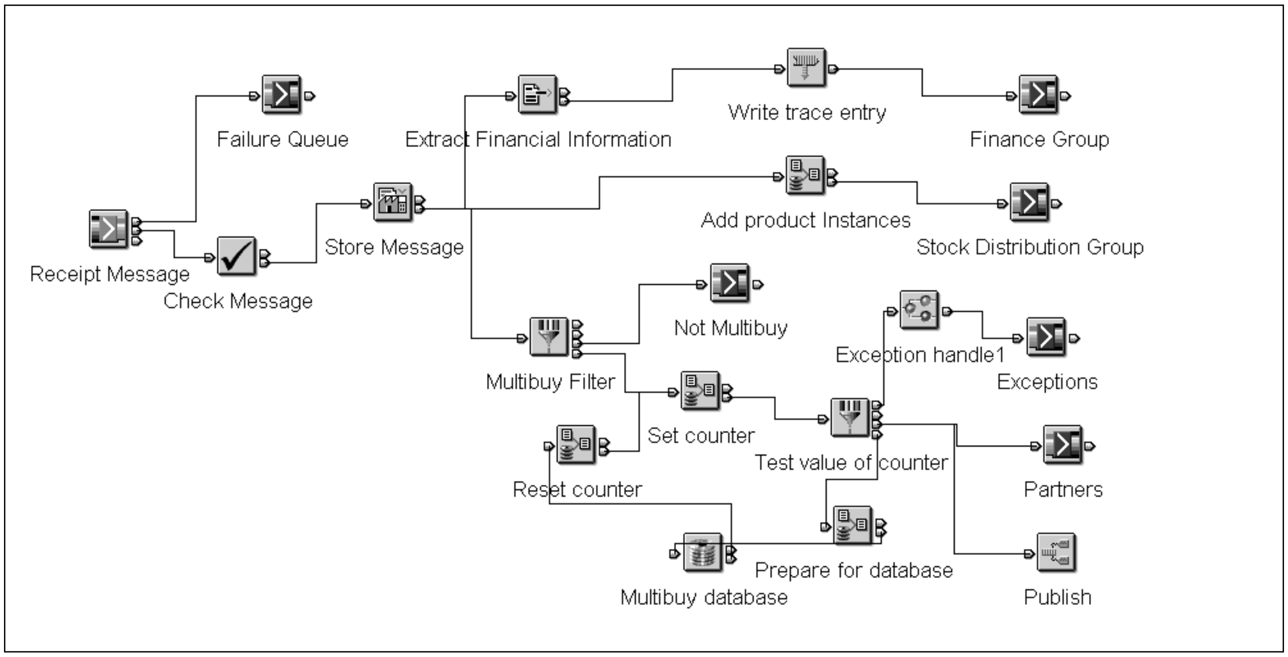


Figure 63. Scenario message flow

The following sections look at the messages and message flows necessary to fulfil the business requirements of the scenario.

You can work through the scenario either with a self-defining XML message or with a message defined in the MQSeries Integrator message repository. The nodes used in the scenario differ slightly depending on which type of message you use.

The following sections describe the different messages (self-defining XML message and message sets) used and how these messages are used in configuring the nodes within the message flow.

The receipt message as an XML message

A self-defining XML message can be passed through a message flow without having to be defined as part of a message set defined to the message repository through the Control Center. However, you cannot use some of the nodes without having a message repository message set.

Figure 64 shows an example of the type of generic XML message used:

```
<Message>
 <receiptmsg>
  <transactionlog>
   <storedetailselement>
    <storename>SRUCorp</storename>
    <branchnum>9</branchnum>
    <cashiernum>05</cashiernum>
    <tillnum>09</tillnum>
    <date>01/04/99</date>
    <time>14:30</time>
   </storedetailselement>
   <purchaseselement>
    <itemname>Shampoo</itemname>
    <itemcode>00056734097</itemcode>
    <itemprice>2.99</itemprice>
    <itemquantity>1</itemquantity>
   </purchaseselement>
   <purchaseselement>
    <itemname>Shampoo</itemname>
    <itemcode>00056734097</itemcode>
    <itemprice>2.99</itemprice>
    <itemquantity>1</itemquantity>
   </purchaseselement>
   <purchaseselement>
    <itemname>Toothpaste</itemname>
    <itemcode>0005663548</itemcode>
    <itemprice>1.99</itemprice>
    <itemquantity>1</itemquantity>
   </purchaseselement>
   <totalselement>
    <totalitems>10</totalitems>
    <multibuy>No</multibuy>
    <totalsales>34.98</totalsales>
    <change>5.02</change>
   </totalselement>
  </transactionlog>
 </receiptmsg>
</Message>
```

Figure 64. XML message

Defining the message in the message repository

The logical structure and the physical structure (the wire format) of the message need to be defined to the message repository using the Control Center. This section provides a systematic example that shows you how to create a message for the receipt data. It shows you how to create a message using the bottom-up approach but there is nothing to stop you using a top-down approach.

The message set you create will contain two messages called Receipt Message and Stock Message. The purpose of the receipt message is to take information from a shop receipt and, through the message flow, feed the information to the people who need it. For example, a financial controller needs to know the sales figures from each branch. The Stock Message is used to illustrate how information from one message can be modified and mapped into another message. For example, the stock controller needs to know the total quantity of a particular item per receipt. The Compute node in the Stock flow adds the number of items and puts that value into the Stock Message.

The message uses structured compound elements that you populate with simple elements. Each of these elements defines a unit of information.

Refer to “Creating message sets” on page 57 and “Creating messages” on page 59 for details on how to perform each of the tasks below. This appendix focuses on how you set up the properties to make this example work.

1. Create a message set.

   Give this message set any name. In our example, it is called Receipt Messages. Check that the parser on the Run Time tab is set to MRM. When you click Finish, MQSeries Integrator assigns the message set a unique identifier and writes this into the Identifier field of the message set properties. This is the identifier you will need to name on either the MQInput node or in the MQRFH2 header.

2. Create simple elements: the lowest-level units of information. You can give them any name and identifier you want. Table 28 on page 260 summarizes the simple elements, the type selected for each one, and the name and identifier used in our example.

   The XML descriptor tag for the element in the message must match the identifier used in the definition. For example, the element Store Name has an identifier storename and is represented in the message as <storename>.

   Note that, for the elements Date and Time, after you click Finish on the element property pages but before you move on to the next element, you should go to the COBOL tab and change the default settings of the COBOL Language Name property from DATE and TIME to something else. (Using a COBOL keyword in these fields is not permitted.)

Table 28. Simple elements, types, names, and identifiers

Simple element name	Identifier	Type
Store Name	storename	STRING
Branch Number	branchnum	INTEGER
Cashier Number	cashiernum	INTEGER
Till Number	tillnum	INTEGER
Date	date	STRING
Time	time	STRING
Item Name	itemname	STRING
Item Code	itemcode	INTEGER
Item Price	itemprice	FLOAT
Item Quantity	itemquantity	INTEGER
Total Items	totalitems	INTEGER
Multibuy	multibuy	STRING
Total Sales	totalsales	FLOAT
Change	change	FLOAT
Total Item Quantity	totalitemquantity	INTEGER

3. Create element lengths for elements of type STRING.

   You can give them any name and identifier you want. Table 29 summarizes the STRING elements, the length defined for each one, and the name and identifier used in our example.

Table 29. STRING elements, lengths, names, and identifiers

Element name	Element length name	Maximum Length	Element length identifier
Store Name	Store Name Length	20	storenamelen
Date	Date Length	10	datelen
Time	Time Length	10	timelen
Item Name	Item Name Length	40	itemnamelen
Multibuy	Multibuy Length	5	multibuylen

4. Add the lengths to the corresponding string elements. For example, add Store Name Length to the element Store Name.
5. Create element valid values for some of the elements. You can give them any name and identifier you want. Type must be the same type of the element that the valid value is associated with. Table 30 on page 261 summarizes the INTEGER elements, the minimum and maximum valid value defined for each one, and the name and identifier used in our example.

Table 30. INTEGER elements, values, names, and identifiers

Element name	Element valid value name	Element valid value identifier	Type	Minimum Valid Value	Maximum Valid Value
Branch Number	Branch Number Value	branchnumval	INTEGER	00000000	99999999
Cashier Number	Cashier Number Value	cashiernumval	INTEGER	000	999
Till Number	Till Number Value	tillnumval	INTEGER	000	999

6. Add the valid values to the corresponding elements. For example, add Branch Number Value to the element Branch Number.
7. Create compound types. These will be used as the type for compound elements (higher-level elements) within the message. Transactionlog will be used as the type for the message itself, thereby bringing all the lower-level structures together. You can give them any name and identifier you want. Table 31 summarizes the compound types, names, and identifiers used in our example.

Table 31. Compound type names and identifiers

Compound type name	Identifier
Store Details	storedetails
Purchases	purchases
Totals	totals
Transaction Log	transactionlog
Output Transaction Log	outputtransactionlog

8. Add elements to the compound types. (Leave *transactionlog* and *outputtransactionlog* for now.) The order of elements in the message being passed through the message flow **must** match the order of elements in the message definition. This order is defined by the order of elements in the compound types. When you add elements to a compound type, they are added in reverse order. For example, selecting Store Name then Branch Number will produce an order of Branch Number then Store Name. There is a Reorder option on the Types pulldown to resequence the elements within a type. To match the message shown in Figure 65 on page 264, add the elements in the sequence shown in Table 32 on page 262. Use Ctrl+left-click to select multiple elements.

Table 32. Elements to add to compound types

Compound type	Elements to be added
Store Details	• Time • Date • Till Number • Cashier Number • Branch Number • Store Name
Purchases	• Item Quantity • Item Price • Item Code • Item Name
Totals	• Change • Total Sales • Multibuy • Total Items • Total Item Quantity

9. Create elements with compound types. These elements bring together a number of lower-level elements. Because you added the simple elements to the compound type, when you create the compound element, those simple elements are automatically associated with it. You can give them any name and identifier you want. Table 33 summarizes the compound element names, types, and identifiers used in our example.

   The XML descriptor tag for the element in the message must match the identifier used in the message repository definition. For example, the element Store Details Element has an identifier storedetailselement and is represented in the message as <storedetailselement>.

Table 33. Compound type names, identifiers, and types

Compound element name	Identifier	Type
Store Details Element	storedetailselement	storedetails
Purchases Element	purchaseselement	purchases
Totals Element	totalselement	totals

10. Add the compound elements Totals Element, Purchases Element, and Store Details Element (in that order) to the compound type Transactionlog. This pulls all the elements of the receipt message together in a single type.
11. Add the elements Total Item Quantity, Purchases Element, Time, Date, Branch Number and Store Name (in that order) to the compound type outputtransactionlog. This pulls all the elements of the stock message together in a single type.
12. Create a message of type transactionlog. Give it any name or identifier you like. In our example, the message name is Receipt Message and the identifier is receiptmsg. The identifier is the one that you will need to name on either the MQInput node or in the MQRFH2 header.

13. Create a message of type outputtransactionlog. Give it any name or identifier you like. In our example, the message name is Stock Message and the identifier is stockmsg. The identifier is the one that you will need to name on either the MQInput node or in the MQRFH2 header.
14. Make the Purchases Element a repeating element. Make sure that the types transactionlog and outputtransactionlog are checked out. In the Receipt Message, check out the Purchases Element and open its properties pages. On the Connection tab, change **Repeat** to yes. Click Apply. Repeat this step for the Stock Message.
15. Create a category to contain the messages. This is optional but might be useful if you want to experiment with the functions to generate documentation about the message set. You can give the category any name and identifier you like. In our example, we used a category name of Transaction Log Messages and an identifier of transactionlogmsgs.

    Add the receipt message and the stock message to the category.
16. Save the definitions to the shared repository. (Select **File —> Check In — All(Save to Shared)**.)

Associating the receipt message with a message repository definition

When a message coming into the MQInput node has a corresponding definition in the message repository, you have to associate the incoming message with that definition. MQSeries Integrator needs to know which parser you are expecting to use for the message (called the message domain), which message set the message belongs to (called the message set) and which is the identifier of the message definition (called the message type).

There are two ways of doing this:

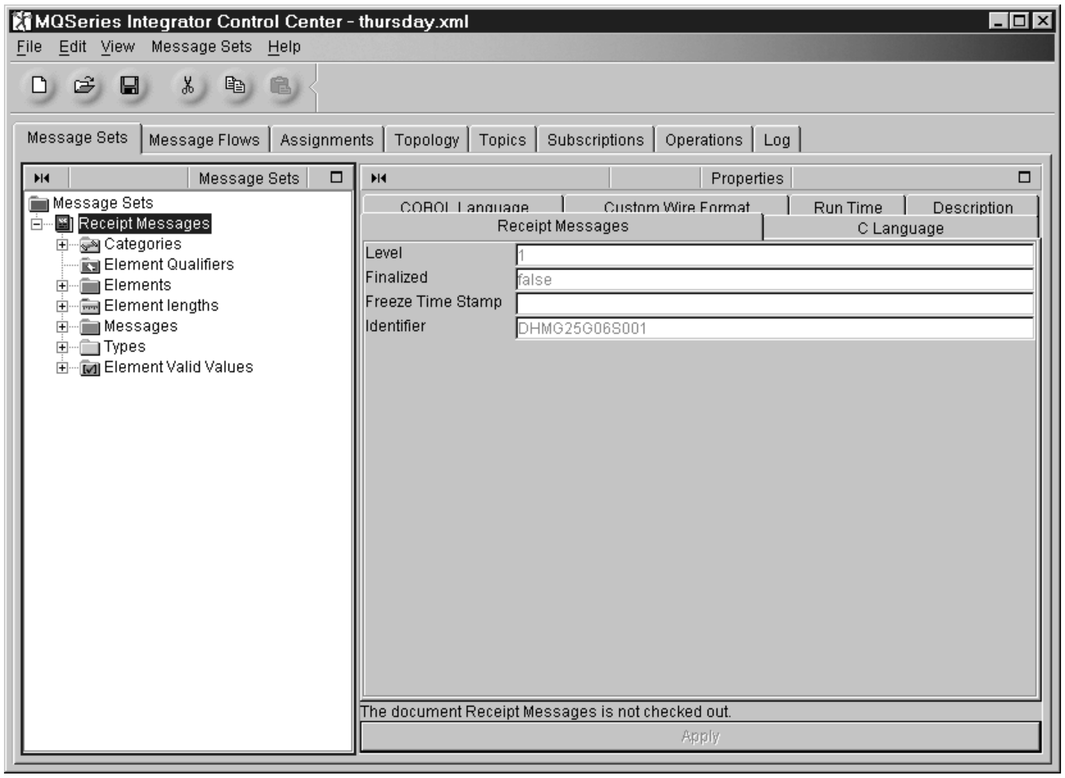
1. Define the message domain, message set, and message type on the **Default** tab of the MQInput node. See Figure 67 on page 267.
2. Define the message domain, message set, and message type on the NAMEVALUEDATA part of an MQRFH2 header.

Figure 66 on page 265 shows the receipt message defined in the previous section extended with an MQRFH2 header. The mcd folder within the message provides basic definition information, the psc folder contains information specific to publications.

- mcd folder

  Msd The parser to be used for this message. This is MRM in our example. Other values are BLOB, XML, and NEON. It must be entered in uppercase.

  Set The identifier of the message set to which the message belongs. This is the identifier assigned by MQSeries Integrator when you create the message set in the Control Center. In our example, this is DHMG25G06S001 (see Figure 65 on page 264).



Level: 1
Finalized: false
Freeze Time Stamp:
Identifier: DHMG25G06S001

The document Receipt Messages is not checked out.

Figure 65. The message set properties, showing the identifier

Type The identifier of the message definition to which this message maps. It is the identifier you assign when you define the message in the Control Center. In our example, this is receiptmsg. You cannot copy and paste the identifier from the message properties in the Control Center so be sure to enter it exactly as shown there.

Fmt This is the custom wire format of the message. In our example, it is XML. Other possible values are CWF and PDF.

- psc folder

Command Specifies that the message is to be published.

Topic Specifies the topic for publication.

Figure 66 on page 265 illustrates the receipt message extended with an MQRFH2 header.

```
<mcd><Msd>MRM</Msd><Set>DHMG25G06S001></Set><Type>receiptmsg</Type><Fmt>XML</Fmt></mcd>
<psc><Command>Publish</Command><Topic>Multibuy</Topic></psc>
<?xml version="1.01?>
<!DOCTYPE MRM PUBLIC "www.mrmnames.net/DHMG25G06S001" "DHMG25G06S001">
<MRM>
<receiptmsg>
  <storedetailselement>
    <storename>SRUCorp</storename>
    <branchnum>9</branchnum>
    <cashiernum>05</cashiernum>
    <tillnum>09</tillnum>
    <date>01/04/00</date>
    <time>14:30</time>
  </storedetailselement>
  <purchaseselement>
    <itemname>Shampoo</itemname>
    <itemcode>0005663548</itemcode>
    <itemprice>1.99</itemprice>
    <itemquantity>1</itemquantity>
  </purchaseselement>
  <purchaseselement>
    <itemname>Shampoo</itemname>
    <itemcode>0005663548</itemcode>
    <itemprice>1.99</itemprice>
    <itemquantity>1</itemquantity>
  </purchaseselement>
  <purchaseselement>
    <itemname>Toothpaste</itemname>
    <itemcode>0005663548</itemcode>
    <itemprice>1.99</itemprice>
    <itemquantity>1</itemquantity>
  </purchaseselement>
  <totalselement>
    <totalitems>10</totalitems>
    <multibuy>Yes</multibuy>
    <totalsales>13.49</totalsales>
    <change>5.02</change>
  </totalselement>
</receiptmsg>
</MRM>
```

Figure 66. The receipt message extended with an MQRFH2 header.

Assigning the message set to the broker

For the message flow to process a message that has a definition in the message repository (that is, an MRM message set) you have to assign the message set to the broker.

In the Assignments view, check out the broker. Drag and drop the message set (Receipt Messages) onto the name of the broker in the graphic in the Domain Topology pane. Check in the broker when you have finished these updates. The message set information is sent to the broker in the form of a message dictionary when you deploy your changes (see “Deploying the configuration” on page 286 for information about deploying).

Message flows

The scenarios makes use of three separate message flows:

1. The main flow.

   The main flow has one input node and many output nodes. It is configured to provide different back-end systems with different data, according to their requirements:

   - The **Audit** branch of the flow (described in “Audit flow” on page 268) is used to check that the incoming message belongs to the expected message set, and is therefore valid to be processed by the rest of the flow. If the message is valid, it also stores the receipt information in a database for later retrieval. See “Checking the message (predefined message only)” on page 268 and “Storing the entire message” on page 269.
   - The **Finance** branch of the flow (described in “Finance flow” on page 270) extracts some information from the message to suit the Finance department’s needs, writes a trace record to a file to record the information that has been extracted, and passes the message on to the Finance Department. See “Extracting elements from the message” on page 271 and “Writing a trace entry” on page 272.
   - The **Stock** branch of the flow (described in “Stock flow” on page 273) is used to add up all instances of an item sold. This information is passed on to the Distribution group so that they can maintain stock levels.
   - The **Partner** branch of the flow (described in “Partner Flow” on page 276) is used to filter messages that contain more than one purchase (multibuys) of the same item (“Filtering multibuy records” on page 276), store multibuy information in a database (“Updating the Multibuy database” on page 279), and publish messages containing multibuys to subscribers (“Publishing the message” on page 282).

2. The register subscriptions flow.

   This flow (described in “The Register subscriptions flow” on page 282) is used to subscribe to publications based on a topic or on a part of the content of the message.

3. The exception handling flow.

   This flow (described in “Including one message flow in another” on page 284) is an example of a message flow that has been created to be embedded within a higher-level flow. This illustrates the use of a repeated sequence of actions that can be included at several points within a larger message flow to provide common function.

How to create a message flow

To create a message flow like those described here, you must be in the Message Flows view. Select the Message Flows root object, right click and select **Create — Message Flow**. You are presented with a dialog in which you must enter the message flow name.

The new message flow, for example, the *Audit* flow, is listed under the Message Flows root in the left hand pane. The right hand pane is empty, awaiting the new message processing nodes. You can drag and drop the primitives in the left hand pane into the right hand pane to create the flow you want. The nodes are allocated a default name when they appear in the right hand pane: to rename them to the names illustrated here you must right click the node and select **Rename**.

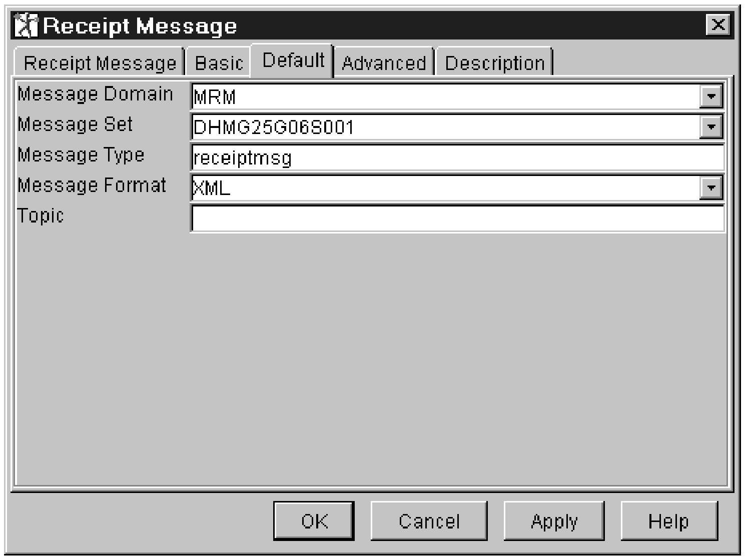
To complete the flows for the scenario, use the instructions for each to decide which nodes you need, how to connect them, and how to configure their properties.

For more details about how to create message flows and connect nodes, see “Creating a message flow” on page 89.

Getting the message

The first node in the message flow, Receipt Message, is an MQInput node. This node gets the message from an MQSeries queue on the queue manager hosting the broker (MQSI_SAMPLE_QM for MQSI_SAMPLE_BROKER). In our example, all of the properties, except the queue name, have been left to default.

If you have an MRM-defined message and you don’t specify the message domain, message set, message type, message format, and topic in the MQRFH2 header, you must specify these on the Default tab of the MQInput node, as shown in Figure 67.



Message Domain	MRM
Message Set	DHMG25G06S001
Message Type	receiptmsg
Message Format	XML
Topic	

Figure 67. MQInput node properties

Two terminals of the Receipt Message MQInput node are connected:

- Out connects to the next node in the flow. For a message repository defined message, this is Check Messages. For an XML message, this is Store Messages.
- Failure connects to an MQOutput node which puts messages to a failure queue.

Audit flow

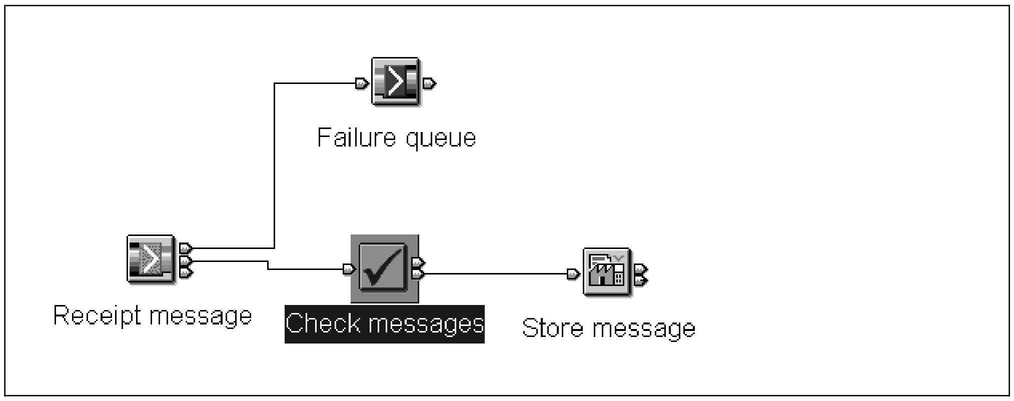


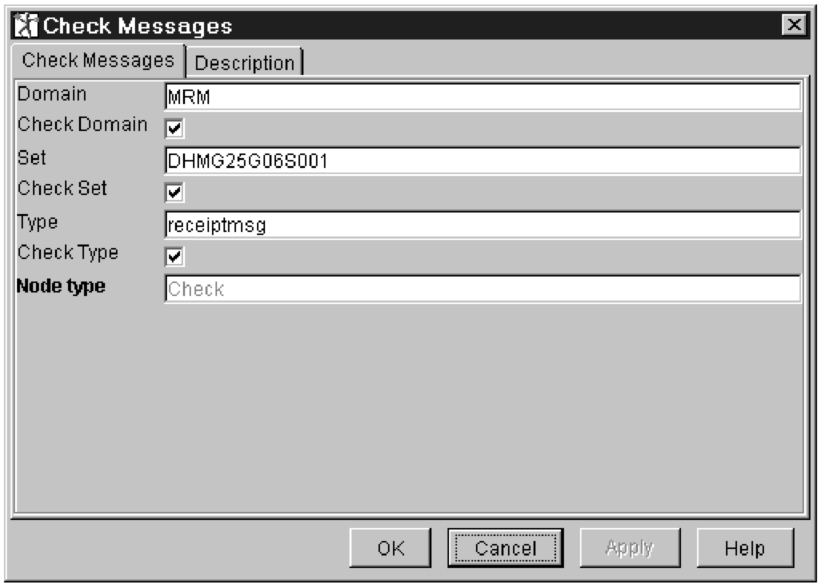
Figure 68. Audit message flow

The Audit flow for a message repository defined message contains two nodes: Check and Warehouse. The flow is used to check that the incoming message belongs to the expected message set and is therefore valid to pass through the rest of the flow, and to store the receipt information in a database for retrieval later.

The Audit flow for a self-defining XML message contains a Database node. No checking is necessary against a message set and the Warehouse node is for use only with a message repository defined message.

Checking the message (predefined message only)

This node applies only when you have an incoming message that has been predefined using the Control Center. If you are using an XML message, leave this node out. See “Check node” on page 106 for details of the Check node.



Domain	MRM
Check Domain	✓
Set	DHMG25G06S001
Check Set	✓
Type	receiptmsg
Check Type	✓
Node type	Check

Figure 69. Check node properties

For this example, configure the node properties as shown in Figure 69. Note that the message set number is the identifier given to the message set created in

"Defining the message in the message repository" on page 259. The message type is the identifier given to the message definition.

Storing the entire message

You need a message repository definition of the message to be able to use the Warehouse node.

If you are using an XML message, you would replace this node with a Database node, configured in a similar way to the Multibuy Database node, described in "Updating the Multibuy database" on page 279. You create a database schema for every element of the message and itemize every element in the ESQL used in the node to insert values into database columns. Part of the ESQL is shown below:

```
INSERT INTO Database.RECEIPTINFO2 (Storename, Branchnum, Cashiernum, Till
num, Date, Time, Itemname, Itemcode, Itemprice, Itemquantity, TotalItems,
Multibuy, Totalsales, Change)
VALUES
(Body.Message.receiptmsg.transactionlog.storedetailselement.storename,
Body.Message.receiptmsg.transactionlog.storedetailselement.branchnum,
Body.Message.receiptmsg.transactionlog.storedetailselement.cashiernum,
Body.Message.receiptmsg.transactionlog.storedetailselement.tillnum,

(and so on)
```

For more information about using these field references in ESQL, see "Referring to simple fields in a message" on page 299.

Configuring the Warehouse node: The Warehouse node stores the message as a binary object with a timestamp.

Before you can complete the Warehouse node, you must create the following:

- A database called MYDB (in this example)
- An ODBC connection to the MYDB database
- A table called receiptinfo in the MYDB database
- The columns spmsg and msgtime in the receiptinfo table

You must also ensure that the broker to which you will deploy this message flow has access to the database MYDB.

The following extract of ESQL illustrates how you can create the table and two columns in a DB2 database. From a DB2 command window, enter the following:

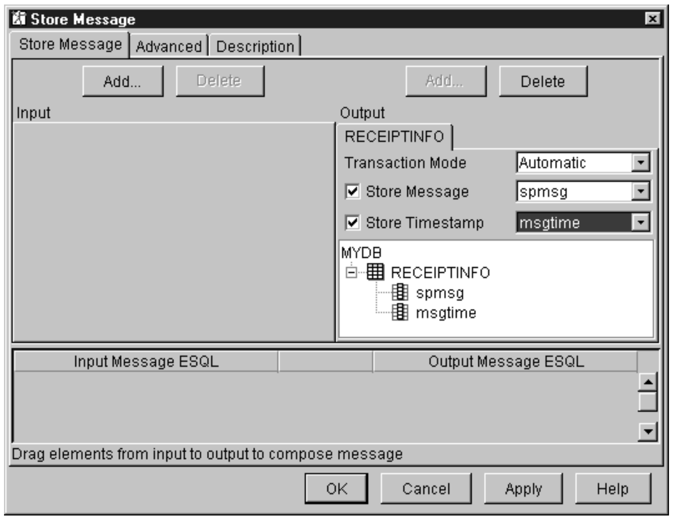
```
db2 connect to MYDB
db2 create table receiptinfo (spmsg BLOB(4M) not null, msgtime TIMESTAMP)
```

When you have set up the database in this way, you can set up the Warehouse node. In the node properties:

- Click the Output Add button and add the database MYDB and table RECEIPTINFO, then the columns SPMSG and TIMESTAMP.

  Note: You do not need to add the Input message, receiptmsg, because you are storing the entire message, and do not need to refer to elements within the message. If you want to warehouse a subset of the input message, you do need to add this on the Warehouse node properties dialog.
- Check the box to Store Message and select the column SPMSG.
- Check the box to Store Timestamp and select the column MSGTIME.

- Click OK.



Transaction Mode: Automatic
[✓] Store Message: spmsg
[✓] Store Timestamp: msgtime

MYDB
RECEIPTINFO
spmsg
msgtime

Drag elements from input to output to compose message

Figure 70. Warehouse node properties

You can check whether the message is stored in the Warehouse. For example, in a DB2 command window type

```
db2 connect to MYDB
db2 select * from receiptinfo
```

You won't see the text of the message because of the way it is stored (as a BLOB) but you will see the timestamp at the bottom.

The out terminal of the Warehouse node (Store Messages) is connected to the in terminal of three nodes:

- Extract node (Extract financial information) at the start of the Finance flow.
- Compute node (Add product instances) at the start of the Stock flow.
- Filter node (Multibuy filter) at the start of the Partners flow

Finance flow

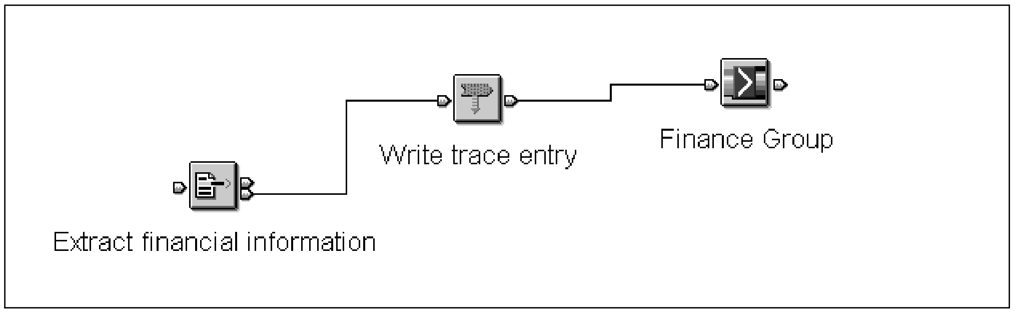


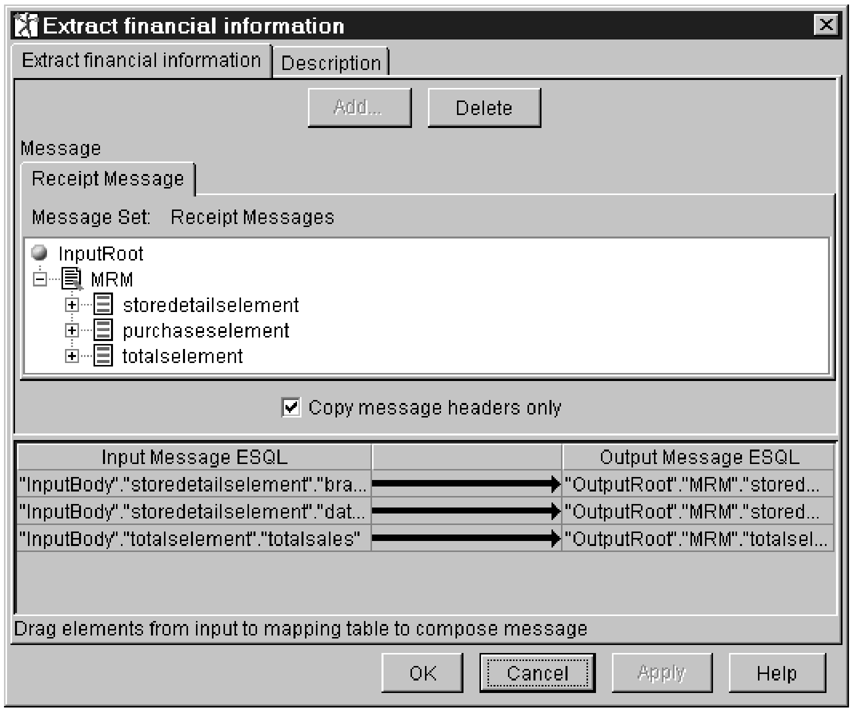
Figure 71. Finance message flow

The Finance flow contains three nodes: Extract (or Compute) Trace, and Output. The Finance department wants to receive only part of the information from the receipt message. The Extract Financial information node extracts the branch number, date, and total sales information. The Trace node writes a trace record to a file to record the information that has been extracted. The Output node passes the finance message to the Finance department.

Extracting elements from the message

The node you use here depends on your message definition:

- If you have a message repository definition of the message, use an Extract node.



Message Set: Receipt Messages

☑ Copy message headers only

Input Message ESQL		Output Message ESQL
"InputBody"."storedetailselement"."bra...	→	"OutputRoot"."MRM"."stored...
"InputBody"."storedetailselement"."dat...	→	"OutputRoot"."MRM"."stored...
"InputBody"."totalselement"."totalsales"	→	"OutputRoot"."MRM"."totalsel...

Drag elements from input to mapping table to compose message

Figure 72. Extract node properties

In the Extract node properties:

– Click Add and select the message set called Receipt Messages and the message called receiptmsg.

– Expand the elements storedetailselement and totalselement. Drag branchnum, date, and totalsales into the mapping window below.

– Click OK.

- If you have a self-defining XML message, use a Compute node. In the Compute node properties:
  1. Select **Copy message headers**.
  2. On the ESQL tab, use the following ESQL:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
SET OutputRoot.*[I] = InputRoot.*[I];
SET I = I + 1;
END WHILE;
SET OutputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.branchnum
  = InputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.branchnum;
SET OutputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.date
  = InputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.date;
SET OutputRoot.XML.Message.receiptmsg.transactionlog.totalselement.totalsales
  = InputRoot.XML.Message.receiptmsg.transactionlog.totalselement.totalsales;
```

  3. Click OK.

You can browse the extracted message using MQSeries Explorer. Select the message on the output queue, click Properties, then the Data tab.

The out terminal of the Extract or Compute nodes is connected to the Trace node.

Writing a trace entry

The trace node writes a trace entry according to the pattern that you define for the node properties. This can include any text that appears as comments (documentation), and values that are substituted from the message (for example, field contents). The values to be substituted must be surrounded by the characters `${(value)}`.

The following ESQL pattern is used to write a trace entry containing the three extracted elements and a simple timestamp to a file:

- For the message with the message repository definition:

```
Message passed through the Trace node with the following fields:

Branch number is: ${Body.storedetailselement.branchnum}
Date is: ${Body.storedetailselement.date}
Total sales are: ${Body.totalselement.totalsales}

Time is: ${EXTRACT(HOUR FROM CURRENT_TIMESTAMP)}:${EXTRACT(MINUTE FROM CURRENT_TIMESTAMP}
```

- For the self-defining XML message:

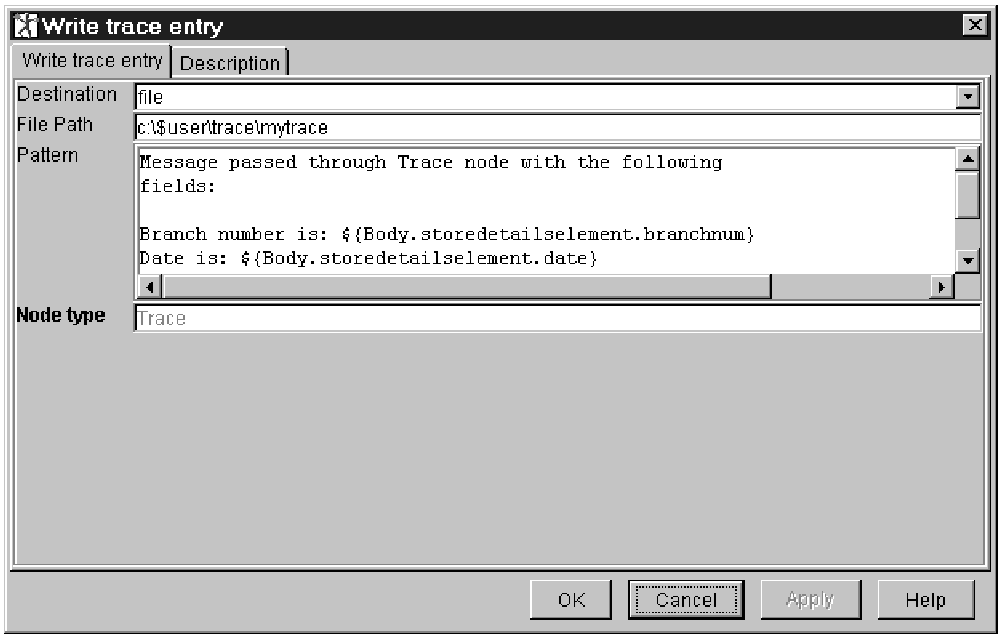
```
Message passed through the Trace node with the following fields:

Branch number is: ${Body.XML.Message.receiptmsg.transactionlog.storedetailselement.branchnum}
Date is: ${Body.XML.Message.receiptmsg.transactionlog.storedetailselement.date}
Total sales are: ${Body.XML.Message.receiptmsg.transactionlog.totalselement.totalsales}

Time is: ${EXTRACT(HOUR FROM CURRENT_TIMESTAMP)}:${EXTRACT(MINUTE FROM CURRENT_TIMESTAMP)}
```

The trace file in this example is called `mytrace` in location `c:\$user\trace`.

Figure 73 illustrates the node properties set up for the MRM message. The property Destination has been changed to file.



Destination: file
File Path: c:\$user\trace\mytrace
Pattern:

```
Message passed through Trace node with the following
fields:

Branch number is: ${Body.storedetailselement.branchnum}
Date is: ${Body.storedetailselement.date}
```

Node type: Trace

Figure 73. Trace node properties

The out terminal of the Trace node is connected to an MQOutput node, that names an MQSeries queue on which the message from the Finance flow will be put.

Stock flow

The stock flow is used to add up all instances of an item sold and this information is passed to the Distribution group so that they can maintain stock levels. For example, if a shopper buys two bottles of shampoo, the receipt will contain two instances of shampoo, as shown in the messages described in Figure 64 on page 258 and Figure 66 on page 265.

The stock flow contains a Compute node that adds up product instances as shown in Figure 74 on page 274 The example below shows how you can use ESQL to add the product instances and put the value into a new field (total item quantity) in the message being output from the node. For a message repository defined message, the example in “Using the stock flow with a predefined message” on page 275 illustrates how you can use the drag-and-drop capabilities of the node to map selected elements from an input message (Receipt Message) to a different output message (Stock Message) as well as calculating the total item quantity.

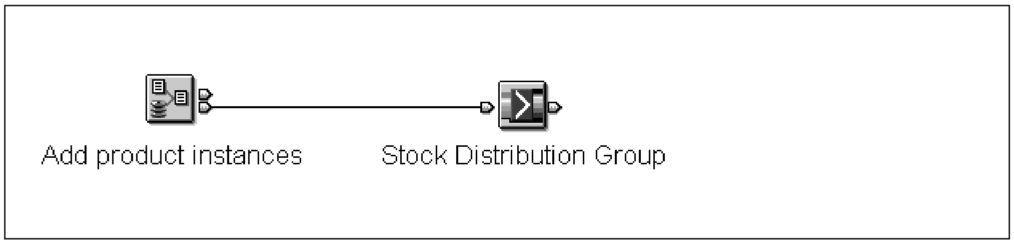


Figure 74. Stock message flow

Using the stock flow with an XML message

You can use the following ESQL to add up product instances in a Compute node for self-defining XML messages:

```
SET OutputRoot = InputRoot;
DECLARE TotalItemQuantity INTEGER;
SET TotalItemQuantity = (SELECT SUM(CAST(T.itemquantity AS INT))
FROM InputBody.Message.receiptmsg.transactionlog.purchaseselement.[] AS T
WHERE CAST(T.itemname AS CHAR) = 'Shampoo');
SET OutputRoot.XML.Message.receiptmsg.transactionlog.totalselement.totalitemquantity
  = TotalItemQuantity;
```

This declares a new element called `TotalItemQuantity` as an integer and sets its value to the sum of `ItemQuantity` where the `ItemName` is (in this example) Shampoo. The `TotalItemQuantity` element is placed within the `Totals` compound element in the output message.

Alternatively, you can use the following ESQL using a WHILE loop to output the same message:

```
SET OutputRoot = InputRoot;
DECLARE TotalItemQuantity INTEGER;
SET TotalItemQuantity = 0;
DECLARE current INTEGER;
DECLARE stop INTEGER;
SET current = 1;
SET stop = CARDINALITY(InputBody.Message.receiptmsg.transactionlog.*[]);

WHILE current <= stop DO
  IF CAST(InputBody.Message.receiptmsg.transactionlog.purchaseselement[current].
  itemname AS CHAR) = 'Shampoo' THEN
     SET TotalItemQuantity =  TotalItemQuantity +
     CAST(InputBody.Message.receiptmsg.transactionlog.purchaseselement[current]itemquantity AS INTEGER)
  END IF;
  SET current = current + 1;
END WHILE;
SET OutputRoot.XML.Message.receiptmsg.transactionlog.totalselement.totalitemquantity
  = TotalItemQuantity;
```

This loops through the receipt message increasing the value of TotalItemQuantity by one each time it comes across an instance of Shampoo, therefore adding up all instances of the Shampoo product in the receipt. Again, the `TotalItemQuantity` element is placed within the `Totals` compound element in the output message.

Using the stock flow with a predefined message

The example below extracts the store name, branch number, date, time, and purchases details from the incoming message (Receipt message) and puts them, and a value for total item quantity, into a different output message (Stock Message).

In the Compute node:

1. Click Add to add an input message. Select the message set Receipt Messages and the message Receipt Message.
2. Click Add to add an output message. Select the message set Receipt Messages and the message Stock Message.
3. Select **Use as message body**.
4. Select **Copy message headers**
5. Expand the storedetailselement and totalselement of the input message.

   Drag simple elements of storedetailselement (for example, storename, branchnum, and so on) from the input message onto their equivalent in the output message. You will see the mappings build up on the Mappings tab.
6. On the ESQL tab, edit the ESQL as shown below. (Much of the ESQL will have been generated for you already by the selections you made on the node properties and by the mappings.)

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
     SET OutputRoot.*[I] = InputRoot.*[I];
     SET I=I+1;
END WHILE;
SET "OutputRoot"."MRM"."storedetailselement"."storename" =
 "InputBody"."storedetailselement"."storename";
SET "OutputRoot"."MRM"."storedetailselement"."branchnum" =
 "InputBody"."storedetailselement"."branchnum";
SET "OutputRoot"."MRM"."storedetailselement"."date" =
 "InputBody"."storedetailselement"."date";
SET "OutputRoot"."MRM"."storedetailselement"."time" =
 "InputBody"."storedetailselement"."time";

DECLARE stop INTEGER;
DECLARE countitems INTEGER;
DECLARE current INTEGER;

SET stop = CARDINALITY("InputBody"."purchaseselement"[]);
SET current = 1;
SET countitems = 0;

WHILE current <= stop DO
SET "OutputRoot"."MRM"."purchaseselement"[current] =
  "InputBody"."purchaseselement"[current];
IF "InputBody"."purchaseselement"[current]."itemname" = 'Shampoo' THEN
SET countitems = countitems +
  "InputBody"."purchaseselement"[current]."itemquantity";
END IF;
SET current = current + 1;
END WHILE;
```

```
    SET "OutputRoot"."MRM"."outputtotalselement"."totalitemquantity" = countitems;
    SET OutputRoot.Properties.MessageSet = 'DHM695G070001';
    SET OutputRoot.Properties.MessageType = 'stockmsg';
```

The out terminal of the Compute node is connected to an MQOutput node, which names an MQSeries queue on which the message from the Stock flow will be put.

Partner Flow

The partner flow is used to track and keep details of products that are selling well. If more than one of the same product is bought on the same transaction, this is called a ‘multibuy’. Each multibuy record is placed into a database for easy access and reference by partners.

The message flow contains a Filter node to filter ‘multibuy’ records and a Database node to insert the records into the Multibuy database for partners.

| Database updates are completed in a loop within the flow: this allows the database
| to be updated for every item in the receipt message.

| The messages are also published through a Publication node. If the message is a
| self-defining XML message, a Compute node is used to add the required MQRFH2
| header information to the message before routing it to the Publication node. The
| MRM-defined message used in this scenario has the MQRFH2 header in it already.

| Partners subscribe to the publications, either by topic ("Multibuy"), or by filtering on
| the content of the message (the value of the "itemname" field).

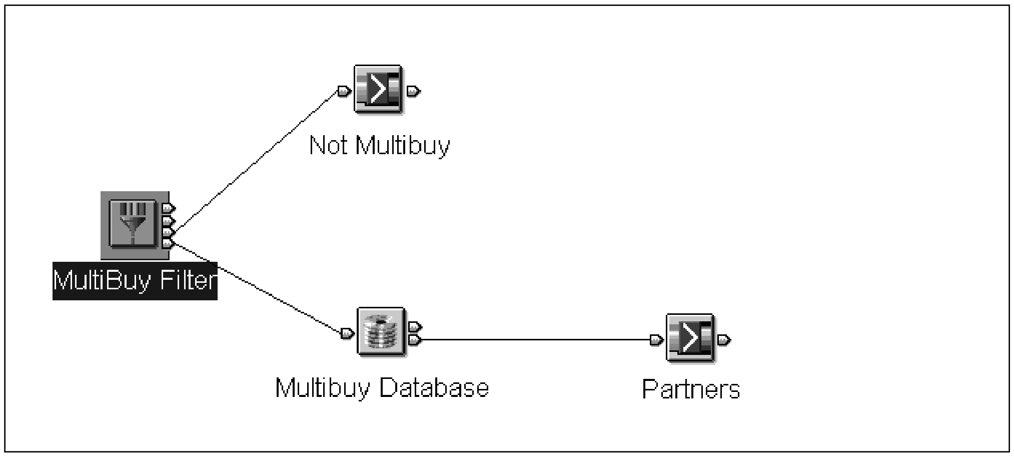


Figure 75. Partner message flow

Filtering multibuy records

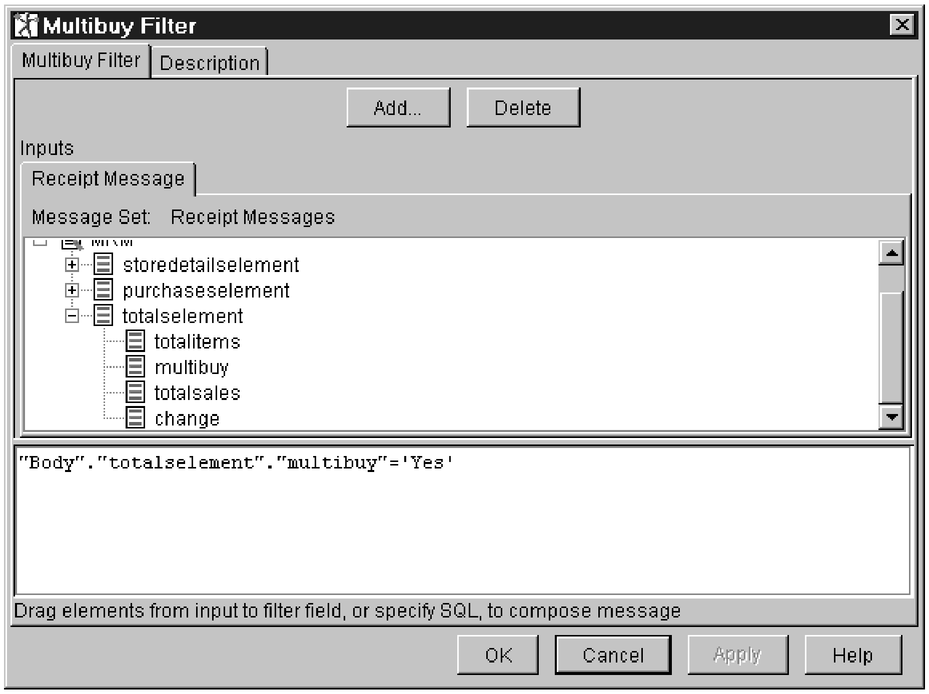
The Filter node is set up to filter all messages with the value `Yes` in the `Multibuy` field on to the Database or DataInsert node.

To configure the filter node for a self-defining XML message, use the following ESQL:

```
Body.Message.receiptmsg.transactionlg.totalselement.multibuy = 'Yes'
```

To configure the Filter node to use the MRM-defined message set:

1. Click Add and select the message set Receipt messages and the message receiptmsg.
2. Expand totalselement. Drag and drop the multibuy element into the filter field below.
3. Edit the ESQL by adding `='Yes'` to the expression that was generated by the drag-and-drop.



```
"Body"."totalselement"."multibuy"='Yes'
```

Figure 76. Filter node properties showing the set up for the MRM-defined message

Two terminals of the Filter node are connected:

- The true terminal of the Filter node is connected to the next node in the flow (the "Set counter" compute node).
- The false terminal of the Filter node is connected to an MQOutput node that names an MQSeries queue on which messages will be put when their “multibuy” value is “no”.

Using a loop to update the database

The loop in the message flow ensures that every item in the receipt message is recorded in the Multibuy database. The loop is made up of five nodes:

1. A compute node named "Set counter", that sets up a flag count.
2. A Filter node, named "Test value of counter". This tests if the value of the flag counter is greater than zero. If the expression evaluates to true, the message continues through the loop. If the expression evaluates to false, the message is propagated to the Compute node "Create Publication" to continue its route through the main message flow.
3. A Compute node named "Prepare for database": this prepares values for corresponding columns in the Multibuy database.
4. A Database node named "Multibuy database", that inserts values into the Multibuy database table.

5. A Compute node named "Reset counter" that resets the flag counter and connects back to the "Test value of counter" Filter node.

Initializing the loop control flag

The "Set counter" compute node sets the initial value for a counter to control the loop.

1. Access the node properties.
2. Select **Copy entire message**.
3. On the ESQL tab, enter the following ESQL below the comment line:

```
SET OutputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.f_reserve =
  CARDINALITY(InputRoot.XML.Message.receiptmsg.transactionlog.purchaseselement[]);
```

4. Click OK to confirm the property updates.
5. Connect the out terminal of the node to the in terminal of the "Test value of counter" node.

Figure 77 illustrates the nodes that make up the loop, and their connections with the main message flow.

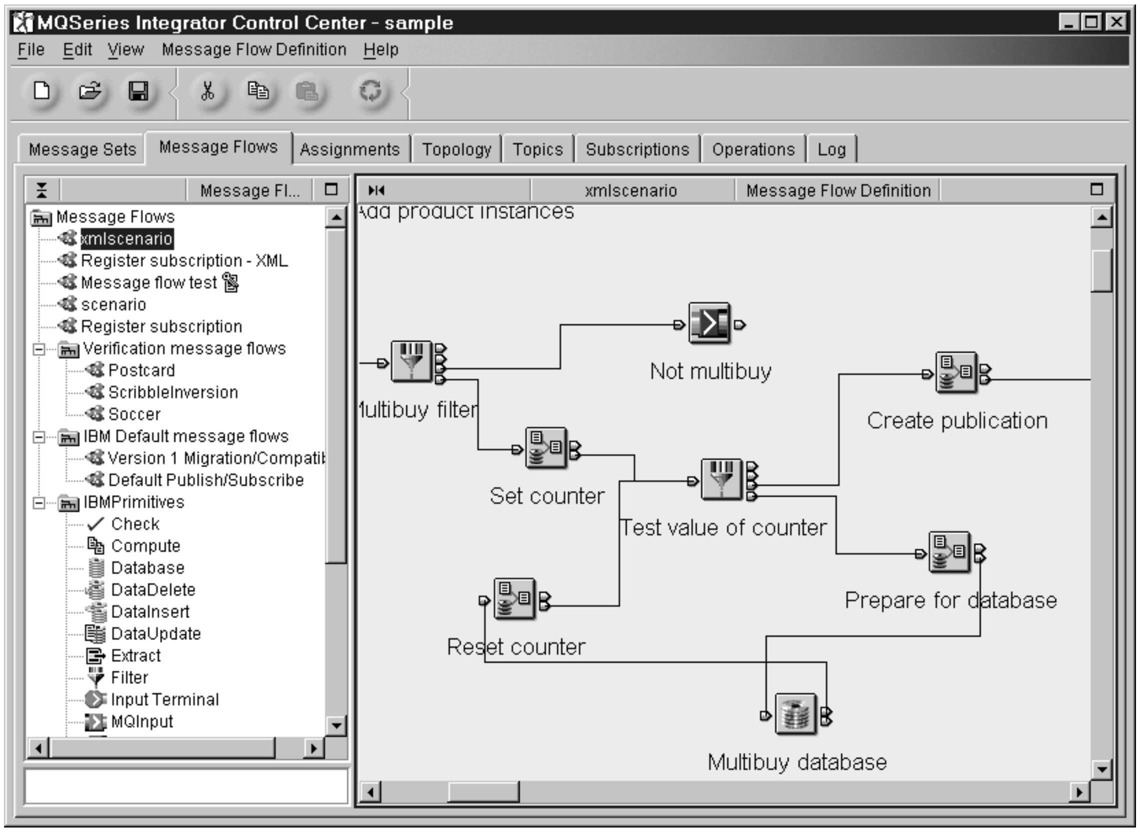


Figure 77. The loop to record data in the database

Testing the value of the loop control flag

The "Test value of counter" filter node ensures the loop is terminated when the total number of items have been recorded in the database.

To configure this node:

1. Enter the following ESQL:

   ```
   Body.Message.receiptmsg.transactionlog.storedetailselement.f_reserve > 0
   ```

2. Connect the true terminal (when the counter indicates the loop must be reiterated) to the in terminal of the "Prepare for database" compute node. This ensures the loop is executed one more time.
3. Connect the false terminal (when the counter indicates the loop must be terminated) to route the message to the remainder of the main message flow:
   a. For the XML message, connect to the in terminal of the "Create Publication" compute node.
   b. For the MRM message, connect to the in terminal of the "Publish" node.

Preparing the values for insertion in the database

You must change the data type of some fields in the message that you want to store in the database.

TO configure this node:

1. Click **Copy entire message**.
2. On the ESQL tab, use the following ESQL:

```
DECLARE elementnum INTEGER;
SET OutputRoot = InputRoot;
SET elementnum =
 CAST(InputRoot.XML.Message.receiptmsg.transactionlog.storedetailselement.f_reserve AS INTEGER);
SET OutputRoot.XML.Message.receiptmsg.transactionlog. storedetailselement.n_reserve
 = InputRoot.XML.Message.receiptmsg.transactionlog.purchaseselement[elementnum].itemname;
SET OutputRoot.XML.Message.receiptmsg.transactionlog. storedetailselement.q_reserve
 = InputRoot.XML.Message.receiptmsg.transactionlog.purchaseselement[elementnum].itemquantity;
```

3. Connect the out terminal of this node to the in terminal of the "Multibuy database" database node.

Updating the Multibuy database

You can use either a Database or a DataInsert node to update the Multibuy database.

If you are using an XML message, use the Database node. Before you can complete the Database node, you must create the following:

- A database called MYDB
- An ODBC connection to the MYDB database
- A table called MULTIBUY in the MYDB database
- The columns BRANCHNUM, ITEMNAME, and QUANTITY in the MULTIBUY table

You can achieve this using the following commands, entered in a DB2 command window:

```
db2 create database MYDB
db2 connect to MYDB
db2 create table MULTIBUY (branchnum integer not null, itemname char(40)
 not null, quantity integer not null)
```

You must also ensure that the broker to which you will deploy this message flow has access to this database.

When you have set up the database in this way, you can define the ESQL to access the information in the Database node.

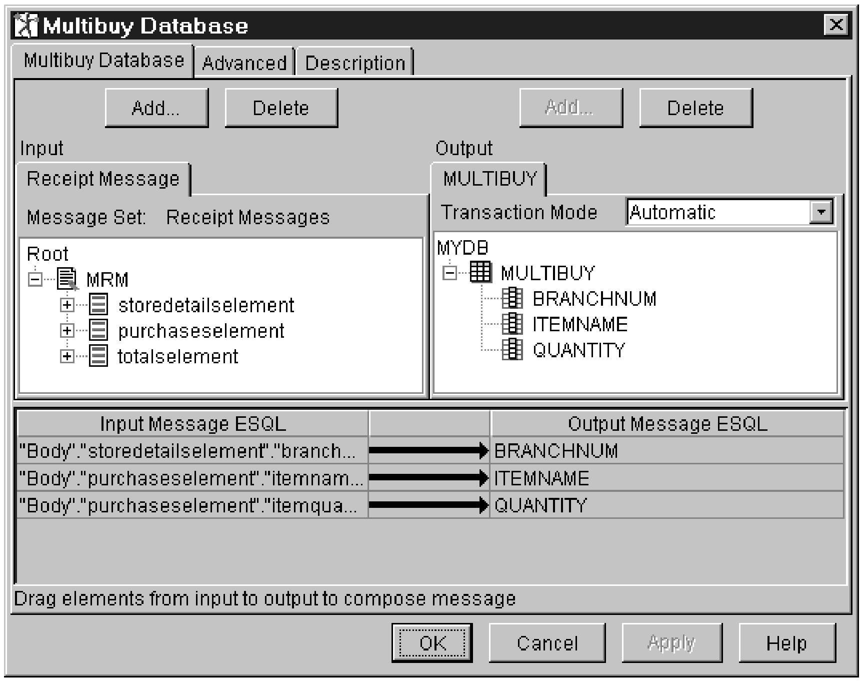
1. Select Add to add the Output Database Table. Enter the name of the database and the table.
2. Enter the following ESQL:

   ```
   INSERT INTO Database.MULTIBUY(branchnum,itemname,quantity)
   VALUES(Body.Message.receiptmsg.transactionlog.storedetailselement.branchnum,
   Body.Message.receiptmsg.transactionlog.storedetailselement.n_reserve,
   Body.Message.receiptmsg.transactionlog.storedetailselement.q_reserve)
   ```
3. Select the Advanced tab. Check the box for Treat warnings as errors. This action is not specifically required, but it is taken here both to illustrate the use of this option, and to allow easier testing of the error processing within the message flow.

If you have created an MRM-defined message set, you can use the DataInsert node for inserting information into a database in place of the Database node.

To configure the DataInsert node:

1. Click Add and select the message set Receipt messages and the message receiptmsg.
2. Click Add and add the database name, table, and columns.
3. Expand storedetailselement and purchaseselement. Drag and drop branchnum, itemname, and quantity onto the name of the target column.
4. Select the Advanced tab. Check the box for Treat warnings as errors.



Message Set: Receipt Messages

Transaction Mode: Automatic

Root
- MRM
  - storedetailselement
  - purchaseselement
  - totalselement

MYDB
- MULTIBUY
  - BRANCHNUM
  - ITEMNAME
  - QUANTITY

Input Message ESQL		Output Message ESQL
"Body"."storedetailselement"."branch...	→	BRANCHNUM
"Body"."purchaseselement"."itemnam...	→	ITEMNAME
"Body"."purchaseselement"."itemqua...	→	QUANTITY

Drag elements from input to output to compose message

Figure 78. Data Insert node properties

Connect the out terminal of this node to the in terminal of the "Reset counter" compute node.

Updating the value of the loop control counter

You must configure the "Reset counter" compute node to update the loop control counter to ensure the loop is executed the correct number of times.

To configure the node:

1. Click **Copy entire message**.
2. On the ESQL tab, use the following ESQL:

   ```
   SET OutputRoot = InputRoot;
   SET OutputRoot.XML.Message.receiptmsg.transactionlog.
     storedetailselement.f_reserve =
     CAST(InputRoot.XML.Message.receiptmsg.transactionlog.
          storedetailselement.f_reserve AS INTEGER)-1;
   ```

3. Connect the out terminal of this node to the in terminal of the "Test value of counter" filter node.

Creating the publication (XML message only)

In this scenario, the self-defining XML message does not have an MQRFH2 header. Therefore you must add a header to the message to define it as a publication and to set a topic of "Multibuy". A Compute node (named Create Publication) is used to do this. (The predefined message shown in Figure 66 on page 265 already has the required header information, so this action is not necessary if you are using the message defined to the MRM.)

To configure the Compute node:

1. Click **Copy message headers**.
2. On the ESQL tab, use the following ESQL:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I = I+1;
END WHILE;
SET OutputRoot.MQRFH2.psc.Command = 'Publish';
SET OutputRoot.MQRFH2.psc.Topic = 'Multibuy';
SET OutputRoot.XML = InputBody;
```

Publishing the message

A Publication node is added to the flow. All properties are left to default on this node.

The Register subscriptions flow

A separate message flow, the Register subscriptions flow illustrated in Figure 79, is used to register a subscription.

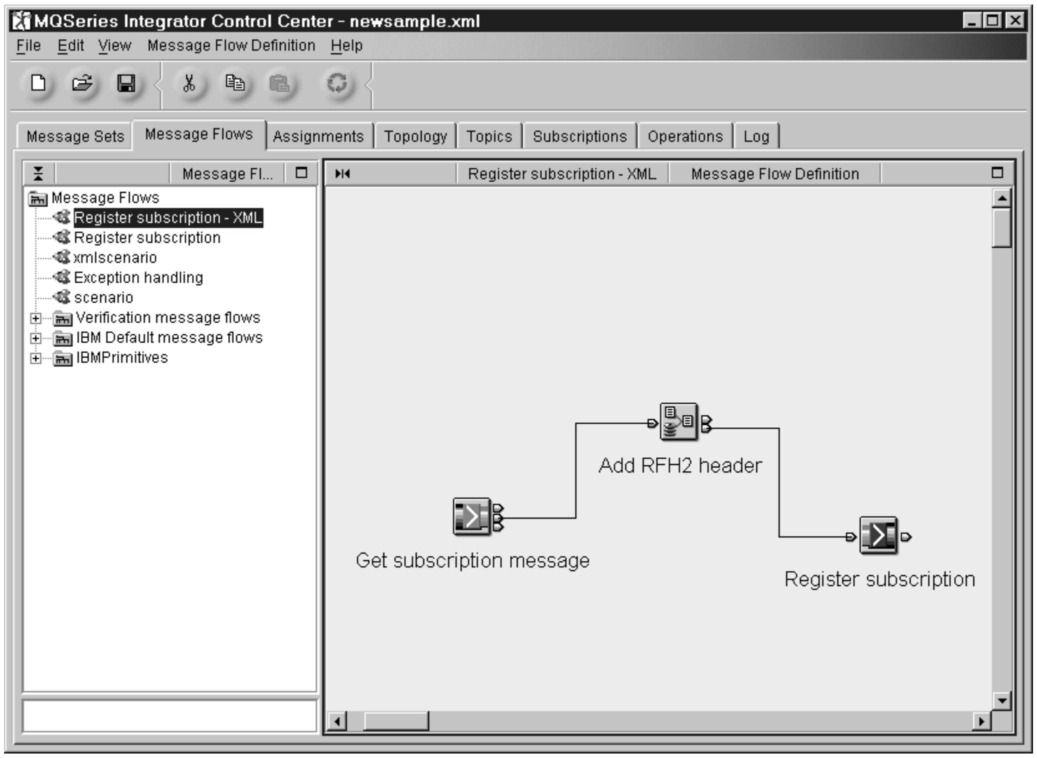


Figure 79. The Register subscriptions flow

The actions performed by this flow are:

1. You put an XML message to the queue you define as the input queue (SUBIN in this example).
2. The MQInput node retrieves the message.

3. The subscription registration information and any additional filter criteria are set in the MQRFH2 header.
4. The subscription message is put, through the MQOutput node, to the broker's queue SYSTEM.BROKER.CONTROL.QUEUE. If you have set the subscription message type to 1 (MQMT_REQUEST), the broker sends back a response. If you want to see this response (which should be "Completion OK"), you must update the MQMD of the XML subscription message to include a ReplyToQ and ReplyToQmgr. The broker sends the response to the subscription message to this queue.
5. Any published messages that match the criteria set in the MQRFH2 header of the subscription message are put on to the queue also named in the MQRFH2 header (in the QName and QMName fields).

This additional flow must be assigned to the broker in the same way as the other scenario flows. See "Assigning message flows to the execution group" on page 286 for further details.

The example shows filtering on the value of a field in the message (content-based filtering). If you want to subscribe only by topic, remove the filter line. Be aware that if you set up a subscription on a topic of "Multibuy", you will have to delete that subscription if you then want to do a content-based filter from the same subscriber application ID. The second, content-based, subscription does not overwrite the more general topic-based one. You can delete subscriptions from the Control Center Subscriptions view.

Configuring the Register subscriptions flow

You must configure the nodes within the subscription flow as follows:

1. Specify the name of the queue from which the messages are read by the MQInput queue. The queue is called SUBIN.
2. The Compute node is used to add the MQRFH2 header information. To configure the Compute node:
   - Click on **Copy message headers**.
   - On the ESQL tab, use the following ESQL:

     ```
     .
     DECLARE I INTEGER;
     SET I = 1;
     WHILE I < CARDINALITY(InputRoot.*[]) DO
       SET OutputRoot.*[I] = InputRoot.*[I];
       SET I = I+1;
     END WHILE;
     SET OutputRoot.MQRFH2.psc.Command = 'RegSub';
     SET OutputRoot.MQRFH2.psc.Topic = 'Multibuy';
     ```

     If you are subscribing to a self-defining XML message, set the filter like this (note that all quotes used in this example are single quotes):

     ```
     .
     SET OutputRoot.MQRFH2.psc.Filter
       = 'Message.receiptmsg.transactionlog.purchaseselement.itemname = ''Shampoo''';
     ```

     If you are subscribing to an MRM-defined message, set the filter like this (note that all quotes used in this example are single quotes):

```
SET OutputRoot.MQRFH2.psc.Filter = 'purchaseselement.itemname =''Shampoo''';

SET OutputRoot.MQRFH2.psc.QMName = 'MQSI_SAMPLE_QM';
SET OutputRoot.MQRFH2.psc.QName = 'SUBS';
SET OutputRoot.XML = InputBody;
```

QName and QMName identify the queue on which you want to receive any publication messages that match the subscription.

You might have noticed that the form of the filter in this header and in the Filter node look different. The expressions in the Filter node were:

- Body.Message.receiptmsg.transactionlog.purchaseselement.multibuy (for the XML message).
- "Body"."totalselement"."multibuy" (for the MRM-defined message).

This is because when the message is tested by the broker to see if it matches a subscription request, it has already been parsed and processed. It is no longer in the parse tree, so there is no concept of Body.xxx, as there is in the Filter node. In the subscription, you filter on the message data only.

3. The MQOutput node has SYSTEM.BROKER.CONTROL.QUEUE as the Queue Name. The Queue Manager Name is the name of the queue manager hosting the broker.

Including one message flow in another

The company in the scenario has connected the failure terminal of the MQInput node (called Receipt Message). Any failure downstream causes the message to be rolled back to the MQInput node and propagated through the failure terminal to an MQSeries queue called FAILED. You must define this queue to receive these messages.

If you connect the failure terminal of any node, this action makes you responsible for handling exceptions generated by the message flow. Therefore MQSeries Integrator does not report any messages to the local error log to show which of the nodes downstream caused the failure.

When the company extends its message flow for this retail scenario, potentially to several hundred nodes, it must be able to trap exception information where it occurs, and take appropriate action.

The company decides to create a simple message flow to handle these error situations, and to include this message flow within the larger message flows. The error subflow has one Input Terminal connected to a Compute node that is in turn connected to two Output Terminals.

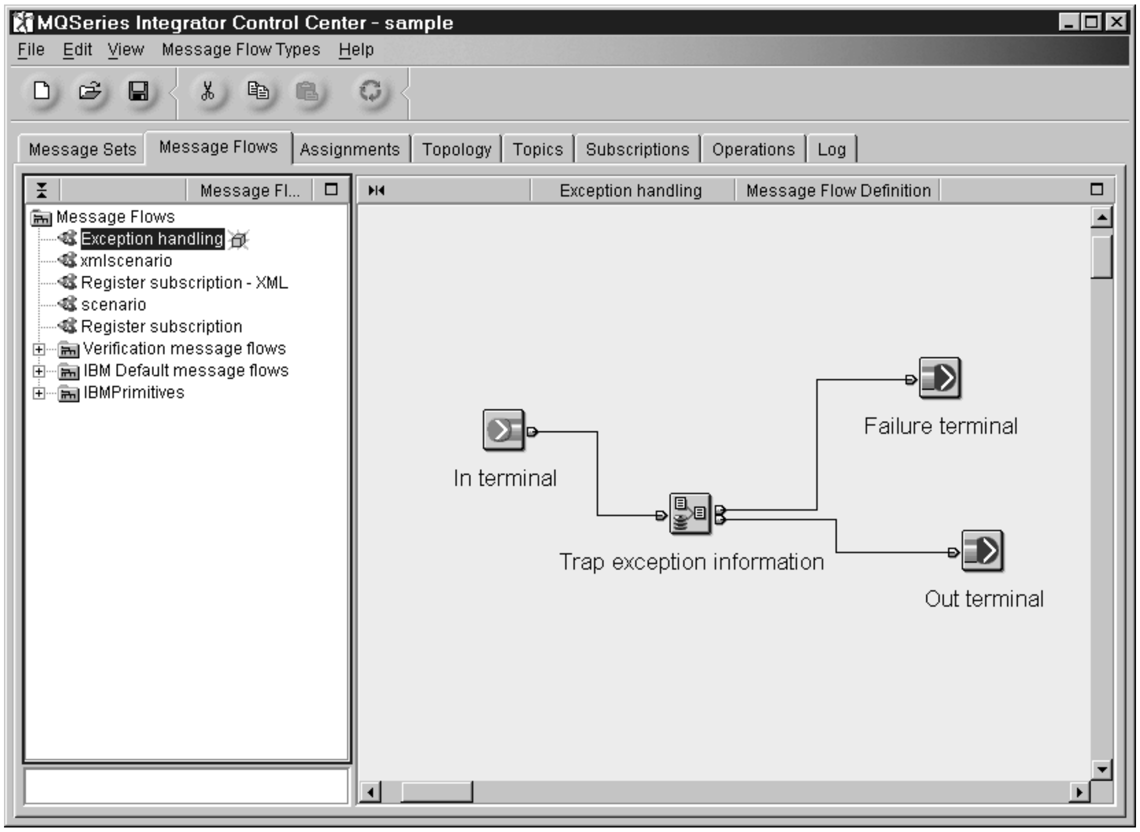


Figure 80. The error subflow

This simple flow is attached to the failure terminal of the Database (Multibuy Database) node. If an XML message is passed through the Database node with a value that does not match the ESQL statements in the node (for example, <receiptmsg1> instead of <receiptmsg>), a warning message is generated. This is handled as an error because of the Treat warnings as errors property of the Database node, and it causes a failure.

You can reuse this simple error subflow from any node and in any higher-level message flow, with just a simple customization of the text field within the Compute node ESQL.

The Compute node in the simple flow is used to create an XML message that contains:

- User-written text to identify the point of failure (the node to which the Exceptions flow was attached) and, if known, probable causes.
- The message that caused the failure
- The exception list that is generated by MQSeries Integrator

The Output terminal of the simple message flow is connected to an MQOutput node, which interacts with the MQSeries queue EXCEPTION.

To set up the embedded flow:

1. Create the simple message flow shown in Figure 80. Drag and connect the Input Terminal, the Compute node, and the Output Terminals.

2. Configure the Compute node:
   a. Click **Copy message headers**.
   b. On the ESQL tab, add the following lines to the generated ESQL:
      ```
      SET OutputRoot.XML.ERROR.TEXT = 'Failure in Multibuy database node
        - check XML attributes and SQL references match';
      SET OutputRoot.XML.ERROR.(XML.tag)MSG = InputBody;
      SET OutputRoot.XML.ERROR.EXCEPTLST = InputExceptionList;
      ```
3. Drag the simple message flow into the scenario flow. Connect the failure terminal of the Multibuy Database (database) node to the Input Terminal of the Exceptions flow. Connect the Output Terminal of the Exceptions flow to an MQOutput node naming the queue EXCEPTION as the destination for the exception message. You must define this queue to receive these messages.

   Figure 63 on page 257 shows how the loop is used within the complete flow.

Assigning message flows to the execution group

You must assign message flows to an execution group in a broker before they are able to process messages.

In the Assignments view, check out the execution group. Drag and drop the main flow and the Register subscription flow onto the name of the execution group in the graphic in the Domain Topology pane. The exception flow is assigned automatically, as part of the main flow. Check in the execution group.

Deploying the configuration

Finally, to use the configuration set up, you must deploy it. To do this, select **File —> Deploy —> Complete configuration (all types) —> Normal**. When the configuration has been successfully deployed, you can put receipt messages and subscription messages to the appropriate queues and the messages will be processed through the message flows. You can check the success of the deploy by refreshing and viewing the messages displayed in the Log view.

Appendix B. C and COBOL default mappings

This appendix describes the defaults that the C and COBOL importers use when mapping C datatypes or COBOL datatypes to MRM datatypes. The data designer defining a message set in the Control Center might want to follow these defaults, but this decision will depend on the business usage of the data.

The MRM:

- Does not support pointer datatypes.
- Does not suppport the COBOL construct REDEFINES.
- Does not support the COBOL datatypes DBCS, external floating point, or binary items that have a PIC declaration greater than 9 digits.
- Does not fully support the C datatype long double.

Mapping C datatypes to MRM datatypes

Table 34 on page 288 defines the datatype mappings for C structures.

| **Notes:**

| 1. Long Double is outside the scope of the importer.

Table 34. C datatypes and their default settings in the MRM

C datatype	MRM logical type	Physical type	Length	Sign	String justification	Repeat
Long	Integer	Integer	4	Signed		
Char	String	Fixed Length	1			
Char[10]	String	Fixed Length	10		Left justify	
Char[10][3]	String	Fixed Length	3		Left justify	10
Char[10][3][6]	String	Fixed Length	6		Left justify	30
Int	Integer	Integer	4	Signed		
Int[2]	Integer	Integer	4	Signed		2
Int[2][3]	Integer	Integer	4	Signed		6
Unsigned Int	Integer	Integer	4	Unsigned		
Unsigned Short	Integer	Integer	2	Unsigned		
Float	Float	Float	4			
Double	Float	Float	8			
Short	Integer	Integer	2	Signed		
Unsigned char	Integer	Integer	1			
Unsigned char[2]	Binary	Binary	2			
`(#define)BOOL int`	Boolean	Boolean				
`(#define)Boolean`	Boolean	Boolean				

Mapping COBOL datatypes to MRM datatypes

Columns 1 to 5 in Table 35 on page 290 describe some examples of COBOL data definitions. Columns 6 to 12 describe the equivalent data mappings used to store these definitions in the MRM.

Notes:

1. Column 5 (of 12) Internal representation assumes an ASCII Big Endian code page.
2. Column 12 (of 12) **Jst.** indicates the justification of the datatype.
3. The following datatypes are outside the scope of the importer:
   - Binary (10 to 18 digits)
   - External floating point
   - DBCS

Table 35 (Page 1 of 3). COBOL datatypes and their default settings in the MRM

COBOL datatype	Permitted symbols	PICTURE and USAGE and optional SIGN clause	Value	Internal representation	MRM Logical type	Physical type	Length in bytes	Sign	Virtual dec. point	Pad. char.	Jst.
External decimal (Zoned Decimal)	`9 P S V`				If > 9 digits: float If a fraction: float Else: integer	extended decimal	=num of digits. If sign is separate, add 1				
		`PIC S9999 DISPLAY`	+1234	31 32 33 34	Integer	extended decimal	4	Y including trailing	0		
			-1234	31 32 33 74							
			1234	31 32 33 34							
		`PIC 9999 DISPLAY`	1234	31 32 33 34	integer	extended decimal	4	N	0		
		`PIC 99V99 DISPLAY`	1234	31 32 33 34	float	extended decimal	4	N	2		
		`PIC S9999 DISPLAY SIGN LEADING`	+1234	31 32 33 34	integer	extended decimal	4	Y including leading	0		
			-1234	71 32 33 34							
		`PIC S9999 DISPLAY SIGN LEADING SEPARATE`	+1234	2B 31 32 33 34	integer	extended decimal	5	Y separate leading	0		
			-1234	2D 31 32 33 34							
		`PIC S9999 DISPLAY SIGN TRAILING SEPARATE`	+1234	31 32 33 34 2B	integer	extended decimal	5	Y separate trailing	0		
			-1234	31 32 33 34 2D							

Table 35 (Page 2 of 3). COBOL datatypes and their default settings in the MRM

COBOL datatype	Permitted symbols	PICTURE and USAGE and optional SIGN clause	Value	Internal representation	MRM Logical type	Physical type	Length in bytes	Sign	Virtual dec. point	Pad. char.	Jst.
Binary	`9 P S V`				integer	integer	if <5 decimal digits, 2 bytes If 5 thru 9 digits, 4 bytes				
		`PIC S9999 BINARY` or `COMP` or `COMP-4` or `COMP-5`	+1234	04 D2	integer	integer	2	Y			
			-1234	FB 2E							
		`PIC 9999 BINARY` or `COMP` or `COMP-4` or `COMP-5`	1234	04 D2	integer	integer	2	N			
Internal Decimal (Packed Decimal)	`9 P S V`				If > 9 digits, float If a fraction, float Else integer		Rounded down result of (Num of digits+2)/2				
		`PIC S9999 PACKED-DECIMAL` or `COMP-3`	+1234	01 23 4C	integer	packed decimal	3	Y			
			-1234	01 23 4D							
		`PIC 9999 PACKED-DECIMAL` or `COMP-3`	1234	01 23 4F	integer	packed decimal	3	N			
Internal floating point											
	no PIC clause	`COMP-1`	+1234	44 9A 40 00	float	float	4	Y			
			-1234	C4 9A 40 00							
	no PIC clause	`COMP-2`	+1234	40 93 48 00 00 00 00 00	float	float	8	Y			

Table 35 (Page 3 of 3). COBOL datatypes and their default settings in the MRM

COBOL datatype	Permitted symbols	PICTURE and USAGE and optional SIGN clause	Value	Internal representation	MRM Logical type	Physical type	Length in bytes	Sign	Virtual dec. point	Pad. char.	Jst.
			-1234	C0 93 48 00 00 00 00 00							
Alphabetic	A	`PIC A(3) DISPLAY`	ABC	41 42 43	string	fixed length	3 chars			space	L
Alphanumeric	X	`PIC XXXX DISPLAY`	DEF	44 45 46 20	string	fixed length	4 chars			space	default L
		`PIC X(4) JUSTIFIED RIGHT`	DEF	20 44 45 46	string	fixed length	4 chars			space	R
		`JUST RIGHT`									
Alphanumeric edited	`X B 0 9 /`	`PIC BX/9 DISPLAY`	A/3	20 41 2F 33	string	fixed length	4 chars			space	L
Numeric edited	`B P V Z 9 0/` `comma symbol` `period symbol` `+ - CR DB * $`						length in chars=sum of chars in PIC string excluding V			space	default R
		`PIC 9999`	0123	30 31 32 33	string	fixed length	4			space	R
		`PIC ZZZ9`	123	20 31 32 33							
		`PIC $$Z9`	$123	24 31 32 33							
			$12	20 24 31 32							
		`PIC 999V9`	1234	31 32 33 34							
		`PIC ZZZV9`	1234	31 32 33 34							
		`PIC $ZZZ,ZZ9V.99`	$123,456.78	24 31 32 33 2C 34 35 36 2E 37 38	string	fixed length	11			space	R

Appendix C. ESQL reference

This appendix describes how to use the ESQL expressions that are necessary for configuring Filter, Compute, and Database nodes in MQSeries Integrator. A common syntax, based on standard SQL and known an ESQL (Extended SQL) is used in these nodes. Exceptions to standard SQL constructs are included here: the standard constructs are not documented in detail.

This appendix does not provide information about content-based filters. Refer to Appendix A in the *MQSeries Integrator Programming Guide* which discusses and explains this topic.

Basic message structure

The example below is the text of a simple XML message. Many of the examples in this appendix are based on this message or messages of this form.

```
<Trade type='buy'
 Company='IBM'
 Price='200.20'
 Date='2000-01-01'
 Quantity='1000'/>
```

If this message is received on an MQSeries queue, the message originated in the message flow from an MQInput node, and had an MQRFH2 header. The tree representation of this message would look like this (where indentation shows containment):

```
Root
      Properties
         CreationTime=GMTTIMESTAMP '1999-11-24 13:10:00'
          (a GMT timestamp field)

  ... and other fields ...

      MQMD
         PutDate=DATE '19991124'
          (a date field)

         PutTime=GMTTIME '131000'
          (a GMTTIME field)

  ... and other fields ...

      MQRFH
          mcd
          msd='xml'
             (a character string field)

         .. and other fields ...

      XML
         Trade
          type='buy'
          (a character string field)

         Company='IBM'
          (a character string field)

         Price='200'
          (a character string field)

         Date='2000-01-01'
          (a character string field)

         Quantity='1000'
          (a character string field)
```

A good way to see the structure of the syntax element tree of a message is to use a simple message flow that contains a Trace node configured using a trace pattern such as "${Root}". The Trace node produces a trace entry that contains a tree similar to the one shown above. For an explanation of the different datatypes that fields can have, such as character string and GMT timestamp, see “Data types” on page 300.

For more information on basic message concepts, see Chapter 3, “Defining messages” on page 29.

Understanding the message structure

When a message is received from an MQSeries queue, the broker parses it into a tree format. This is illustrated in “Basic message concepts” on page 29.

The first child of Root is the Properties folder. The next sibling of Properties is the folder for the MQMD of the incoming message. The next sibling is the folder for the MQRFH2 header. The last child of Root is the Body that represents the message content.

If the tree structure is not clear from just from looking at your message, you can generate a trace record to show it. Create, assign, and deploy a simple message flow containing an MQInput node to get the message from an MQSeries queue, and a Trace node with a Destination property of file, a File Path property set to a fully-qualified path name of the file where the trace record will be written, and a Pattern property of ${Root}.

The following figure shows the structure of the predefined receipt message used in the example scenario (see Figure 66 on page 265).

```
(
  (0x1000000)Properties = (
    (0x3000000)MessageSet     = 'DHOOL7806S001'
    (0x3000000)MessageType    = 'receiptmsg'
    (0x3000000)MessageFormat  = 'XML'
    (0x3000000)Encoding       = 546
    (0x3000000)CodedCharSetId = 437
    (0x3000000)Transactional  = TRUE
    (0x3000000)Persistence    = FALSE
    (0x3000000)CreationTime   = GMTTIMESTAMP '2000-06-15 14:02:46.770'
    (0x3000000)ExpirationTime = -1
    (0x3000000)Priority       = 0
    (0x3000000)Topic          = 'Multibuy'
  )
  (0x1000000)MQMD       = (
    (0x3000000)SourceQueue      = 'IN'
    (0x3000000)Transactional    = TRUE
    (0x3000000)Encoding         = 546
    (0x3000000)CodedCharSetId   = 437
    (0x3000000)Format           = 'MQHRF2  '
    (0x3000000)Version          = 2
    (0x3000000)Report           = 0
    (0x3000000)MsgType          = 8
    (0x3000000)Expiry           = -1
    (0x3000000)Feedback         = 0
    (0x3000000)Priority         = 0
    (0x3000000)Persistence      = 0
    (0x3000000)MsgId            = X'414d51204d5153495f53414d504c455f8e12463913700300'
    (0x3000000)CorrelId         = X'000000000000000000000000000000000000000000000000'
    (0x3000000)BackoutCount     = 0
    (0x3000000)ReplyToQ         = '                                                '
    (0x3000000)ReplyToQMgr      = 'MQSI_SAMPLE_QM                                  '
    (0x3000000)UserIdentifier   = 'gb055075    '
    (0x3000000)AccountingToken  = X'16010515000000021355b0f025f447e8c3b3515e80300000000000000000000b'
    (0x3000000)ApplIdentityData = '                                '
    (0x3000000)PutApplType      = 11
    (0x3000000)PutApplName      = 'C:\Argo\scenario\argoput.exe'
    (0x3000000)PutDate          = DATE '2000-06-15'
    (0x3000000)PutTime          = GMTTIME '14:02:46.770'
    (0x3000000)ApplOriginData   = '    '
    (0x3000000)GroupId          = X'000000000000000000000000000000000000000000000000'
    (0x3000000)MsgSeqNumber     = 1
    (0x3000000)Offset           = 0
    (0x3000000)MsgFlags         = 0
    (0x3000000)OriginalLength   = 1096
  )
  (0x1000000)MQRFH2     = (
    (0x3000000)Version        = 2
    (0x3000000)Format         = '        '
    (0x3000000)Encoding       = 546
    (0x3000000)CodedCharSetId = 437
    (0x3000000)Flags          = 0
    (0x3000000)NameValueCCSID = 437
    (0x1000000)mcd            = (
      (0x1000000)Msd  = (
        (0x2000000) = 'MRM'
      )
      (0x1000000)Set  = (
        (0x2000000) = 'DHOOL7806S001'
      )
      (0x1000000)Type = (
        (0x2000000) = 'receiptmsg'
      )
      (0x1000000)Fmt  = (
        (0x2000000) = 'XML'
      )
    )

    (0x1000000)psc            = (
      (0x1000000)Command = (
        (0x2000000) = 'Publish'
      )
      (0x1000000)Topic   = (
        (0x2000000) = 'Multibuy'
      )
    )
  )
  (0x1000008)MRM        = (
    (0x1000001)storedetailselement = (
      (0x3000001)storename  = 'ShopAholics'
      (0x3000001)branchnum  = 9
      (0x3000001)cashiernum = 5
      (0x3000001)tillnum    = 9
      (0x3000001)date       = '01/04/00'
      (0x3000001)time       = '14:30'
    )
    (0x1000001)purchaseselement    = (
      (0x3000001)itemname     = 'Shampoo'
      (0x3000001)itemcode     = 56734097
      (0x3000001)itemprice    = 2.99E+0
      (0x3000001)itemquantity = 1
    )
    (0x1000004)purchaseselement    = (
      (0x3000001)itemname     = 'Shampoo'
      (0x3000001)itemcode     = 56734097
      (0x3000001)itemprice    = 2.99E+0
      (0x3000001)itemquantity = 1
    )
    (0x1000004)purchaseselement    = (
      (0x3000001)itemname     = 'Shampoo'
      (0x3000001)itemcode     = 5663548
      (0x3000001)itemprice    = 1.99E+0
      (0x3000001)itemquantity = 1
    )
    (0x1000001)totalselement       = (
      (0x3000001)totalitems = 10
      (0x3000001)multibuy   = 'Yes'
      (0x3000001)totalsales = 1.348E+3
      (0x3000001)change     = 5.02E+0
    )
  )
)
```

The following figure shows the equivalent self-defining XML message, that does not have an MQRFH2 header.

```
(
  (0x1000000)Properties = (
    (0x3000000)MessageSet     = ''
    (0x3000000)MessageType    = ''
    (0x3000000)MessageFormat  = ''
    (0x3000000)Encoding       = 546
    (0x3000000)CodedCharSetId = 437
    (0x3000000)Transactional  = TRUE
    (0x3000000)Persistence    = FALSE
    (0x3000000)CreationTime   =
GMTTIMESTAMP '2000-06-15 12:47:25.210'
    (0x3000000)ExpirationTime = -1
    (0x3000000)Priority       = 0
    (0x3000000)Topic          = NULL
  )
  (0x1000000)MQMD       = (
    (0x3000000)SourceQueue      = 'IN'
    (0x3000000)Transactional    = TRUE
    (0x3000000)Encoding         = 546
    (0x3000000)CodedCharSetId   = 437
    (0x3000000)Format           = 'XML     '
    (0x3000000)Version          = 2
    (0x3000000)Report           = 0
    (0x3000000)MsgType          = 8
    (0x3000000)Expiry           = -1
    (0x3000000)Feedback         = 0
    (0x3000000)Priority         = 0
    (0x3000000)Persistence      = 0
    (0x3000000)MsgId            = X'414d51204d5153495f53414d504c455f8e12463913100300'
    (0x3000000)CorrelId         = X'000000000000000000000000000000000000000000000000'
    (0x3000000)BackoutCount     = 0
    (0x3000000)ReplyToQ         = '                                                '
    (0x3000000)ReplyToQMgr      = 'MQSI_SAMPLE_QM                                  '
    (0x3000000)UserIdentifier   = 'gb055075    '
    (0x3000000)AccountingToken  = X'16010515000000021355b0f025f447e8c3b3515e80300000000000000000000000b'
    (0x3000000)ApplIdentityData = '                                '
    (0x3000000)PutApplType      = 11
    (0x3000000)PutApplName      = 'C:\Argo\scenario\argoput.exe'
    (0x3000000)PutDate          = DATE '2000-06-15'
    (0x3000000)PutTime          = GMTTIME '12:47:25.210'
    (0x3000000)ApplOriginData   = '    '
    (0x3000000)GroupId          = X'000000000000000000000000000000000000000000000000'
    (0x3000000)MsgSeqNumber     = 1
    (0x3000000)Offset           = 0
    (0x3000000)MsgFlags         = 0
    (0x3000000)OriginalLength   = 858
  )
  (0x1000010)XML        = (
    (0x1000000)Message = (
      (0x1000000)receiptmsg = (
        (0x1000000)transactionlog = (
          (0x1000000)storedetailselement = (
            (0x1000000)storename  = (
              (0x2000000) = 'ShopAholics'
            )
            (0x1000000)branchnum  = (
              (0x2000000) = '9'
            )
            (0x1000000)cashiernum = (
              (0x2000000) = '05'
            )
            (0x1000000)tillnum    = (
              (0x2000000) = '09'
            )
            (0x1000000)date       = (
              (0x2000000) = '01/04/00'
            )
            (0x1000000)time       = (
              (0x2000000) = '14:30'
            )
          )
          (0x1000000)purchaseselement    = (
            (0x1000000)itemname     = (
              (0x2000000) = 'Shampoo'
            )
            (0x1000000)itemcode     = (
              (0x2000000) = '00056734097'
            )
            (0x1000000)itemprice    = (
              (0x2000000) = '2.99'
            )
            (0x1000000)itemquantity = (
              (0x2000000) = '1'
            )
          )
          (0x1000000)purchaseselement    = (
            (0x1000000)itemname     = (
              (0x2000000) = 'Shampoo'
            )
            (0x1000000)itemcode     = (
              (0x2000000) = '00056734097'
            )
            (0x1000000)itemprice    = (
              (0x2000000) = '2.99'
            )
            (0x1000000)itemquantity = (
              (0x2000000) = '1'
            )
          )

          (0x1000000)purchaseselement    = (
            (0x1000000)itemname     = (
              (0x2000000) = 'Shampoo'
            )
            (0x1000000)itemcode     = (
              (0x2000000) = '0005663548'
            )
            (0x1000000)itemprice    = (
              (0x2000000) = '1.99'
            )
            (0x1000000)itemquantity = (
              (0x2000000) = '1'
            )
          )
          (0x1000000)totalselement       = (
            (0x1000000)totalitems = (
              (0x2000000) = '10'
            )
            (0x1000000)multibuy   = (
              (0x2000000) = 'Yes'
            )
            (0x1000000)totalsales = (
              (0x2000000) = '1348'
            )
            (0x1000000)change     = (
              (0x2000000) = '5.02'
            )
          )
        )
      )
    )
  )
)
```

Referring to simple fields in a message

You refer to fields in a message using a *field reference* (further described in “Field references” on page 316). A field reference has a very similar format and meaning to a path in a file system. In its simplest form, a field reference consists of a period-separated sequence of identifiers. These identify the path in the message tree that get to the field that you want. The simplest form of identifier is a sequence of alphanumeric characters, the first of which must be an alphabetic character. Not all paths have to start at the root of the tree, so the first identifier in the chain indicates the starting point for the navigation.

The starting point of the field reference (the first identifier in the chain) is called the correlation name. For example, the correlation name "Root" means "start at the root of the tree", and "Body" means "start at the body of the message".

The following message root names are defined:

Root name	**Used in node**
Root	Filter
InputRoot	Compute
OutputRoot	Compute

The following pseudo message roots are also recognized. These pseudo roots do not refer to the root of a particular message, but to the root element of one of the addressable message parsers immediately beneath the root element.

Pseudo root name	**Equivalent path**	**Used in node**
Body	Root.*[LAST]	Filter Database
InputBody	InputRoot.*[LAST]	Compute
OutputBody	OutputRoot.*[LAST]	Compute
InputProperties	InputRoot.Properties	Compute
OutputProperties	OutputRoot.Properties	Compute

The correlation name differs depending on where you are making the reference to the field.

- If you are making the reference from a node:
  1. If you can alter the message in a node (for example, in a Compute node), you must use:
     - InputRoot (the root of the message coming into the node)
     - OutputRoot (the root of the message going out of the node)
     - InputBody (= InputRoot.Body) (the body of the input message)
  2. If you cannot alter the message in a node (for example, in a Filter node), use:
     - Root (the root of the message coming in and going out of the node)
     - Body (= Root.Body)

     Body refers to the last element of the root tree and depends on the type of message. If you have a self-defining XML message, Body equals XML so both of the following statements are valid:

     ```
     Body.Trade.Quantity.
     Root.XML.Trade.Quantity.
     ```

The first form is the recommended form.

You can also further refer to specific elements that share a type and name by using the bracketed index value after the name. You can use an integer value or the defined constant `LAST` (shown above). The index numbering starts at value 1, and numbering is assigned from left to right. For example, you can refer to the second occurrence of the `Initial` field within a parent with children `Salutation`, `Initial`, `Initial`, `Surname`, by specifying `Initial[2]`.

Using quotes in the field reference

If you need to refer to fields with periods or spaces in their names, you must use double quotes around the reference:

```
Body."Companies on Wall Street"."mycompany.com"
```

If you need to refer to fields that contain double quotes, you must use two sets of double quotes around the reference:

```
Body.""hello""
```

If you need to refer to fields that have the same name as an ESQL keyword, you must use double quotes around the reference:

```
Body."Set"
```

For a full list of keywords, see “Reserved keywords” on page 360.

Data types

Now that you have been introduced to simple filter expressions, you need to understand some more about the types of value that you can work with.

Note: It is important to realize when working with generic XML messages, such as the one shown in the previous example, that all of the values derived from a generic XML message are character strings. In some situations, the character string values are implicitly cast to other types, but in other situations it is necessary explicitly to cast the value into one of the correct type. For a more detailed explanation of the implications of these actions, see “CASTs” on page 338.

When working with message formats that are managed through the Control Center, either XML or record-oriented messages, you define the datatype associated with each field. For example, in this case you would have defined the `Quantity` field to be an integer type, and the field would be represented in the tree as an integer, rather than as a string.

Numbers

The standard SQL datatypes `INTEGER`, `FLOAT` and `DECIMAL` are supported, but you should take note of the following:

Integers

The integer datatype stores numbers using 64-bit binary precision, so giving a range of values between -9223372036854775808 and 9223372036854775807. In addition to the normal integer literal format, integer literals can be written in hexadecimal notation, for example `0x1234abc`.

The hexadecimal letters A to F can be written in uppercase or lowercase, as can the `'x'` after the initial zero.

Note: If a literal of this form is too large to be represented as an integer, it is represented as a decimal.

Floats

A value of the float datatype is a 64 bit approximation of a real number. A float literal is defined using the scientific notation, as in `6.6260755e-34`.

The case of the "e" is not significant so "E" can be used instead if necessary. It is the "e" that identifies this value as a float literal.

Strings

Strings can be character strings, byte strings, or bit strings.

- A string of any type must be enclosed in single quotes (as shown in the examples below, and throughout this appendix).
- If you want to include a single quote within a character string literal, you must use another single quote as an escape character.

  For example, the assignment SET X='he''was''' puts the value `he'was'` into X.

Character strings

Character strings in MQSeries Integrator are stored using Unicode.

Byte strings

A byte string is a series of 8-bit bytes that is used to represent arbitrary binary data. A byte string literal is defined using a string of hexadecimal digits, as in the following example:

```
X'0123456789ABCDEF'
```

There must be an even number of digits in the string, because two digits are required to define each byte. Each digit can be one of the hexadecimal digits. The hexadecimal letters can be specified in uppercase or lowercase.

Bit strings

A bit string is a series of bits used to represent arbitrary binary data that does not contain an exact number of bytes. Bit string literals are defined in a similar way to byte string literals, for example:

```
B'0100101001'
```

Any number of digits, which must be either 0 or 1, can be specified.

Datetime types

The `DATE`, `TIME`, `TIMESTAMP`, `GMTTIME` and `GMTTIMESTAMP` datatypes are collectively known as *datetime datatypes*. The following are examples of literals for these datatypes:

```
DATE '1999-11-18'
DATE '2000-02-29'
TIME '12:02:00'
TIME '06:00:00'
TIME '11:49:23.656'
TIMESTAMP '1999-12-31 23:59:59'
```

The format of the character string following the `DATE` keyword is `'yyyy-mm-dd'`. The character string includes a 4-digit year field, followed by a 2-digit month field, in which `'01'` represents January. (Note that a leading zero is required as the field must always be two digits.) The month field is followed by a 2-digit day field, which must also always be 2 digits, so a leading zero might be required. Each of the hour, minute and second fields in a `TIME` or `TIMESTAMP` literal must always be two digits. The exception is the optional fractional seconds field which, if present, can be up to 6 digits in length.

GMTTime
The GMTTime datatype is very similar to the Time datatype, except that the time values are interpreted as values in Greenwich Mean Time. GMTTime values are defined in much the same way as Time values, that is, as `GMTTIME '12:00:00'`.

GMTTimestamp
As with the GMTTime datatype, the GMTTimestamp datatype is very similar to the Timestamp datatype, except that the values are interpreted as values in Greenwich Mean Time. GMTTimestamp values are defined in much the same way as Timestamp values, that is as `GMTTIMESTAMP '1999-12-31 23:59:59.999999'`.

Interval

An interval value represents an interval of time. There are two kinds of interval values:

- One that is specified in years and months.
- One that is specified in days, hours, minutes and seconds (including fractions of a second).

The split between months and days arises because the number of days in each month varies. An interval of one month and a day is not really meaningful, and certainly cannot be sensibly converted into an equivalent interval in numbers of days only.

An interval value has a qualifier associated with it that indicates which fields are present. If it contains both a year and a month value, the month value must be within the range `[0, 11]`. However, in the case of an interval containing just a month value, that value is unconstrained. So, for example, an interval value of 18 months is valid, but an interval value of 2 years and 18 months is not valid.

A day interval contains a sign and a contiguous sequence of fields from the list DAY, HOUR, MINUTE, and SECOND. The qualifier indicates which fields are present. As with year-and-month intervals, the value of the first field is unconstrained, but the values of the subsequent fields are constrained as follows:

Field	Valid values
HOUR	0-23
MINUTE	0-59
SECOND	0-59.999...

Some examples of valid interval values are:

- 72 hours
- 3 days and 23 hours
- 3600 seconds
- 90 minutes and 5 seconds

Some examples of invalid interval values are:

- 3 days and 36 hours

  A day field is specified, so the hours field is constrained to `[0,23]`.
- 1 hour and 90 minutes

  An hour field is specified, so minutes are constrained to `[0,59]`.

An interval literal is defined by the following syntax:

```
INTERVAL <interval string> <interval qualifier>
```

The format of interval string and interval qualifier are defined by the following table:

Table 36. Format of interval strings and qualifiers

Interval qualifier	**Interval string format**	**Example**
YEAR	'<year>' or '<sign> <year>'	'10'
YEAR TO MONTH	'<year>-<month>' or '<sign> <year>-<month>'	'- 2-06'
MONTH	'<month>' or '<sign> <month>'	'18'
DAY	'<day>' or '<sign> <day>'	'-30'
DAY TO HOUR	'<day> <hour>' or <sign> <day> <hour>'	'1 02'
DAY TO MINUTE	'<day> <hour>:<minute>' or '<sign> <day> <hour>:<minute>'	'1 02:30'
DAY TO SECOND	'<day> <hour>:<minute>:<second>' or '<sign> <day> <hour>:<minute>:<second>'	'1 02:30:15' or '-1 02:30:15.333'
HOUR	'<hour>' or '<sign> <hour>'	'24'
HOUR TO MINUTE	'<hour>:<minute>' or '<sign> <hour>:<minute>'	'1:30'
HOUR TO SECOND	'<hour>:<minute>:<second>' or '<sign> <hour>:<minute>:<second>'	'1:29:59' or '1:29:59.333'
MINUTE	'<minute>' or '<sign> <minute>'	'90'
MINUTE TO SECOND	'<minute>:<second>' or '<sign> <minute>:<second>'	'89:59'
SECOND	'<second>' or '<sign> <second>'	'15' or '15.7'

Here are some simple examples of interval literals:

```
INTERVAL '1' HOUR
INTERVAL '90' MINUTE
INTERVAL '1-06' YEAR TO MONTH
```

Boolean

A boolean represents a true or false value although there are exceptions. See "Optional fields and NULLs" on page 310 for more information.

A valid filter expression must always return a boolean value. A literal boolean can be defined using one of the keywords TRUE, FALSE, or UNKNOWN.

Predicates

The expression used to configure a Filter node must produce a boolean result. That means that in general it will consist of one kind of predicate. Many of the standard predicates are supported, and are listed in this section. Predicates can be combined using the AND, OR and NOT operators. In the following description, only the differences from standard SQL are described.

BETWEEN predicate

The standard default asymmetric form of the BETWEEN predicate is supported. This requires you to specify the smallest end-point value first, followed by the largest. You can use the ASYMMETRIC keyword, but in its absence the asymmetric form is implied.

If you prefer you can make the BETWEEN predicate symmetric by specifying the optional keyword SYMMETRIC after BETWEEN. In the symmetric form of the predicate, the order in which you specify the two end-point values is not significant. For example, the following two expressions are identical:

```
2 BETWEEN SYMMETRIC 1 AND 3
2 BETWEEN SYMMETRIC 3 AND 1
```

Both expressions return the value "TRUE".

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The standard LIKE predicate for performing simple string-pattern matching is supported.

The pattern is specified by a string in which the percent (%) and underscore (_) characters can be used to have special meaning:

- The underscore character _ represents any single character.

  For example, the following predicate finds matches for 'IBM' and for 'IGI', but not for 'International Business Machines' or 'IBM Corp':

  ```
  Body.Trade.Company LIKE 'I__'
  ```

- The percent character % represents a string of zero or more characters.

  For example, the following predicate finds matches for 'IBM', 'IGI', 'International Business Machines', and 'IBM Corp':

  ```
  Body.Trade.Company LIKE 'I%'
  ```

If you want to use the percent and underscore characters within the expressions that are to be matched, you must precede these with an ESCAPE character, which defaults to the backslash (\) character.

For example, the following predicate finds a match for 'IBM_Corp'.

```
Body.Trade.Company LIKE ‘IBM\_Corp’
```

You can specify a different escape character by using the ESCAPE clause on the LIKE predicate. For example, you could also specify the previous example like this:

```
Body.Trade.Company LIKE ‘IBM$_Corp’ ESCAPE ‘$’
```

IN predicate

An IN predicate of the following form is supported:

```
expression IN (expressiona, expressionb, ..., expressionk)
```

The IN predicate:

- Evaluates to TRUE if the comparison between the first expression and one of the expressions inside the parentheses evaluates to TRUE.
- Evaluates to FALSE if the comparison between the left-hand expression and all of the expressions inside the parentheses evaluate to FALSE.
- Evaluates to UNKNOWN if at least one comparison evaluates to UNKNOWN, and none evaluate to TRUE.

Other sorts of expression

The examples of predicates so far have all used simple literals or field references as the operands. However, you can use more general expressions.

CASE expressions

Both the simple and searched forms of the ESQL CASE expression are supported. You can only use CASE as an expression, not as a statement.

```
►►──CASE──────────────────────────────────────────────────────────────►

                                       ┌─ELSE NULL──────────────┐
►──┬─┤ searched-when-clause ├─┬───────┼────────────────────────┼──END──►◄
   └─┤ simple-when-clause ├───┘       └─ELSE result-expression─┘

searched-when-clause:

   ┌────────────────────────────────────────────────────┐
├──▼─WHEN search-condition──THEN──┬─result-expression─┬─┴──────────────┤
                                  └─NULL──────────────┘

simple-when-clause:
├──expression─────────────────────────────────────────────────────────►

   ┌───────────────────────────────────────────────┐
►──▼─WHEN ──expression──THEN──┬─result-expression─┬─┴───────────────┤
                              └─NULL──────────────┘
```

If you use the simple form, the value of the expression prior to the first WHEN keyword is tested for equality with the value of the expression following the WHEN keyword. The datatype of the expression prior to the first WHEN keyword must therefore be comparable to the datatype of each expression following a WHEN keyword.

The following examples show CASE expressions used as part of a Filter expression:

```
Body.TestCase.Result = CASE SUBSTRING(Body.TestCase.Val1 FROM 1 FOR 1)
WHEN 'A' THEN 'Administration'
WHEN 'B' THEN 'Human Resources'
WHEN 'C' THEN 'Accounting'
WHEN 'D' THEN 'Design'
WHEN 'E' THEN 'Operations'
ELSE 'Manufacturing'
END

Body.TestCase.Result = CASE
WHEN CAST(Body.TestCase.Val1 AS INT) < 15 THEN 'SECONDARY'
WHEN CAST(Body.TestCase.Val1 AS INT) < 19 THEN 'COLLEGE'
END
```

The following two examples show CASE expressions as part of a Filter expression where the CASE is being used within a SELECT against an external database.

```
Body.TestCase.Val1 =
   THE (SELECT ITEM CASE SUBSTRING(B.broker_firstname FROM 1 FOR 1)
        WHEN 'D' THEN 'Dave' ELSE 'noname' END
       FROM Database.broker_details AS B
       WHERE B.broker_id = CAST(Body.TestCase.Val2 AS INT))

CAST(Body.TestCase.Val1 AS INT) =
  THE (SELECT ITEM C.cust_id FROM Database.customer_details AS C WHERE
       C.cust_id = CAST(Body.TestCase.Val2 AS INT) AND
       C.cust_status = CASE WHEN
         CAST(Body.TestCase.Val3 AS INT) = 1 THEN 'A'
         ELSE 'I' END)
```

EVAL expressions

```
►►──EVAL──( expression )──────────────────────────────────►◄
```

You can use EVAL in two ways:

1. You can use it in place of a complete ESQL statement.
2. You can use it in place of an expression that forms part of an ESQL statement.

EVAL cannot be used in place of a non-expressional part of a statement.

EVAL takes one parameter in the form of an expression, and it evaluates this expression and casts the resulting value to a character string if it is not one already. The expression that is passed to EVAL must therefore be able to be represented as a character string.

After this first stage evaluation is complete, the behavior of EVAL depends on whether it is being used in place of a complete ESQL statement, or in place of an expression that forms part of an ESQL statement:

1. If it is a complete ESQL statement, the character string derived from the first stage evaluation is executed as if it were an ESQL statement.

2. If it is an expression that forms part of an ESQL statement, the character string is evaluated as if it were an expression and EVAL returns the result.

In the following examples A and B are integer scalar variables, and scalarVar1, operatorAsString are character string scalar variables.

The following statements are valid uses of EVAL:

-
  ```
  SET OutputRoot.XML.Data.Result = EVAL(A+B);
  ```

  The expression A+B is acceptable because, although it returns an integer value, integer values are representable as character strings, and the necessary cast is therefore performed before EVAL continues with its second stage of evaluation.

-
  ```
  SET OutputRoot.XML.Data.Result
   = EVAL('A' || operatorAsString || 'B');
  ```

-
  ```
  EVAL('SET ' || scalarVar1 || ' = 2;');
  ```

  The semicolon included at the end of the final string literal is necessary because if EVAL is being used in place of an ESQL statement, then its first stage evaluation must return a string that represents a valid ESQL statement, including the terminating semicolon.

The following are not valid uses of EVAL:

-
  ```
  SET EVAL(scalarVar1) = 2;
  ```

  In this example, EVAL is being used to replace a field reference, not an expression.

-
  ```
  SET OutputRoot.XML.Data.Result[]
   = EVAL((SELECT T.x FROM Database.y));
  ```

  In this example, the (SELECT T.x FROM Database.y) passed to EVAL returns a list, which is not representable as a character string.

The following example is acceptable because '(SELECT T.x FROM Database.y)' is a character string literal, not an expression in itself, and therefore is of course representable as a character string.

```
SET OutputRoot.XML.Data.Result[]
 = EVAL('(SELECT T.x FROM Database.y)');
```

The real power of EVAL is that it allows you to dynamically construct ESQL statements or expressions. In the second and third valid examples shown, for example, the value of scalarVar1 or operatorAsString can be set according to the value of an incoming message field, or other dynamic value, thus allowing you to effectively control what ESQL is executed without requiring a potentially lengthy IF...THEN ladder.

However, you must consider the performance implications in using EVAL - dynamic construction and execution of statements or expressions is necessarily more time-consuming than simply executing pre-constructed ones. If performance is vital, you might find it preferable to write more specific, but faster, ESQL.

Comments

Comments can be included in an expression. There are two forms of comment:

- Line comments are initiated by two consecutive minus signs, and the comment is terminated by an end-of-line character.
- Block comments are initiated by "/*" and are terminated by "*/". Block comments can be nested.

In arithmetic expressions you must take care not to initiate a line comment accidentally. For example, consider the expression:

```
1 - -2
```

Removing all white space from the expression results in:

```
1--2
```

which is interpreted as the number 1, followed by a line comment.

Using MQSeries constants in message headers

You can reference and update the fields within the MQSeries headers that are associated with each message. Every message has at least an MQMD, and most messages have one or more additional headers, for example the MQIIH (IMS/ESA® bridge header).

You can use the defined MQSeries constants to test and assign values to the message header fields, both in their symbolic form, and as defined values. However, in a small number of cases, the datatype of the constant has been changed to be consistent with the MQSeries Integrator implementation of the field to which it refers. Check Appendix D, “MQSeries message header parsers” on page 381 for individual field properties.

All MQSeries constants are registered in the global namespace (unqualified) and you can use them in any context without providing a qualifier. Constants are also registered in a qualified (parser-specific) namespace, and in this case must be qualified by the characters `namespace.`. For example, each parser resgisters a contstant of type CHARACTER called `ParserClassName`. The parser class for the MQMD must therefore be referenced by the string `MQMD.ParserClassName`.

For example, to set the message type field, you can specify:

```
SET OutputRoot.MQMD.MsgType = MQMT_DATAGRAM
```

You can achieve the same effect using the numeric value of MQMT_DATAGRAM:

```
SET OutputRoot.MQMD.MsgType = 8
```

When you use these constants in the ESQL in a node, the syntax checker accepts their use. However, if you use the *Check message flow* facility, the use of MQSeries constants might generate errors at this stage, with a message to indicate that the constants are not defined. You can ignore these messages: the constants will be processed correctly at run-time.

Table 37 on page 309 tells you where you can find the definitions of the MQSeries constants for the MQSeries headers supported by MQSeries Integrator.

Table 37. MQSeries constants references

Header	Description	Reference
MQPCF	PCF header comprising: • MQCFH Command format header • MQCFIN PCF integer parameter • MQCFST PCF string parameter • MQCFIL PCF integer list parameter • MQCFSL PCF string list parameter	*MQSeries Programmable System Management*
MQCIH	CICS® bridge	*MQSeries Application Programming Reference*
MQDLH	Dead letter	*MQSeries Application Programming Reference*
MQIIH	IMS bridge	*MQSeries Application Programming Reference*
MQMD	Message descriptor	*MQSeries Application Programming Reference*
MQMDE	MQMD extension	*MQSeries Application Programming Reference*
MQRFH	Rules and formats	*MQSeries Publish/Subscribe User's Guide*
MQRFH2	Rules and formats version 2	*MQSeries Integrator Programming Guide*
MQRMH	Reference message	*MQSeries Application Programming Reference*
MQSAPH SMQ_BMH	SAP R/3 Link headers	*MQSeries Link for R/3 User's Guide*
MQWIH	Workload information	*MQSeries Application Programming Reference*

CodedCharSetId, Encoding, and data conversion

The broker supports the MQSeries manifest constants for CodedCharSetId and Encoding fields, and follows the MQSeries architecture rules for supporting MQCCSI_INHERIT and MQCCSI_DEFAULT as necessary when generating a bitstream.

Optional fields and NULLs

If you want to process XML messages based on a field that is not always present in a message, you can use a DTD and define default values for attributes. With other messages, you can write a field reference in an expression that refers to any field, regardless of whether such a field exists in the message, or could ever exist in the message. If, when that field reference is evaluated, no matching field is found, a NULL value is returned. A NULL value indicates the absence of a value, and should not be confused with, for example, the empty string. You can always write a filter expression such as:

```
Body.Every.Field.Is.Valid > 123
```

However, it is probable that the field reference will result in a NULL value.

NULLs and expressions

The effect on a predicate or expression if one of the values is NULL is that the whole expression evaluates to NULL. The expression in the previous example attempts to compare the NULL value to the integer 123, which results in the NULL value. This behavior is based on interpreting NULL as "Could be anything". If the value of one field in an expression could be anything, the result of the expression could be anything.

Note that the UNKNOWN boolean value is interpreted as a NULL so, for example, comparing TRUE with UNKNOWN results in UNKNOWN, in the same way that comparing an integer with NULL results in unknown.

The logical operations AND and OR treat null values differently. The effect of NULL expressions on the values P and Q in AND | OR | NOT operations (as in standard SQL usage) are shown below.

Table 38. Logical operators and NULL values

P Q	P AND Q	P OR Q	
T T	T	T	
F T	F	T	
T F	F	T	
F F	F	F	
U T	U	T	Note the AND and OR results.
U F	F	U	Note the AND and OR results.
T U	U	T	Note the AND and OR results.
F U	F	U	Note the AND and OR results.
U U	U	U	Note the AND and OR results.

The NULL predicate

Given that NULLs can have undesirable effects on expressions, you can guard against getting unexpected NULL values in your expressions by testing optional expressions to see whether the field evaluates to NULL using the NULL predicate before using it. The NULL predicate has the following form:

```
Body.Invoice.Quantity IS NOT NULL
```

The above expression returns true if the `Invoice.Quantity` field appeared in the message.

The NOT keyword can be omitted to reverse the result.

Logical operators

```
►►──┬─expression_primary──────────────────────────────┬──────────────────►◄
    ├─boolean_expression AND boolean_expression───────┤
    ├─boolean_expression OR boolean_expression────────┤
    ├─NOT boolean_expression──────────────────────────┤
    ├─predicate───────────────────────────────────────┤
    └─expression_primary──IS──┬───────┬──┬─TRUE────┬──┘
                              └─NOT───┘  ├─FALSE───┤
                                         └─UNKNOWN─┘
```

An `expression_primary` must result in a boolean expression. A run-time error is generated if it does not.

The boolean values are combined according to the results documented in Table 38 on page 310.

Note:

NOT(TRUE) = FALSE

NOT(FALSE) TRUE

NOT(UNKNOWN) = UNKNOWN

Repeating fields

The examples so far have been based on a relatively simple message. However, messages are very likely to contain repeating fields, and these are supported by MQSeries Integrator.

Figure 81 defines a message with some repeating fields that illustrate some of these facilities. This message contains product order information, such as might appear in an invoice message, or an online bookshop purchase.

```
<Invoice>
 <Customer>
  <Name>Albert Einstein</Name>
  <InvoiceAddress>
   <Address>Patent Office</Address>
   <Address>Bern</Address>
   <Address>Switzerland</Address>
  </InvoiceAddress>
 </Customer>
```

Figure 81 (Part 1 of 2). Repeating fields in a message

```
 <Item>
  <Book>
   <Title>Principia Mathmatica</Title>
   <Author>Isaac Newton</Author>
   <ISBN>0-520-0881606</ISBN>
  </Book>
  <Price>60</Price>
  <Quantity>1</Quantity>
 </Item>

 <Item>
  <Book>
   <Title>A Brief History of Time</Title>
   <Author>Stephen Hawking</Author>
   <ISBN>0-553-175211</ISBN>
  </Book>
  <Price>7.99</Price>
  <Quantity>1</Quantity>
 </Item>

 <Item>
  <Stationary>pencil</Stationary>
  <Price>0.20</Price>
  <Quantity>200</Quantity>
 </Item>
 <Item>
  <Stationary>paper</Stationary>
  <Price>1.99</Price>
  <Quantity>100</Quantity>
 </Item>
</Invoice>
```

Figure 81 (Part 2 of 2). Repeating fields in a message

Array indices

If you know how many instances there are of a repeating field, and you want to access a specific instance of such a field, you can use an array index as part of a field reference. For example, if you wanted to filter on the first line of an address, to expedite the delivery of an order, you could write an expression such as:

```
Body.Invoice.Customer.InvoiceAddress.Address[1] = '10 Downing Street'
```

The array index [1] indicates that it is the first instance of the repeating field that you are interested in (array indices start at 1). An array index such as this can be used at any point in a field reference, so you could, for example, filter on:

```
Body.Invoice."Item"[1].Quantity > 2
```

If you do not know exactly how many instances of a repeating field there are, you can look at the last instance, or a relative field (for example, the third field from the end). You can refer to the last instance of a repeat by using the special `LAST` array index, as in:

```
Body.Invoice."Item"[LAST]
```

Alternatively, you can use the CARDINALITY function to determine how many instances of a repeating field there are, and use the result to refer to the second to last, for example. The following example shows how to do this:

```
Body.Invoice."Item"[CARDINALITY(Body.Invoice."Item"[]) - 2]
```

In this case, the CARDINALITY function is passed a field reference that ends in `[]`. The meaning of this is "count all instances of the Item field". The [] at the end appears superfluous, because the context indicates that this is the meaning, but its presence is required. This makes the syntax consistent with other instances where it is necessary to refer to "all instances" of something. Remember that array indices start at 1, so the array index in the above example refers to the third-from-last instance of the `Item` field.

Arbitrary repeats: the quantified predicate

It is more likely that you do not know how many instances of a repeating field there are in a message. This is the situation that arises with the `Item` field in the example message. In order to write a filter that takes into account all instances of the `Item` field, you need to use a construct that can iterate over all instances of a repeating field. The quantified predicate allows you to execute a predicate against all instances of a repeating field, and collate the results.

For example, you might want to verify that none of the items that were being ordered had a unit price greater than 1000 pounds. To do this you could write:

```
FOR ALL Body.Invoice."Item"[] AS I (I.Quantity <= 1000)
```

There are several things to note about this example. Firstly, you have to put double quotation marks around the `Item` in the field reference `Body.Invoice.Item[]`. This is because Item is a keyword, and the double quotation marks are necessary to prevent it from being interpreted as a keyword and so giving a syntax error. Secondly, note that the expression "`I.Quantity <= 1000`" compares a character string value (the value of the `Quantity` field from the message) with an integer (the literal 1000). This makes use of the implicit casting of the character string to an integer that occurs in this instance.

With the quantified predicate itself, the first thing to note is the "[]" on the end of the field reference after the "FOR ALL". The square brackets tell you that you are iterating over all instances of the `Item` field.

In some cases, this syntax appears unnecessary because you can get that information from the context, but it is done for consistency with other pieces of syntax.

| The "AS" clause associates the name `I` with the current instance of the repeating
| field. This is similar to the concept of iterator classes used in some object oriented
| languages such as C++. The expression in parentheses is a predicate that is
| evaluated for each instance of the `Item` field.

A fuller description of this example is:

1. Iterate over all instances of the field Item inside `Body.Invoice`.
2. For each iteration:
   a. Bind the name `I` to the current instance of `Item`.
   b. Evaluate the predicate `I.Quantity <= 1000`. If the predicate:
      - Evaluates to TRUE for all of the instances of `Item`, return TRUE.
      - FALSE for any instance of `Item`, return FALSE.
      - Returns a mixture of TRUE and UNKNOWN, return UNKNOWN.

The above is a description of how the predicate is evaluated if the "ALL" keyword is used. An alternative is to specify "SOME", or "ANY", which are equivalent. In this case the quantified predicate returns TRUE if the sub-predicate returns TRUE for any instance of the repeating field. Only if the sub-predicate returns FALSE for all instances of the repeating field does the quantified predicate return FALSE. If a mixture of FALSE and UNKNOWN values is returned from the sub-predicate, an overall value of UNKNOWN is returned.

Another example of the quantified predicate shows how you can take special action when someone orders a copy of "Principia Mathematica". You can write the following filter expression:

```
FOR ANY Body.Invoice."Item"[] AS I (I.Book.Title = 'Principia Mathematica')
```

Note: The sub-predicate evaluates to UNKNOWN for the last two instances of `Item` in the message, because they do not contain a `Book` field. This does not affect the result in the case of an invoice that does contain an order for a copy of "Principia Mathematica", but it does mean that if a copy of that book does not appear on the invoice, the quantified predicate returns the value UNKNOWN.

This is an example of a case where great care must be taken to deal with the possibility of null values appearing. You are therefore recommended to write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice."Item"[] AS I (I.Book IS NOT NULL AND
I.Book.Title = 'Principia Mathematica')
```

The "IS NOT NULL" predicate ensures that if an `Item` field does not contain a `Book`, a FALSE value is returned from the sub-predicate.

Arbitrary repeats: the SELECT expression

Another way of dealing with arbitrary repeats of fields within a message is to use a SELECT expression. Suppose that you want to perform a special action on invoices that have a total order value greater that a certain amount. In order to calculate the total order value of an `Invoice` field, you need to multiply the `Price` fields by the `Quantity` fields in all of the `Items` in the message, and total the result. You can do this using a SELECT expression as follows:

```
(
 SELECT SUM( CAST(I.Price AS DECIMAL) * CAST(I.Quantity AS INTEGER) )
  FROM Body.Invoice."Item"[] AS I
) > 100
```

It is necessary to use CAST expressions to cast the string values of the fields `Price` and `Quantity` into the correct datatypes. The cast of the `Price` field into a decimal produces a decimal value with the "natural" scale and precision, that is, whatever scale and precision is necessary to represent the number.

The SELECT expression works in a similar way to the quantified predicate, and works in much the same way in which a SELECT works in standard database SQL. The FROM clause specifies what we are iterating over, in this case, all `Item` fields in `Invoice`, and establishes that the current instance of `Item` can be referred to using "`I`". This form of SELECT involves a column function, in this case the SUM function, so the SELECT is evaluated by adding together the results of evaluating the expression inside the SUM function for each `Item` field in the `Invoice`. As with standard SQL, NULL values are ignored by column functions, with the exception of the count(*) column function explained below, and a NULL value is returned by the column function only if there are no non-NULL values to combine.

The other column functions that are provided are MAX, MIN, and COUNT. The COUNT function has two forms which work in different ways with regard to NULLs. In the first form you use it much like the SUM function above, so, for example:

```
SELECT COUNT(I.Quantity)
FROM Body.Invoice."Item"[] AS I
```

This expression returns the number of `Item` fields for which the `Quantity` field is non-NULL. That is, the COUNT function counts non-NULL values, in the same way that the SUM function adds non-NULL values. The alternative way of using the COUNT function is as follows:

```
SELECT COUNT(*)
FROM Body.Invoice."Item"[] AS I
```

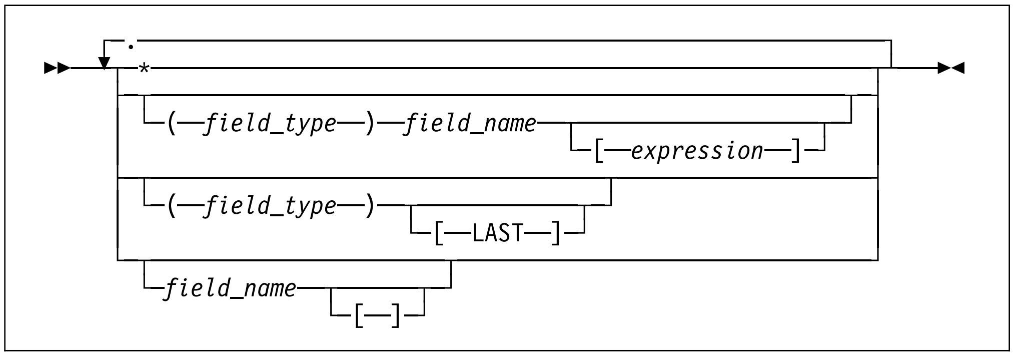
Using COUNT(*) counts the total number of `Item` fields, regardless of whether any of the fields is NULL. The above example is in fact equivalent to using the CARDINALITY function, as in:

```
CARDINALITY(Body.Invoice."Item"[])
```

In all of the examples of SELECT given here, just as in standard SQL, a WHERE clause could have been specified to provide filtering on the fields. Note that the SELECT, FROM and WHERE clauses are the only clauses supported. You cannot specify GROUP BY, HAVING, or ORDER BY, nor can you use the ALL or DISTINCT qualifiers in the SELECT clause.

Field references

The full syntax for field references is defined:



Within this syntax, `field_name` is an identifier and `field_type` is a symbolic constant.

So far, this appendix has explained only those path elements consisting of a `field_name`. The meaning of the first part of the path element is to define search parameters to find the correct syntax element. If only a field name is supplied, that is an instruction to search for elements that have a field name, regardless of the field type that they might have. Similarly, if a path element specifies only a field type, that is an instruction to search for elements that have the given element type, regardless of the name that they might have.

An asterisk in a path element indicates that all syntax elements should be searched, regardless of the field names or field types. These two options are discussed more in the following sections.

Anonymous field names

It is possible to refer to the array of all children of a particular entity by using a path element of "*". So, for example:

```
InputRoot.*[]
```

is a path that identifies the array of all children of `InputRoot`. This is often used in conjunction with an array subscript to refer to a particular child of an entity by position, rather than by name. So, for example:

`InputRoot.*[LAST]`	Refers to the last child of the root of the input message, that is, the "body" of the message.
`InputRoot.*[1]`	Refers to the first child of the root of the input message.

It is useful to be able to find out the name of an entity that has been identified with a path of this kind. To do this, you can use the FIELDNAME function. This function takes a path as its only parameter and returns as a string the field name of the entity to which the path refers. Here are some examples of its usage:

`FIELDNAME(InputRoot.XML)`	Returns `'XML'`.
`FIELDNAME(InputBody)`	Returns the name of the last child of InputRoot, which could be `'XML'`.

`FIELDNAME(InputRoot.*[LAST])`	Returns the name of the last child of InputRoot, which could be `'XML'`.

Field types for the XML parser

There are some instances when it is not enough to identify a field just by name and array subscript. Some message parsers have more complicated models to expose; it is to cope with these cases that an optional type can be associated with element. The message model exposed by the generic XML parser makes heavy use of this facility to deal with the more complicated XML features.

When a type is not present in a path element, it specifies that the type of the syntax element is not important. That is, a path element of "name" matches any syntax element that has a name of "name", regardless of the element type.

In the same way that a path element can specify a name and not a type, a path element can specify a type and not a name. Such a path element matches any syntax element that has the specified type, regardless of name. An example of this is shown below:

```
FIELDNAME(InputBody.(XML.tag)[1])
```

This example returns the name of the first tag in the body of the message (assuming that it is an XML message). For an example of when it is necessary to use types in paths, consider the following generic XML:

```
<tag1 attr1='abc'>
  <attr1>123</attr1>
</tag1>
```

The path "InputBody.tag1.attr1" refers to the attribute called "attr1", because attributes appear before nested tags in a syntax tree generated by an XML parser. In order to refer to the tag called "attr1" it would be necessary to use a path "InputBody.tag1.(XML.tag)attr1". However, it would be advisable always to include types in these situations to be explicit about which entity is being referred to.

Field types for MQRFH2 headers

When you construct MQRFH2 headers in a compute node, there are two types of fields:

1. Fields in the MQRFH2 header structure (for example, Format and NameValueCCSID
2. Fields in the MQRFH2 NameValue buffer (for exampl mcd and psc)

To differentiate between these two possible field types, you must insert a value in front of the referenced field in the MQRFH2 field to identify its type (a value for the NameValue buffer is not required because this is the default). The value you must specify for the header structure is (MQRFH2.Field).

For example:

- To create or change an MQRFH2 Format field, specify the following ESQL:

  ```
  SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR   ';
  ```

- To create or change change the psc folder with a topic:

  ```
  SET OutputRoot.MQRFH2.psc.Topic = 'department';
  ```

- To add an MQRFH2 header to an outgoing message that is to be used to make a subscription request:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
 SET OutputRoot.*[I] = InputRoot.*[I];
 SET I=I+1;
END WHILE;
SET OutputRoot.MQRFH2.(MQRFH2.Field)Version = 2;
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR';
SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
SET OutputRoot.MQRFH2.psc.Command = 'RegSub';
SET OutputRoot.MQRFH2.psc.Topic = "InputRoot"."MRM"."topel";
SET OutputRoot.MQRFH2.psc.QMgrName = 'DebugQM';
SET OutputRoot.MQRFH2.psc.QName = 'PUBOUT';
SET OutputRoot.MQRFH2.psc.RegOpt = 'PersAsPub';
```

Compute node ESQL

The Compute node and the Filter node share a common expression syntax. In its simplest form, a Compute node provides a way of building up a new message using a set of assignment statements. The expressions that appear on the right hand side of the assignment, that is, the source expressions, are expressions of exactly the same form as can appear in a Filter node. But, they are not restricted to returning single boolean values in the same way that a filter expression is.

A Compute node works by constructing a tree representation of a new message based on a list of assignment statements. A new message is always (at least conceptually) constructed, because the message passed to the node must be preserved in its original form (it is not permissible in a message flow to modify pass information back "upstream"). The simplest possible Compute node simply constructs a new message as an exact copy of the input message. Such a Compute node would consist of the following statement

```
SET OutputRoot = InputRoot;
```

There are a number of things to discuss about this example. First, statements in a Compute node are all semicolon (";") terminated. The semicolon is a terminator, and not a separator, so it must appear at the end of every statement, even the last one.

Because there are two messages involved in a Compute node, it is not sufficient to refer to "Root" as can be done in a Filter node where there is only one message. Instead you have to refer to "InputRoot" and "OutputRoot" in a Compute node. You can also refer to "InputBody" in a Compute node in the same way that you can refer to "Body" in a Filter node, though you cannot refer to "OutputBody", because there is no fixed concept of what the "body" of the output message is until the output message has been fully constructed.

The above example causes a complete copy of the input message to be propagated to the output terminal of the Compute node because when the right hand side of an assignment statement consists of a field reference, a complete recursive tree copy is performed to duplicate the tree representation of the input message.

Assignment statement

The general form of an assignment statement is:

```
SET field_reference = expression ;
```

The field reference on the left of the assignment identifies the field in the output message which is to be set, and takes the same form as a field reference in a filter expression, except that it must start with "OutputRoot" or "OutputProperties". The field referenced will be created if it doesn't already exist in the output message; if the value already exists in the output message, its value will be overwritten. Note that when array indices are used in the field reference, only one instance of a particular field will ever get created, so for example if you write as assignment statement starting:

```
SET OutputRoot.XML.Message.Structure[2].Field = ...
```

at least one instance of "Structure" must already exist in the message. That is, the only elements in the tree that are created are ones on a direct path from the root to the element identified by the field reference. A common example of Compute node will consist of a node which makes a modification to a message, either changing a field, or maybe adding a new field to the original message. Such a Compute node would be programmed by statements like the following:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Order.Name = UPPER(OutputRoot.XML.Order.Name);
```

This example simply puts one field in the message into uppercase. The first statement constructs an output message which is a complete copy of the input message (as per the very first simple example). The second statement sets the value of the "Order.Name" field (which it is assumed the message flow writer knows will exist in the input message) to a new value, as defined by the expression on the right.

It is interesting to note what the effect is if the Order.Name field hadn't existed in the original input message. Because it didn't exist in the input message, it won't exist in the output message as generated by the first statement. The expression on the right of the second statement will return NULL, because the field referenced inside the UPPER function call does not exist). Assigning the NULL value to a field has the effect of deleting it if it already exists, and so the effect is that the second statement has no effect.

Whenever the right hand side of an assignment statement is a field reference, a tree copy will be performed. This does not only happen when a whole message is copied. For example:

```
SET OutputRoot = InputRoot;
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.InputMessage = InputRoot;
```

All these result in a tree copy.

DECLARE statement

The DECLARE statement declares a simple scalar variable that can be used to store some temporary value. The syntax of the declare statements is:

```
DECLARE variable_name datatype
```

where datatype is one of the following:

CHARACTER or CHAR
FLOAT
DATE
DECIMAL
INTEGER or INT
INTERVAL
TIME
TIMESTAMP
GMTTIME
GMTTIMESTAMP
BIT
BLOB
BOOLEAN

For an example of the DECLARE statement see the example in the description of the WHILE statement.

WHILE statement

A WHILE statement executes a sequence of statements repeatedly while the controlling predicate evaluates to true. The same caveats apply to using the WHILE statement as apply in any language, that is, it is up to you to ensure that the loop will terminate. Note that if the control expression evaluates to unknown the loop terminates: unknown and false are treated in the same way in this respect. The WHILE statement takes the following form:

```
WHILE predicate DO
  controlled statements
END WHILE;
```

For example:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I <= 10 DO
  SET I = I + 1;
END WHILE;
```

IF statement

An IF statement controls execution of one set of statements or another based on the result of evaluating a predicate.

Note that if the control expression evaluates to unknown, the "else" statements are executed; "unknown" is treated the same as false.

The IF statement takes one of the following forms:

```
IF condition THEN
  controlled statements
END IF;
```

or:

```
IF condition THEN
  controlled statements 1
ELSE
  controlled statements 2
END IF;
```

Using the compute node for data conversion

You can use the ESQL within a compute node to provide data conversion for code page and encoding of messages. You must set MQMD CCSID and Encoding fields of the output message, plus the CCSID and Encoding of any headers, to the required target value.

The following example illustrates what is required for a CWF message to pass from MQSeries Integrator to IMS on OS/390®.

1. You have defined the input message in XML and are using an MQRFH2 header. The header must be removed before the message is passed to IMS.
2. The message passed to IMS must have MQIIH header, and must be in the OS/390 codepage. This message is defined in the MRM and has identifier m_IMS1. The PIC X fields in this message must be defined as logical type string for EBCDIC <-> ASCII conversion to take place. If they are logical type binary, no data conversion occurs.
3. The message received from IMS is also defined in the MRM and has identifier m_IMS2. The PIC X fields in this message must be defined as logical type string for EBCDIC <-> ASCII conversion to take place. If they are logical type binary, no data conversion occurs.
4. The reply message must be converted to the Windows NT codepage. The MQIIH header is retained on this message.
5. You have created a message flow that contains:
   a. The outbound flow, **MQInput1** —> **Compute1** —> **MQOutput1**.
   b. The inbound flow, **MQInput2** —> **Compute2** —> **MQOutput2**.
6. You must set up the ESQL in Compute1 (outbound) node as follows:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) - 1 DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;
SET OutputRoot.MQMD.CodedCharSetId = 500;
SET OutputRoot.MQMD.Encoding = 785;
SET OutputRoot.MQMD.Format = 'MQIMS   ';
SET OutputRoot.MQIIH.StrucId = 'IIH ';
SET OutputRoot.MQIIH.Version = 1;
SET OutputRoot.MQIIH.StrucLength = 84;
SET OutputRoot.MQIIH.Encoding = 785;
SET OutputRoot.MQIIH.CodedCharSetId = 500;
SET OutputRoot.MQIIH.Format = 'MQIMSVS ';
SET OutputRoot.MQIIH.Flags = 0;
SET OutputRoot.MQIIH.LTermOverride = '        ';
SET OutputRoot.MQIIH.MFSMapName = '        ';
SET OutputRoot.MQIIH.ReplyToFormat = 'MQIMSVS ';
SET OutputRoot.MQIIH.Authenticator = '        ';
SET OutputRoot.MQIIH.TranInstanceId = X'00000000000000000000000000000000';
SET OutputRoot.MQIIH.TranState = ' ';
SET OutputRoot.MQIIH.CommitMode = '0';
SET OutputRoot.MQIIH.SecurityScope = 'C';
SET OutputRoot.MQIIH.Reserved = ' ';
SET OutputRoot.MRM.e_elen08 = 30;
SET OutputRoot.MRM.e_elen09 = 0;
SET OutputRoot.MRM.e_string08 = InputBody.e_string01;
SET OutputRoot.MRM.e_binary02 = X'31323334353637383940';
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'm_IMS1';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

7. You must set up the ESQL in Compute2 (inbound) node as follows:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;
SET OutputRoot.MQMD.CodedCharSetId = 437;
SET OutputRoot.MQMD.Encoding = 546;
SET OutputRoot.MQMD.Format = 'MQIMS   ';
SET OutputRoot.MQIIH.CodedCharSetId = 437;
SET OutputRoot.MQIIH.Encoding = 546;
SET OutputRoot.MQIIH.Format = '        ';
SET OutputRoot.MRM = InputBody;
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'm_IMS2';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

You do not have to set any specific values for the MQInput1 node properties because the message and message set are identified in the MQRFH2 header, and no conversion is required.

You must set values for Domain, set, type and format in the MQInput node for the inbound message flow (MQInput2). You do not need to set conversion parameters.

Using the compute node for message transformation

You can use the ESQL within a compute node to transform a message from one format to another.

For example, if you want to transform a generic XML message into an MRM message, you can:

1. Add the MRM message to Output Messages on the basic tab of the compute node properties dialog.
2. If you want to retain the headers of the message, select **Copy message headers**.
3. Select the **Use as message body** check box. This generates ESQL similar to:

   ```
   SET OutputRoot.Properties.MessageSet = 'DHOP5F709S001';
   SET OutputRoot.Properties.MessageType = 'test_message';
   ```

   Note that it is the message identifier that is required in the MessageType field.
4. Specify the output format of the message (this must be one of CWF, PDF, or XML). For example:

   ```
   SET OutputRoot.Properties.MessageFormat = 'CWF';
   ```
5. Specify the new message domain (in this transformation, this step is not necessary because MRM is the default, but you are recommended to include this for completeness):

   ```
   SET OutputRoot.Properties.MessageDomain = 'MRM';
   ```
6. Create ESQL statements to populate your output message, either manually or by using drag and drop to generate automatic mappings.

The same principles apply for other message transformations.

More complicated SELECTs: ROWs and LISTs

The examples of the SELECT expression given above all involved column functions, because these are a common form in filter expressions However, more general subselects can also be used. These operate in much the same way that standard SQL selects do, but the "result sets" that are generated from the different forms need some discussion. As a way of illustrating the various forms that a SELECT clause can take, consider the following examples of assigning the results of database queries to fields in a message using a Compute node.

Example 1

Using the Control Center, create a message flow consisting of an MQInput node wired to a Compute node, wired to an MQOutput node. Configure the queue names on the MQInput node and MQOutput node to point to suitable queues, and set the Message Domain attribute on the Defaults tab of the MQInput node property editor to be "XML". Configure the Compute node using the following ESQL statements:

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result[] =
(SELECT T.Field4, T.Structure1 FROM InputBody.Test.Input[] AS T);
```

Deploy the message flow to a suitable broker, and then send a simple trigger message like the following to the input queue:

```
<Test>
 <Input>
  <Field1>value1</Field1>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
  <Field4>value4</Field4>
 </Input>
 <Input>
  <Field1>value5</Field1>
  <Structure1>
   <Field2>value6</Field2>
   <Field3>value7</Field3>
  </Structure1>
  <Field4>value8</Field4>
 </Input>
</Test>
```

You should receive the following message on the output queue:

```
<Test>
 <Result>
  <Field4>value4</Field4>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
 </Result>
 <Result>
  <Field4>value8</Field4>
  <Structure1>
   <Field2>value6</Field2>
   <Field3>value7</Field3>
  </Structure1>
 </Result>
</Test>
```

The order in which the tags appear inside the Result tag reflects the order in which the items appeared in the select clause, not the order in which the fields appeared in the original message. Also, the Structure1 fields are copied in their entirety from the input message:that is, a tree copy has been performed. You can of course rename the fields by using an AS clause after some or all of the items in the SELECT clause.

Example 2

The following example shows the use of the ITEM keyword, which selects one item and creates a single value. (Example 1 shows a structure that creates a single field.)

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result[] =
(SELECT ITEM T.Field1 FROM InputBody.Test.Input[] AS T);
```

Sending the same trigger message will result in a message on the output queue which looks like this:

```
<Test>
 <Result>value1</Result>
 <Result>value5</Result>
</Test>
```

Comparing this message to the one which is produced if the ITEM keyword is omitted:

```
<Test>
 <Result>
  <Field1>value1</Field1>
 </Result>
 <Result>
  <Field1>value5</Field1>
 </Result>
</Test>
```

illustrates the effect of the ITEM keyword. The evaluation of the ESQL expressions happens independently of any information about the schema of the target message. In the case of generating a generic XML message there is no message schema for the message being generated, so the structure of the message that is generated must be defined entirely by the ESQL.

Example 3

The two examples above have both specified a list as the source of the SELECT in the FROM clause (so the field reference had a [] at the end), and so in general the SELECT will generate a list of results. Because of this it was necessary to specify a list as the target of the assignment (thus the "Result[]" as the target of the assignment). However, often you will know that the WHERE clause that you specify as part of the SELECT will only return true for one item in the list. In this case the "THE" keyword can be used to indicate this. The following shows the effect of using the THE keyword

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result =
THE (SELECT T.Field4, T.Structure1 FROM InputBody.Test.Input[]
 AS T WHERE T.Field1 = 'value1';
```

The "THE" keyword means that the target of the assignment becomes "OutputRoot.XML.Test.Result" (the "[]" is no longer necessary, or even allowed). This results in the following message:

```
<Test>
 <Result>
  <Field4>value4</Field4>
  <Structure1>
   <Field2>value2</Field2>
   <Field3>value3</Field3>
  </Structure1>
 </Result>
</Test>
```

Example 4

Using selects for projection:

```
SET OutputRoot.XML.Projection =
 (SELECT M.field1,
   M.field2,
   CAST(M.field3 AS INTEGER) *CAST(M.field4 AS INTEGER) AS field5
 FROM InputBody.Message AS M);
```

equivalent to:

```
SET OutputRoot.XML.Projection.field1 = InputBody.Message.field1;
SET OutputRoot.XML.Projection.field2 = InputBody.Message.field2;
SET OutputRoot.XML.Projection.field5 =
    CAST(InputBody.Message.field3 AS INTEGER)
  * CAST(InputBody.Message.field4 AS INTEGER);
```

Example 5

The FROM clause is not restricted to having one item. Specifying multiple items in the FROM clause has the usual "joining" effect that it does in standard SQL. For example:

```
SELECT A.a, B.b
FROM  InputBody.Test.A[], InputBody.Test.B[]
```

The following message:

```
<Test>
 <A>
  <a>1</a>
 </A>
 <A>
  <a>2</a>
 </A>
 <B>
  <b>3</b>
 </B>
 <B>
  <b>4</b>
 </B>
</Test>
```

produces the following output message:

```
<Test>
 <Result>
  <a>1</a>
  <b>3</b>
 </Result>
 <Result>
  <a>1</a>
  <b>4</b>
 </Result>
 <Result>
  <a>2</a>
  <b>3</b>
 </Result>
 <Result>
  <a>2</a>
  <b>4</b>
 </Result>
</Test>
```

Note that because repeating fields can be nested in messages, it is possible to write an expression such as:

```
SELECT A.a, B.b
FROM InputBody.Test.A[], A.B[]
```

In this case, the following message:

```
<Test>
  <A>
   <a>1</a>
   <B>
    <b>2</b>
   </B>
   <B>
    <b>3</b>
   </B>
  </A>
  <A>
   <a>4</a>
   <B>
    <b>5</b>
   </B>
   <B>
    <b>6</b>
   </B>
  </A>
 </Test>
```

produces the following output:

```
<Test>
 <Result>
  <a>1</a>
  <b>2</b>
 </Result>
 <Result>
  <a>1</a>
  <b>3</b>
 </Result>
 <Result>
  <a>4</a>
  <b>5</b>
 </Result>
 <Result>
  <a>4</a>
  <b>6</b>
 </Result>
</Test>
```

Example 6

You can join between a list and a non-list, two non-lists, and so on.

```
OutputRoot.XML.Test.Result1[] =
  (SELECT ... FROM InputBody.Test.A[], InputBody.Test.b);
OutputRoot.XML.Test.Result1 =
  (SELECT ... FROM InputBody.Test.A, InputBody.Test.b);
```

Note carefully the location of the "[]" in each case. Of course, any number of items can be specified in the FROM list, not just one or two, and in each case if any of the items specify "[]" to indicate a list of items, the SELECT will generate a list of results (the list may contain only one item, but the SELECT can potentially return a list of items), and so the target of the assignment must specify a list (so must end in "[]" or else the THE keyword must be used if is known that the WHERE clause will guarantee that only one combination is matched.

Example 7

A SELECT with a column function is not the only form of SELECT that can be used in a scalar expression. You can make a SELECT return a scalar value by issuing both the THE and ITEM keywords as in:

```
1 + THE(SELECT ITEM T.a FROM Body.Test.A[] AS T WHERE T.b = '123')
```

Example 8

Selecting from a list of scalars, consider the sample message:

```
<Test>
 <A>1</A>
 <A>2</A>
 <A>3</A>
 <A>4</A>
 <A>5</A>
</Test>
```

and the ESQL statements

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.A[] = (SELECT A FROM InputBody.Test.A[]
WHERE CAST(A AS INTEGER) BETWEEN 2 AND 4);
```

Other expressions

In all of the examples above, simple field references are used in the SELECT clause. As usual, more complicated expressions can be used. In these cases, an AS clause must be used to give a name to the computed field as in:

```
SELECT T.Price, T.Quantity, T.Price * T.Quantity AS TotalValue
FROM Body.Invoice."Item"[]
```

EXISTS predicate

You can use the EXISTS predicate to test whether a WHERE clause successfully matches any items of a repeating structure in the same way as you can use standard database SQL. The form of the EXISTS predicate is:

```
EXISTS(SELECT * FROM something WHERE predicate)
```

Querying external databases

Queries against external databases can be done in much the same way as can be done in, for example, embedded SQL.

In order to include a query against an external database in a Filter or Compute node, the node must be configured with the connection information for the database. This consists of an ODBC datasource name. It is up to the MQSeries Integrator or database administrator to ensure that a suitable ODBC datasource has been created on the systems on which the brokers, to which the message flows are deployed, are running.

The connection to the database is performed using the database user ID and password supplied on the **mqsicreatebroker** command that created the individual broker. The MQSeries Integrator or database administrator must therefore ensure that that user has sufficient database privileges to query the required database tables. If not, a run-time error is generated by the broker when it attempts to process a message and attempts to connect to the database for the first time.

Whilst the standard SQL SELECT syntax is supported for queries to external database, there are a number of points to be borne in mind. It is necessary to prefix the name of the table with the keyword "Database" in order to indicate that the SELECT is to be targeted at the external database, rather than at a repeating structure in the message.

Therefore the basic form of database SELECT is:

```
SELECT ...
FROM Database.TABLE1
WHERE ...
```

If necessary a schema name can be given:

```
SELECT ...
FROM Database.SCHEMA.TABLE1
WHERE ...
```

where SCHEMA is the name of the schema in which the table TABLE1 is defined.

References to column names must be qualified with either the table name or the correlation name defined for the table by the FROM clause. So, where you could normally execute a query such as:

```
SELECT column1, column2 FROM table1
```

| it is necessary to write one of the following two forms:

```
SELECT T.column1, T.column2 FROM Database.table1 AS T

SELECT table1.column1, table2.column2 FROM Database.table1
```

This is necessary in order to distinguish references to database columns from any references to fields in a message which may also appear in the SELECT:

```
SELECT T.column1, T.column2 FROM Database.table1
  AS T WHERE T.column3 = Body.Field2
```

The standard ‘select all’ SQL option is supported in the SELECT clause. If you use this option, you must qualify the column names with either the table name or the correlation name defined for the table. For example:

```
SELECT T.* FROM Database.Table1 AS T
```

The following examples illustrate how the results sets of external database queries are represented in MQSeries Integrator. The results of database queries are assigned to fields in a message using a Compute node.

Example 1

Create a message flow consisting of an MQInput node wired to a Compute node, wired to an MQOutput node. Configure the queue names on the MQInput node and MQOutput node to point to suitable queues, and set the Message Domain attribute on the Defaults tab of the MQInput node property editor to be "XML".

Create a database table called USERTABLE with two char(6) data type columns (or equivalent), called Column1 and Column2. Insert two rows into the table so that it looks like this:

	Column1	**Column2**
Row 1	value1	value3
Row 2	value2	value4

Add a database table input to the Compute node by clicking the **Add** input button on the properties pane of the node and entering the ODBC Data Source Name and table name. The user id and password specified when you created the broker is used for accessing the database, therefore you must ensure that this id and password pair have appropriate permissions within the DBMS.

You are also recommended to ensure that you include the schema name when you create a table, and as the second component of the database table reference (for example, `Database.user1.USERTABLE`) in the Compute node ESQL you specify. This avoids potential confusion that some databases might encounter.

For example, if you create your database table as user id `user1`, but specified user id `user2` when you created the broker, you might find that the broker attempts to access table `user2.USERTABLE`, which does not exist, rather than `user1.USERTABLE`, which does.

You can vary the names of the fields produced by explicitly listing the columns that you want to extract. How you do this depends partly on your database system. Most database systems are case not sensitive with regard to database names. In other words, even though a column might be called "COLUMN1", you can refer to it in a SELECT as "column1".

Configure the Compute node using the following ESQL statements:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.Column1, T.Column2 FROM Database.USERTABLE AS T);
```

To trigger the SELECT, you must send in a trigger message with an XML body that is of the following form:

```
<Test>
 <Result>
  <Column1>value1</Column1>
  <Column2>value2</Column2>
 </Result>
 <Result>
  <Column1>value3</Column1>
  <Column2>value4</Column2>
 </Result>
</Test>
```

The exact structure of the XML is not important, but the enclosing tag must be `<Test>`. If it is not, the ESQL statements will result in top-level enclosing tags being formed, which is not valid XML.

The names of the tags produced exactly match how you referred to the columns in the select clause.

Example 2

If the database system is case sensitive, you must use an alternative approach. This approach is also necessary if you want to change the name of the generated field to something different:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.COLUMN1 AS Column1, T.COLUMN2 AS Column2
  FROM Database.USERTABLE AS T);
```

This example produces the same message as Example 1 above.

Example 3

Suppose that the Compute node were configured using the following ESQL statements:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
 (SELECT ITEM T.Column1 FROM Database.USERTABLE AS T);
```

The same trigger message will produce the following message:

```
<Test>
 <Result>value1</Result>
 <Result>value3</Result>
</Test>
```

The following message is produced if the ITEM keyword is omitted:

```
<Test>
 <Result>
  <Column1>value1</Column1>
 </Result>
 <Result>
  <Column1>value3</Column1>
 </Result>
</Test>
```

Comparing this to the previous generated message illustrates the effect of the ITEM keyword. The evaluation of the ESQL expressions happens independently of any information about the schema of the target message. In the case of generating a generic XML message, there is no message schema for the message being generated, so the structure of the message that is generated must be defined entirely by the ESQL.

Example 4

This example illustrates the use of the WHERE clause: the message generated by this statement is identical to that generated by the previous example.

```
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result =
 THE (SELECT ITEM T.Column1 FROM Database.USERTABLE AS T
 WHERE T.Column2 = ‘value2’);
<Test>
 <Result>value1   </Result>
</Test>
```

Database node statements

The syntax of the statements that are accepted by the Database node is a superset of the statements that are accepted by a Compute node.

Like the Compute node, the Database node is configured using a series of statements. All of the normal compute statements such as SET, WHILE, DECLARE, and IF can be used to control the flow of the series of statements.

Unlike the Compute node, however, the Database node propagates the message that it receives at its input terminal to its output terminal unchanged. This means that, like the Filter node, there is only one message to be referred to in a Database node.

Because you can't modify any part of any message, the assignment statement (the SET statement, not the SET clause of the INSERT statement) can only assign values to temporary variables. Therefore the scope of actions you can take with an assignment statement is limited.

The following sections discuss the extra statements that can be used in a Database node.

INSERT statement

An INSERT statement can be used to add new rows to an external database.

```
►►──INSERT INTO──database.──┬────────────────┬──table_name──────────►
                            └─schema_name.───┘

►──┬──────────────────────────────┬─────────────────────────────────►
   │      ┌─,───────────┐         │
   └─(──▼─column_name─┴──)─┘

►──VALUES──(──▼─scalar_expression─┴──)──────────────────────────────►◄
              └─,─────────────────┘
```

The optional column name list identifies a list of columns in the target table into which values are to be inserted. Any columns not mentioned in the column name list will have their default values inserted.

A run-time error can be generated if problems occur during the insert operation. For example the database table may have constraints defined which the insert operation may violate. In these cases, an attempt will be made to propagate the original message that was received by the node to the failure terminal on the node.

Example

The following example assumes that the dataSource property on the Database node has been configured and that the database identified by that datasource has a table called "TABLE1" with columns A, B, and C. Given a message that has the following generic XML body:

```
<A>
 <B>1</B>
 <C>2</C>
 <D>3</D>
</A>
```

the following INSERT statement will insert a new row into the table with the values (1, 2, 3).

```
INSERT INTO Database.TABLE1(A, B, C) VALUES (Body.A.B, Body.A.C, Body.A.D)
```

DELETE statement

A DELETE statement deletes rows from a table in an external database based on a search condition.

```
►►──DELETE FROM──database.──┬────────────────┬──table_name──────────►
                            └─schema_name.──┘

►──┬─────────────────────────┬──┬────────────────────────────┬──────►◄
   └─AS──correlation_name──┘  └─WHERE──search_condition──┘
```

A correlation name is created that can be used inside the search condition to refer to the values of columns in the table. This correlation name is either the name of the table (without the data source qualifier) or the explicit qualifier specified.

Example

Suppose that you have Database node that has been configured with a connection to a table SHAREHOLDINGS. The following statement could be written to configure the Database node:

```
DELETE FROM Database.SHAREHOLDINGS AS H
 WHERE H.ACCOUNTNO = Body.AccountNumber
```

This will remove all rows from the SHAREHOLDINGS table where the ACCOUNTNO field in the table is equal to the AccountNumber in the message.

UPDATE statement

An UPDATE statement will update the values of specified columns in a table in an external database.

```
►►──UPDATE──database.──┬────────────────┬──table_name──────────────►
                       └─schema_name.──┘

                                         ┌─,─────────────────────────────┐
►──┬─────────────────────────┬──SET────▼─column_name──=──expression─┴───►
   └─AS──correlation_name──┘

►──┬────────────────────────────┬───────────────────────────────────►◄
   └─WHERE──search_condition──┘
```

Example 1

This example updates the PRICE column of the row in the STOCKPRICES table whose COMPANY column matched the value given in the Company field in the message that the Database node is processing.

```
UPDATE Database.StockPrices AS SP
 SET PRICE = Body.StockPrice
 WHERE SP.COMPANY =Body.Company
```

Example 2

In this example, the "INV.QUANTITY" in the right hand side of the assignment refers to the previous value of the column before any updates have taken place.

```
UPDATE Database.INVENTORY AS INV
 SET QUANTITY = INV.QUANTITY - Body.QuantitySold
 WHERE INV.ITEMNUMBER = Body.ItemNumber
```

Example 3

This example shows multiple column updates.

```
UPDATE Database.table AS T
 SET column1 = T.column1+1,
     column2 = T.column2+1;
```

Compare the syntax to the way you assign to multiple fields in a compute:

```
SET field = expression;
SET field = expression;
```

Note also the form of the assignment clause: the column on the left of the assignment must be a single identifier. It must **not** be qualified with a table name or correlation name. In contrast, any column references to the right of the assignment **must** be qualified with a table name or correlation name.

PASSTHRU statement

The behavior of the PASSTHRU function depends on whether it is passed one, two, or more parameters. The first parameter of the PASSTHRU function must always be an ESQL expression that either is, or evaluates to, a string. You must use question marks in the string to denote where any parameter substitution is required.

If only one other parameter is passed, that parameter evaluates to one of the following:

- A single scalar value. If this is the case, it is inserted into the first parameter marker.
- A list of values. If this is the case, the list items are inserted in order into each of the parameter markers within the string.

If two or more other parameters are passed, each parameter is bound to the corresponding question mark in the statement string: that is, the first parameter is bound to the first question mark, the second parameter is bound to the second question mark, and so on.

| Here are some examples that illustrate different ways of using the PASSTHRU
| statement.

```
| SET OutputRoot.XML.Result.Data[] = PASSTHRU('SELECT * FROM user1.stocktable');
```

```
| PASSTHRU('DELETE FROM user2.AccountData WHERE AccountId = ?', InputBody.Data.Account.Id);
```

```
| SET OutputRoot.XML.Result.Data
|   = PASSTHRU('SELECT AccountNum FROM user2.AccountData
|     WHERE AccountId = ?', InputBody.Data.Account.Id);
```

```
| SET OutputRoot.XML.Result.Data[]
|   = PASSTHRU('SELECT AccountNum FROM user2.AccountData
|              WHERE AccountId IN (? , ? , ?)',
|            InputBody.Data.Account.Id[]);
```

```
| PASSTHRU('INSERT INTO user1.stocktable (stock_id, quantity)
|          values (?, ?)', InputBody.Transaction.Id, InputBody.Transaction.Quantity);
```

| You must take the following points into consideration when you construct string
| literals that you will use as the first parameter of the PASSTHRU function:

- | If the ESQL statement that you want to execute against the database contains
  | a single quote, you must escape the single quote when you define the string
  | literal.

  | For example, if you want to execute the following ESQL statement:

  ```
  INSERT INTO TABLE1 VALUES('abc', 'def')
  ```

  you can use the following PASSTHRU statement:

  ```
  PASSTHRU('INSERT INTO TABLE1 VALUES(''abc'', ''def'')');
  ```

  The use of double single quotation marks is required for a definition of a string literal containing a single quote.

- You must take care with statements that you want to execute that are particularly long. You cannot split a string literal across multiple lines.

  For example, if you want to execute the following ESQL statement:

  ```
  SELECT a, b, c
  FROM table1
  WHERE d = 123
  ```

  you cannot use the following form because the string literal contains new lines, which is prohibited.

  ```
  PASSTHRU('SELECT a, b, c
                   FROM table1
                   WHERE d = 123')
  ```

Instead you must use:

```
PASSTHRU('SELECT a, b, c ' ||
                   'FROM table1 ' ||
                   'WHERE d = 123')
```

- You must include trailing spaces in the individual string literals to avoid defining a string containing the text:

```
'SELECT a, b, c FROM table1 WHERE d = 123'
```

Function reference

This section provides a reference summary of functions discussed in this appendix.

CASTs

CAST specifications

```
►►──CAST──(──expression──AS──┬─CHAR──────────────────────────┬──)────►◄
                             ├─CHARACTER─────────────────────┤
                             ├─FLOAT─────────────────────────┤
                             ├─DATE──────────────────────────┤
                             ├─DECIMAL(precision, scale)─────┤
                             ├─INT───────────────────────────┤
                             ├─INTEGER───────────────────────┤
                             ├─INTERVAL interval qualifier───┤
                             ├─TIME──────────────────────────┤
                             ├─TIMESTAMP─────────────────────┤
                             ├─GMTTIME───────────────────────┤
                             ├─GMTTIMESTAMP──────────────────┤
                             ├─BIT───────────────────────────┤
                             ├─BLOB──────────────────────────┤
                             └─BOOLEAN───────────────────────┘
```

Note: For *interval qualifier* formats, see Table 36 on page 303.

A CAST specification returns its first operand cast to the type specified by the data_type. The conversion that is done is the default conversion, More complicated conversions can be performed using user defined functions. In all cases if the source expression is NULL, the result will be NULL. If the source value is not compatible with the target datatype, or if the source value is of the wrong format, a run-time error is generated.

Supported CASTs

A CAST is not supported between every combination of datatypes. Those that are supported are listed below, along with the effect of the CAST.

Table 39 (Page 1 of 5). Supported CASTs

Source datatype	Target datatype	Effect
CHARACTER	BOOLEAN	The character string is interpreted in the same way that a boolean literal is interpreted. That is, the character string must be one of the strings TRUE, FALSE, UNKNOWN (in any case combination).
CHARACTER	FLOAT	The character string is interpreted in the same way as a floating point literal is interpreted.
CHARACTER	DATE	The character string must conform to the rules for a date literal or for the date string. That is, the character string can be either DATE '1998-11-09' or 1998-11-09.

Table 39 (Page 2 of 5). Supported CASTs

Source datatype	Target datatype	Effect
CHARACTER	DECIMAL	The character string is interpreted in the same way as an exact numeric literal is interpreted to form a temporary decimal result with a scale and precision defined by the format of the string. This is then converted into a decimal of the specified precision and scale, with a run-time error being generated if the conversion would result in loss of significant digits.
CHARACTER	INTEGER	The character string is interpreted in the same way as an integer literal is interpreted.
CHARACTER	INTERVAL	The character string must conform to the rules for an interval literal with the same interval qualifier as specified in the CAST specification, or it must conform to the rules for an interval string that apply for the specified interval qualifier.
CHARACTER	TIME	The character string must conform to the rules for a time literal or for the time string. That is, the character string can be either TIME '09:24:15' or 09:24:15.
CHARACTER	TIMESTAMP	The character string must conform to the rules for a timestamp literal or for the timestamp string. That is, the character string can be either TIMESTAMP '1998-11-09 09:24:15' or 1998-11-09 09:24:15.
CHARACTER	GMTTIME	The character string must conform to the rules for a GMT time literal or for the time string. That is, the character string can be either GMTTIME '09:24:15' or 09:24:15.
CHARACTER	GMTTIMESTAMP	The character string must conform to the rules for a GMT timestamp literal or for the timestamp string. That is, the character string can be either GMTTIMESTAMP '1998-11-09 09:24:15' or 1998-11-09 09:24:15.
CHARACTER	BIT	The character string must conform to the rules for a bit string literal or to the rules for the contents of the bit string literal. That is, the character string can be of the form B'bbbbbbb' or bbbbbb (where 'b' can be either '0' or '1').
CHARACTER	BLOB	The character string must conform to the rules for a binary string literal or to the rules for the contents of the binary string literal. That is, the character string can be of the form X'hhhhhh' or hhhhhh (where 'h' can be any hexadecimal digit characters).
BOOLEAN	CHARACTER	If the source value is TRUE, the result is the character string 'TRUE'. If the source value is FALSE, the result is the character string 'FALSE'. Because the UNKNOWN boolean value is the same as the NULL value for booleans, the result will be the NULL character string value if the source value is UNKNOWN.
FLOAT	CHARACTER	The result is the shortest character string that conforms to the definition of an approximate numeric literal and whose mantissa consists of a single digit that is not '0', followed by a period and an unsigned integer, and whose interpreted value is the value of the double.

Table 39 (Page 3 of 5). Supported CASTs

Source datatype	Target datatype	Effect

DATE	CHARACTER	The result is a string conforming to the definition of a date literal, whose interpreted value is the same as the source date value.
		Example:
		`CAST(DATE '1998-11-09' AS CHAR)`
		would return
		`DATE '1998-11-09'`

DECIMAL	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the decimal.
INTEGER	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the integer.

INTERVAL	CHARACTER	The result is a string conforming to the definition of an interval literal, whose interpreted value is the same as the source interval value.
		Example:
		`CAST(INTERVAL '4' YEARS AS CHAR)`
		would return
		`INTERVAL '4' YEARS`

TIME	CHARACTER	The result is a string conforming to the definition of a time literal, whose interpreted value is the same as the source time value.
		Example:
		`CAST(TIME '09:24:15' AS CHAR)`
		would return
		`TIME '09:24:15'`

TIMESTAMP	CHARACTER	The result is a string conforming to the definition of a timestamp literal, whose interpreted value is the same as the source timestamp value.
		Example:
		`CAST(TIMESTAMP '1998-11-09 09:24:15' AS CHAR)`
		would return
		`TIMESTAMP '1998-11-09 09:24:15'`

GMTTIME	CHARACTER	The result is a string conforming to the definition of a gmttime literal whose interpreted value is the same as the source value. The result string will have the form GMTTIME 'hh:mm:ss'.
GMTTIMESTAMP	CHARACTER	The result is a string conforming to the definition of a gmttimestamp literal whose interpreted value is the same as the source value. The result string will have the form GMTTIMESTAMP 'yyyy-mm-dd hh:mm:ss'.
BIT	CHARACTER	The result is a string conforming to the definition of a bit string literal whose interpreted value is the same as the source value. The result string will have the form B'bbbbbb' (where b is either '0' or '1').

Table 39 (Page 4 of 5). Supported CASTs

Source datatype	Target datatype	Effect
BLOB	CHARACTER	The result is a string conforming to the definition of a binary string literal whose interpreted value is the same as the source value. The result string will have the form `X'hhhh'` (where h is any hexadecimal digit character).
TIME	GMTTIME	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIME	TIME	The result value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIMESTAMP	TIMESTAMP	The result value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE).
TIMESTAMP	GMTTIMESTAMP	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE).
INTEGER or DECIMAL	FLOAT	The number is converted, with rounding if necessary.
FLOAT	INTEGER or DECIMAL	If the conversion would not lead to loss of leading significant digits, the conversion will happen with the number being rounded as necessary. If the conversion would lead to loss of leading significant digits, a run-time error is generated. Loss of significant digits can occur when converting an approximate numeric value to an integer, or to a decimal whose precision is not sufficient.
INTEGER or DECIMAL	INTEGER or DECIMAL	If the conversion would not lead to loss of leading significant digits, the conversion will happen with the number being rounded as necessary. If the conversion would lead to loss of leading significant digits, a run-time error is generated. Loss of significant digits can occur when converting (say) a decimal to another decimal with insufficient precision, or an integer to a decimal with insufficient precision.
INTERVAL	INTERVAL	Year-month intervals are only convertible to year-month intervals, and day-second intervals are only convertible to day-second intervals. The conversion is done by converting the source interval into a scalar in units of the least significant field of the target interval qualifier. This value is then normalized into an interval with the target interval qualifier. So for example to convert an interval which has the qualifier MINUTE TO SECOND into an interval with the qualifier DAY TO HOUR, the source value is converted into a scalar in units of hours, and this value is then normalized into an interval with qualifier DAY TO HOUR.
INTERVAL	INTEGER or DECIMAL	If the interval value has a qualifier that has only one field, the result is an exact numeric with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a run-time error is generated.
INTEGER or DECIMAL	INTERVAL	If the interval qualifier specified has only one field, the result will be an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a run-time error is generated.

Table 39 (Page 5 of 5). Supported CASTs

Source datatype	Target datatype	Effect
TIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source time value.
TIMESTAMP	TIME	The result is a value whose fields consist of the time fields of the source timestamp value.
TIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source timestamp value.

CAST expressions

A cast expression is used to cast, or change, a value of one datatype into a corresponding value of another datatype (within the range of supported CASTS listed in Table 39 on page 338).

CAST expressions are used often when dealing with generic XML messages: all fields in a generic XML message have string values, therefore to perform arithmetic calculations or datetime comparisons (for example), the string value of the field must first be cast into a value of the appropriate type.

For example, If you wanted to filter on trade messages where the date of the trade was today, you could write the following expression:

```
CAST(Body.Trade.Date AS DATE) = CURRENT_DATE
```

In this example, the string value of the `Date` field in the message is converted into a date value, and then compared with the current date. The conversion is based on the same character string format as is used for specifying date literals. This is generally the case when casting character strings to other types.

Note: It is not always necessary to cast values between types. Some casts are done implicitly. For example, numbers are implicitly cast between the three numeric types for the purposes of comparison and arithmetic. Character strings are also implicitly cast to other datatypes for the purposes of comparison.

There are three situations in which a data value of one type is implicitly (that is, without an explicit CAST instruction) cast to another type. The behavior and restrictions of the implicit cast are the same as described above for explicit CAST, except where noted in the following sections.

Implicit CASTs for comparisons

The standard SQL comparison operators >, <, >=, <=, =, <> are supported for comparing two values in ESQL.

When the data types of the two values are not the same, one of them can be implicitly cast to the type of the other to allow the comparison to proceed. In the table below, the vertical axis represents the left hand operand, the horizontal axis represents the right hand operand.

An "L" means that the right hand operand is cast to the type of the left hand operand before comparison, an "R" means the opposite, an "X" means no implicit casting takes place, and a blank means that comparison between the values of the two data types is not supported.

Table 40. Implicit CASTs for comparison

	ukn	bln	int	real	dec	chr	tm	gtm	dt	ts	gts	ivl	blb	bit
ukn														
bln		X				L								
int			X	R	R	L								
real			L	X	L	L								
dec			L	R	X	L								
chr		R	R	R	R	X	R	R	R	R	R	R[1]	R	R
tm						L	X	L						
gtm						L	R	X						
dt						L			X	R[2]	R[2]			
ts						L	L[2]	X	L					
gts						L			L[2]	R	X			
ivl						L[1]						X		
blb						L							X	
bit						L								X

Notes:

1. When casting from a character string to an interval, the character string must be of the format "INTERVAL '<values>' <qualifier>". The format "<values>", which is allowable for an explicit CAST, is not allowable here because no qualifier external to the string is supplied.
2. When casting from a DATE to a TIMESTAMP or GMTTIMESTAMP, the time portion of the TIMESTAMP is set to all zero values - '00:00:00'. This is different to the behavior of the explicit CAST, which sets the time portion to the current time.

Numeric types: The comparison operators operate on all three numeric types.

Character strings: You cannot define an alternative collation order that, for example, collates upper and lowercase characters equally.

Note: When comparing character strings, trailing blanks are not significant so the comparison `'hello' = 'hello '` returns true.

Datetime values: Datetime values are compared in accordance with the natural rules of the Gregorian calendar and clock.

You can compare the time zone you are working in with the GMT time zone. The GMT time zone is converted into a local time zone based on the time zone difference between your local time zone and the GMT time specified.

When you compare your local time with the GMT time, the comparison is based on the difference at a given time on a given date.

Conversion is always based on the value of LOCALTIMEZONE. This is because GMTTimestamps are converted to local Timestamps only if it can be done unambiguously. Converting a local Timestamp to a GMTTimestamp has difficulties around the daylight saving cut-over time, and converting between times and GMT times (without date information) has to be done based on the LOCALTIMEZONE value, because you cannot specify which time zone difference to use otherwise.

Booleans: Boolean values can be compared using all or the normal comparison operators. The TRUE value is defined to be greater than the FALSE value. Comparing either value to the UNKNOWN boolean value (which is equivalent to NULL) returns an UNKNOWN result.

Intervals: Intervals are compared by converting the two interval values into intermediate representations, so that both intervals have the same interval qualifier. Year-month intervals can be compared only with other year-month intervals, and day-second intervals can be compared only with other day-second intervals.

For example, if an interval in minutes, such as `INTERVAL '120' MINUTE` is compared with an interval in days to seconds, such as `INTERVAL '0 02:01:00'`, the two intervals are first converted into values that have consistent interval qualifiers, which can then be compared. So, in this example, the first value could be converted into an interval in days to seconds, which will give `INTERVAL '0 02:00:00'` which can then be compared with the second value.

Character strings and other types: If a character string is compared to a value of another type, MQSeries Integrator attempts to cast the character string into a value of the same datatype as the other value.

For example, you could write an expression such as:

```
'1234' > 4567
```

The character string on the left would be converted into an integer before the comparison takes place. This behavior reduces some of the need for explicit CAST operators when comparing values derived from a generic XML message with literal values. (For details of explicit casts that are supported, see Table 39 on page 338.) It is this facility that allows you to write an expression such as:

```
Body.Trade.Quantity > 5000
```

In this example, the field reference on the left evaluates to the character string `'1000'` and, because this is being compared to an integer, that character string is converted into an integer before the comparison takes place.

Note that you must still check whether the price field that you want interpreted as a decimal is greater than a given threshold. You must make sure that the literal you compare it to is a decimal value and not an integer.

For example:

```
Body.Trade.Price > 100
```

would not have the desired effect, because the `Price` field would be converted into an integer, and that conversion would fail because the character string contains a decimal point. However, the following expression will succeed:

```
Body.Trade.Price > 100.00
```

Implicit CASTs for arithmetic operations

Normally the arithmetic operators (+, -, *, and /) operate on operands of the same data type, and return a value of the same data type as the operands. Cases where it is acceptable for the operands to be of different data types, or where the data type of the resulting value is different from the type of the operands, are in Table 41.

Table 41 (Page 1 of 2). Implicit CASTs for arithmetic operations

Left operand data type	Right operand data type	Supported operators	Result data type
INTEGER	REAL	+, -, *, /	REAL[1]
INTEGER	DECIMAL	+, -, *, /	DECIMAL[1]
INTEGER	INTERVAL	*	INTERVAL[4]
REAL	INTEGER	+, -, *, /	REAL[1]
REAL	DECIMAL	+, -, *, /	REAL[1]
REAL	INTERVAL	*	INTERVAL[4]
DECIMAL	INTEGER	+, -, *, /	DECIMAL[1]
DECIMAL	REAL	+, -, *, /	REAL[1]
DECIMAL	INTERVAL	*	INTERVAL[4]
TIME	TIME	-	INTERVAL[2]
TIME	GMTTIME	-	INTERVAL[2]
TIME	INTERVAL	+, -	TIME[3]
GMTTIME	TIME	-	INTERVAL[2]
GMTTIME	GMTTIME	-	INTERVAL[2]
GMTTIME	INTERVAL	+, -	GMTTIME[3]
DATE	DATE	-	INTERVAL[2]
DATE	INTERVAL	+, -	DATE[3]
TIMESTAMP	TIMESTAMP	-	INTERVAL[2]
TIMESTAMP	GMTTIMESTAMP	-	INTERVAL[2]
TIMESTAMP	INTERVAL	+, -	TIMESTAMP[3]
GMTTIMESTAMP	TIMESTAMP	-	INTERVAL[2]
GMTTIMESTAMP	GMTTIMESTAMP	-	INTERVAL[2]
GMTTIMESTAMP	INTERVAL	+, -	GMTTIMESTAMP[3]
INTERVAL	INTEGER	*, /	INTERVAL[4]
INTERVAL	REAL	*, /	INTERVAL[4]
INTERVAL	DECIMAL	*, /	INTERVAL[4]
INTERVAL	TIME	+	TIME[3]
INTERVAL	GMTTIME	+	GMTTIME[3]
INTERVAL	DATE	+	DATE[3]
INTERVAL	TIMESTAMP	+	TIMESTAMP[3]

Table 41 (Page 2 of 2). Implicit CASTs for arithmetic operations

Left operand data type	**Right operand data type**	**Supported operators**	**Result data type**
INTERVAL	GMTTIMESTAMP	+	GMTTIMESTAMP[3]

Notes:

1. The operand which does not match the data type of the result is cast to the data type of the result before the operation proceeds. For example, if the left operand to an addition operator is an INTEGER, and the right operand is a REAL, the left operand is cast to a REAL before the addition operation is performed.
2. Subtracting a (GMT)TIME value from a (GMT)TIME value, a DATE value from a DATE value, or a (GMT)TIMESTAMP value from a (GMT)TIMESTAMP value results in an INTERVAL value representing the time interval between the two operands.
3. Adding or subtracting an INTERVAL from a (GMT)TIME, DATE or (GMT)TIMESTAMP value results in a new value of the data type of the non-INTERVAL operand, representing the point in time represented by the original non-INTERVAL plus or minus the length of time represented by the INTERVAL.
4. Multiplying or dividing an INTERVAL by an INTEGER, REAL or DECIMAL value results in a new INTERVAL representing the length of time represented by the original multiplied or divided by the factor represented by the non-INTERVAL operand. For example, an INTERVAL value 2 hours 16 minutes multiplied by an REAL value of 2.5 results in a new INTERVAL value of 5 hours 40 minutes. The intermediate calculations involved in multiplying or dividing the original INTERVAL are carried out in the data type of the non-INTERVAL, but the individual fields of the INTERVAL (such as HOUR, YEAR, etc) are always integral, so some rounding errors may occur.

Implicit CASTs for assignment

Values can be assigned to one of three entities:

- A message field (or equivalent in an exception or destination list)

  Support for implicit conversion between the MQSeries Integrator data types and the message (in its bitstream form) is dependent on the appropriate parser. For example, the XML parser casts everything as character strings before inserting them into the output message bitstream.

- A field in a database table

  MQSeries Integrator converts each of its data types into a suitable standard SQL C data type, as detailed in Table 42. Conversion between this standard SQL C data type, and the data types supported by each DBMS, is dependent on the DBMS. Consult your DBMS documentation for more details.

Table 42. Conversions from MQSeries Integrator to SQL data types

MQSeries Integrator data type	SQL data type
NULL, or unknown or invalid value	SQL_NULL_DATA
BOOLEAN	SQL_C_BIT
INTEGER	SQL_C_LONG
REAL	SQL_C_DOUBLE
DECIMAL	SQL_C_CHAR[1]
CHARACTER	SQL_C_CHAR
TIME	SQL_C_TIME
GMTTIME	SQL_C_TIME
DATE	SQL_C_DATE
TIMESTAMP	SQL_C_TIMESTAMP
GMTTIMESTAMP	SQL_C_DATE
INTERVAL	not supported[2]
BLOB	SQL_C_BINARY
BIT	not supported[2]

Notes:

1. For convenience, DECIMAL values are passed to the DBMS in character form.
2. There is no suitable standard SQL C data type for INTERVAL or BITARRAY. These must be cast to another data type, such as CHARACTER, if it is necessary to assign them to a database field.

- A scalar variable. When assigning to a scalar variable, if the value being assigned is not of the same data type as the variable being assigned to, then an implicit cast is attempted with exactly the same restrictions and behavior as specified for the explicit CAST function, except when the data type of the variable is INTERVAL or DECIMAL.

  In both these cases, the value being assigned is first cast to an CHARACTER value, then an attempt is made to cast the CHARACTER value to an INTERVAL or DECIMAL. The reason for this is that INTERVAL requires a qualifier and DECIMAL requires a precision and scale: these must be specified in the explicit CAST, but must be obtained from the character string when implicitly casting. Therefore a further restriction is that when implicitly casting to an INTERVAL variable, the character string must be of the form "INTERVAL '<values>' <qualifier>" - the shortened "<values>" form that is acceptable for the explicit CAST is not acceptable here.

Data types of values from external sources

There are two external sources from which data can be extracted by ESQL:

- Message fields
- Database columns

The ESQL data type of message fields depends on the type of the message (XML, Neon, and so on), and the parser used to parse it. The ESQL data type of the value returned by a database column reference depends on the data type of the column in the database.

Table 43 shows which ESQL data types the various built-in DBMS data types (for DB2 Version 6.1, SQL Server Version 7.0, Sybase Version 12.0, and Oracle Version 8.1.5) are cast to when they are accessed by MQSeries Integrator.

Table 43 (Page 1 of 2). Implicit CASTS for database data types to MQSeries Integrator types

MQSeries Integrator	**DB2**	**SQL Server and Sybase**	**Oracle**
BOOLEAN		BIT	
INTEGER	SMALLINT INTEGER BIGINT	INT SMALLINT TINYINT	
REAL	REAL DOUBLE	FLOAT REAL	NUMBER()[1]
DECIMAL	DECIMAL	DECIMAL NUMERIC MONEY SMALLMONEY	NUMBER(P)[1] NUMBER(P,S)[1]
CHARACTER	CHAR VARCHAR CLOB	CHAR VARCHAR TEXT	CHAR NCHAR VARCHAR2 NVARCHAR2 ROWID UROWID LONG CLOB
TIME	TIME		
GMTTIME			
DATE	DATE		
TIMESTAMP	TIMESTAMP	DATETIME SMALLDATETIME	DATE
GMTTIMESTAMP			
INTERVAL			

Table 43 (Page 2 of 2). Implicit CASTS for database data types to MQSeries Integrator types

MQSeries Integrator	**DB2**	**SQL Server and Sybase**	**Oracle**
BLOB	BLOB	BINARY VARBINARY TIMESTAMP IMAGE UNIQUEIDENTIFIER	RAW LONG RAW BLOB
BIT			
not supported	DATALINK GRAPHIC VARGRAPHIC DBCLOB	NTEXT NCHAR NVARCHAR	NCLOB BFILE

Notes:

1. If an Oracle database column with NUMBER data type is defined with an explicit precision (P) and scale (S), then it is cast to an ESQL DECIMAL value; otherwise it is cast to a REAL.

   For example, an ESQL statement like this:

   ```
   SET OutputRoot.xxx
    = (SELECT T.department FROM Database.personnel AS T);
   ```

   where "Database.personnel" resolves to a TINYINT column in a SQL Server database table, results in a list of ESQL INTEGER values being assigned to OutputRoot.xxx.

   By contrast, an identical query where "Database.personnel" resolved to a NUMBER() column in an Oracle database results in a list of ESQL REAL values being assigned to OutputRoot.xxx.

Numeric expressions

Values from the three numeric datatypes can be combined using the normal arithmetic operators, which work in the same way as standard SQL operators. For example:

```
CAST(Body.Trade.Quantity AS INTEGER)
 * CAST(Body.Trade.Price AS DECIMAL) > 1000000
```

Remember that the fields in this sample generic are character strings because it is a generic XML message, so you need to cast the fields to values of the correct type. With all of the arithmetic operators there is the possibility that a run-time error can occur. For example, a run-time error occurs for the divide-by-zero situation, and for the arithmetic-overflow situation. Some other common numeric functions are provided, such as:

ABS	Returns the absolute value of a number.
CEIL	Returns the smallest number greater than or equal to the argument.
FLOOR	Returns the largest number less than or equal to the argument.
MOD	Returns a remainder after integer division.
SQRT	Returns the square root of its argument.

More details of all of these functions can be found in “Numeric functions” on page 354.

Datetime expressions

You can use arithmetic operators to perform various natural calculations on Datetime values. For example, you can calculate the difference between two dates as an interval, or you can add an interval to a timestamp.

Adding an interval to a Datetime value

The simplest operation you can perform is to add an interval to, or subtract an interval from, a Datetime value. For example, you could write the following expressions:

```
DATE '2000-01-29' + INTERVAL '1' MONTH
TIMESTAMP '1999-12-31 23:59:59' + INTERVAL '1' SECOND
```

Adding or subtracting two intervals

Two interval values can be combined using addition or subtraction. The two interval values must be of compatible types. For example, it is not valid to add a year-month interval to a day-second interval. So the following example is not valid:

```
INTERVAL '1-06' YEAR TO MONTH + INTERVAL '20' DAY
```

The interval qualifier of the resultant interval is sufficient to encompass all of the fields present in the two operand intervals. For example:

```
INTERVAL '2 01' DAY TO HOUR + INTERVAL '123:59' MINUTE TO SECOND
```

would result in an interval with qualifier DAY TO SECOND, because both day and second fields are present in at least one of the operand values.

Subtracting two Datetime values

Two Datetime values can be subtracted to return an Interval. In order to do this an interval qualifier must be given in the expression to indicate what precision the result should be returned in. For example:

```
(CURRENT_DATE - DATE '1776-07-04') DAY
```

would return the number of days since the 4th July 1776, whereas:

```
(CURRENT_TIME - TIME '00:00:00') MINUTE TO SECOND
```

would return the age of the day in minutes and seconds.

Scaling Intervals

An interval value can be multiplied by or divided by an integer factor:

```
INTERVAL '2:30' MINUTE TO SECOND / 4
```

Extracting fields from Datetimes and Intervals

You can extract individual fields from datetime values and intervals using the EXTRACT function. For example, you could extract the second field of the current time with the expression:

```
EXTRACT(SECOND FROM CURRENT_TIME)
```

You can use any of the keywords YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND in the EXTRACT function, but you can only extract a field that is present in the source value. Either a parse-time or a run-time error is generated if the requested field does not exist but this depends on how early the error can be detected. Other examples include:

```
EXTRACT(YEAR FROM CURRENT_DATE)
EXTRACT(HOUR FROM LOCAL_TIMEZONE)
```

Functions

Most of the function descriptions here impose restrictions on the datatypes of the arguments that can be passed to the function. If the values passed to the functions do not match the required datatypes, errors are generated at node configuration time if is possible to detect the errors at that point, otherwise run-time errors are generated when the function is evaluated.

String manipulation functions

The following functions perform manipulations on all strings (bit, byte, and character) with the exception of UPPER and LOWER, which operate only on character strings.

In these descriptions, the term 'singleton' is used to refer to a single part (bit, byte, or character) within a string of that type.

POSITION

The POSITION function returns an integer that gives the position of the first occurrence of one string (the `search_string`) in a second string (the `source_string`).

```
POSITION( search_string IN source_string )
```

If the value of either the search_string or the source_string is NULL, the result of the POSITION function is NULL. If the value of search_string has a length of zero, the result is one. If the substring cannot be found, it returns 0.

The behavior of POSITION function is the inverse of SUBSTRING function.

For example,:

```
POSITION('TQ_' IN Body.Trade.Company)
```

This second example returns the value 7:

```
POSITION('World' IN 'Hello World!')
```

LENGTH

```
LENGTH( source_string )
```

The LENGTH function return integer values which give the number of singletons in source_string. If the value of source_string is a NULL value, the result of the LENGTH function is the NULL value.

Examples:

`LENGTH('Hello World!')` returns 11.

`LENGTH('')` returns 0.

TRIM

The TRIM function is used to remove leading and trailing singletons from a string.

You can specify the TRIM function in any of the following formats:

```
TRIM( trim_specification trim_singleton FROM source_string )
TRIM( trim_specification FROM source_string )
TRIM( trim_singleton FROM source_string )
TRIM( source_string )
```

where `trim_specification` is one of LEADING, TRAILING, or BOTH. If `trim_specification` is not specified, BOTH is assumed. If `trim_singleton` is not specified, a default singleton is assumed. This default depends on the data type of `source_string`:

character	' ' (space)
byte	X'00'
bit	B'0'

TRIM returns a string value of the same data type and content as `source_string` but with any leading or trailing singletons that are equal to `trim_singleton` removed (depending on the value of `trim_specification`). If any of the parameters are the NULL value, the TRIM function returns a NULL value of the same data type as `source_string`.

The FROM keyword is not required, and is in fact prohibited if neither a trim specification, for example LEADING or TRAILING, nor a trim singleton, is specified.

If you have a field in a message that is padded at the end with an unknown number of `'x'` characters, and you want to compare the body of the character string to a literal value, you could use the following example:

```
TRIM(TRAILING 'x' FROM Body.Trade.Company) = 'Uncertain'
```

If you want to strip `'x'` characters from the beginning and end of the string, you could write:

```
TRIM('x' FROM Body.Trade.Company) = 'Uncertain'
```

By default, blanks are stripped from a character string, and you can therefore leave out the character altogether, as follows:

```
TRIM(LEADING FROM Body.Market.Sector) = 'Target'
```

To strip blanks from the beginning and end of a character string, you could write:

```
TRIM(Body.Market.Sector) = 'Target'
```

It is often unnecessary to strip trailing blanks from character strings before comparison because the rules of character string comparison mean that trailing blanks are not significant.

The following examples illustrate additional function: Examples:

```
TRIM(TRAILING 'b' FROM 'aaabBb') returns 'aaabB'
TRIM('  a  ') returns 'a'
TRIM(LEADING FROM '  a  ') returns 'a  '
TRIM('b' FROM 'bbbaaabbb') returns 'aaa'
```

LTRIM

```
LTRIM( source_string )
```

This function is equivalent to TRIM(LEADING ' ' FROM source_string).

RTRIM

```
RTRIM( source_string )
```

This function is equivalent to TRIM(TRAILING ' ' FROM source_string).

SUBSTRING

You can use the SUBSTRING function to extract a string of bits, bytes, characters from within another string of that type. You can use the result of SUBSTRING, for example, to compare to a known value.

The format of the function is as follows:

```
SUBSTRING( source_string FROM start_position )
SUBSTRING( source_string FROM start_position FOR string_length )
```

If any of the parameters to the SUBSTRING function are NULL, the result is the NULL string (which is different from the empty string).

The following example:

```
SUBSTRING(Body.Trade.Company FROM 1 FOR 3) = 'TQ_'
```

compares the first three singletons of a string to a given value. The positions in the string start at 1, so the `FROM 1` clause indicates that the substring should start at the first singleton. The `FOR 3` clause indicates that three singletons are included in the substring. This has a similar result to using the LIKE predicate.

This second example returns the string 'World!':

```
SUBSTRING('Hello World!' FROM 7)
```

The SUBSTRING function is implemented using the following algorithm (the purpose of the algorithm is to define how the parameters are normalized to ensure that the start position and the end position both lie inside the source string. The behavior is otherwise as expected.):

- Let C be the value of source_string. Let LC be the length of C and let S be the value of start_position.
- If string_length is specified, let L be the value of string_length and let E be S+L. Otherwise let E be the larger of LC+1 and S.
- If E is less than S, the function returns a NULL value.
- If S is greater than LC, or if E is less than 1, the result of the SUBSTRING function is a zero length string.
- Otherwise Let S1 be the larger of S and 1. Let E1 be the smaller of E and LC+1. Let L1 be E1-S1.
- The result of the SUBSTRING function is a string containing the L1 singletons of C starting at number S1 in the same order that the singletons appear in C.

UPPER, UCASE

```
UPPER( source_string )
UCASE( source_string )
```

The UPPER and UCASE functions both return a new character string which is the same length as the source character string and which is identical to the input string, except is has all lowercase letters replaced with the corresponding uppercase letters. If the source string is NULL, the return value is NULL.

LOWER, LCASE

```
LOWER( source_string )
LCASE( source_string )
```

The LOWER and LCASE functions both return a new character string which is the same length as the source string and which is identical to the input string, except that it has all uppercase letters replaced with the corresponding lowercase letters. If the source string is NULL, the return value is NULL.

OVERLAY

```
OVERLAY( source_string PLACING source_string2 FROM start_position )
OVERLAY( source_string PLACING source_string2 FROM start_position FOR string_length)
```

If any of the parameters is NULL, the result is a NULL value of the same datatype as source_string. If string_length is not specified, string_length is equal to LENGTH(source_string2).

The result of the OVERLAY function is equivalent to:

```
SUBSTRING(source_string FROM 1 FOR start_position -1 ) || source_string2 ||
  SUBSTRING(source_string FROM start_position + LENGTH(source_string2))
```

(where || is the concatenation operator).

Numeric functions

The following functions manipulate numeric strings.

ABS

```
ABS( expression )
ABSVAL( expression )
```

The argument must be a numeric value. The function returns the absolute value of the argument. If the type of the value that the function returns is the same as the type of the argument that the function is called with. The argument can be NULL. If the argument is NULL, the function returns a NULL value.

BITAND

```
BITAND(expression, expression, ...)
```

The BITAND function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise and of the binary representation of the numbers.

BITNOT

```
BITNOT(expression)
```

The BITNOT function takes one parameter which must result in an integer value and results the result of performing the bitwise complement of the binary representation of the number.

BITOR

```
BITOR(expression, expression, ...)
```

The BITOR function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise or of the binary representation of the numbers.

BITXOR

```
BITXOR(expression, expression, ...)
```

The BITXOR function takes two or more parameters that must result in integer values, and returns the result of performing the bitwise XOR of the binary representation of the numbers.

CEIL

```
CEIL( expression )
CEILING( expression )
```

Returns the smallest integer value greater than or equal to the argument. The argument can be any numeric type. If the argument is an integer, the function returns the argument value. The value returned by the function is of the same type as the argument value. If the argument is null, the result is the null value.

FLOOR

```
FLOOR( expression )
FLOOR( expression )
```

Returns the largest integer value less than or equal to the argument. The argument can be any numeric type. If the argument is an integer, the function returns the argument value. The value returned by the function is of the same type as the argument value. If the argument is null, the result is the null value.

MOD

```
MOD(expression, expression)
```

Returns the remainder of the first argument divided by the second argument. The result is negative only if first argument is negative. The arguments must have integer datatypes. The function returns an integer. If any argument is null, the result is the null value.

ROUND

```
ROUND(expression1, expression2)
```

Returns the expression1 rounded to expression2 placed right of the decimal point. If expression2 is negative, expression1 is rounded to the absolute value of expression2 placed to the left of the decimal point. The first argument can be of any built-in numeric data type. The second argument can be integer, decimal or floating point. A decimal argument is converted to double-precision floating-point number for processing by the function. The result of the function is integer if the first argument is integer and double if the first argument is double or decimal. If any argument is null, the result is the null value.

SQRT

```
SQRT(expression)
```

Returns the square root of the argument. The argument can be any built-in numeric data type. It has to be converted to double-precision floating-point number for processing by the function. The result of the function is double-precision floating-point number. If the argument is null, the result is the null value.

TRUNCATE

```
TRUNCATE(expression1, expression2)
```

Returns argument1 truncated to argument2 placed right of decimal point. If argument2 is negative, argument1 is truncated to the absolute value of argument2 placed to the left of the decimal point. The first argument can be any built-in numeric data type. The second argument has to be an integer. Decimal values are converted to double-precision floating-point numbers for processing by the function.

The result of the function is an integer is the first argument is an integer, and a double if the first argument is a double or a decimal. If any argument is null, the result is the null value.

Datetime functions

The following functions allow you to manipulate fields according to date and time values.

EXTRACT

```
EXTRACT(extract_field FROM source_field)
```

where extract_field is one of YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and source_field must be a, expression which results in a date time or interval value. If extract_field is YEAR, MONTH, DAY, HOUR or MINUTE, the result is an integer value. If extract_field is SECOND, the result is a floating point value. If source_field is NULL, the result is NULL.

CURRENT_DATE

The CURRENT_DATE function returns a date value representing the current date in local time. That is, it is equivalent to CAST(CURRENT_TIMESTAMP AS DATE).

```
CURRENT_DATE
```

The CURRENT_DATE function is not a true function in that no parentheses are necessary. All calls to CURRENT_DATE within the processing of one node are guaranteed to return the same value.

CURRENT_TIME

The CURRENT_TIME function returns a non-GMT time value representing the current local time. That is, it is equivalent to CAST(CURRENT_TIMESTAMP AS TIME).

```
CURRENT_TIME
```

The CURRENT_TIME function is not a true function in that no parentheses are necessary. All calls to CURRENT_TIME within the processing of one node are guaranteed to return the same value.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns a non-GMT timestamp value representing the current local time.

```
CURRENT_TIMESTAMP
```

The CURRENT_TIMESTAMP function is not a true function in that no parentheses are necessary. All calls to CURRENT_TIMESTAMP within the processing of one node are guaranteed to return the same value.

CURRENT_GMTDATE

The CURRENT_GMTDATE function returns a date value representing the current date in the GMT time zone. It is equivalent to CAST(CURRENT_GMTTIMESTAMP AS DATE).

```
CURRENT_GMTDATE
```

The CURRENT_GMTDATE function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTDATE within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIME

The CURRENT_GMTTIME function returns a GMT time value representing the current time in the GMT time zone. It is equivalent to CAST(CURRENT_GMTTIMESTAMP AS TIME).

```
CURRENT_GMTTIME
```

The CURRENT_GMTTIME function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTTIME within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIMESTAMP

The CURRENT_GMTTIMESTAMP function returns a GMT timestamp value representing the current time in the GMT time zone.

```
CURRENT_GMTTIMESTAMP
```

The CURRENT_GMTTIMESTAMP function is not a true function in that no parentheses are necessary. All calls to CURRENT_GMTTIMESTAMP within the processing of one node are guaranteed to return the same value.

LOCAL_TIMEZONE

The LOCAL_TIMEZONE function returns an interval value which represents the local time zone displacement from GMT.

```
LOCAL_TIMEZONE
```

The LOCAL_TIMEZONE function is not a true function in that no parentheses are necessary. The value returned is an interval in hours and minutes representing the displacement of the current time zone from Greenwich Mean Time. The sign of the interval is such that a local time could be converted to a time in GMT by subtracting the result of the LOCAL_TIMEZONE function.

Miscellaneous functions

| You can also use the CARDINALITY, FIELDNAME, FIELDTYPE, BITSREAM, COALESCE, and NULLIF functions, as described below.

CARDINALITY

```
CARDINALITY(array)
```

Returns the number of elements in the argument array.

FIELDNAME

```
FIELDNAME(path)
```

Returns the name of the field that the argument path identifies as a string. If the path identifies a nonexistent entity, NULL is returned.

FIELDTYPE

```
FIELDTYPE(path)
```

Returns the type of the field that the argument path identifies as an integer. If the path identifies a nonexistent entity, NULL is returned. The result of this function will typically be compared with a symbolic constant defined by a parser which represents a type value. Note that this is not the datatype of the field that the path identifies.

BITSTREAM

| The BITSTREAM function returns a BLOB representing the actual bit stream of the portion of the message specified.

```
BITSTREAM(path)
```

This function is typically used in message warehouse scenarios, where the bit stream of a message needs to be stored in a database. The function returns the bit stream of the physical portion of the message which the syntax element identified by the path lies in. It does not return the bit stream representing the actual syntax element identified. So for example the following two calls would return the same value:

```
BITSTREAM(Root.MQMD)
BITSTREAM(Root.MQMD.UserIdentifier);
```

COALESCE

COALESCE returns the first argument that is not null. The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result is null only if all the arguments are null. The arguments must be compatible. The COALESCE function can be used to provide a default value for the value of a field which might not exist in a message. For example, the expression:

```
COALESCE(Body.Salary, 0)
```

would return the value of the Salary field in the message if it existed, or 0 (zero) if that field did not exist.

NULLIF

The NULLIF function returns a null value if the arguments are equal; otherwise, it returns the value of the first argument. The arguments must be comparable. The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Reserved keywords

ALL	ESCAPE	MINUTE
AND	EXISTS	MONTH
ANY	FALSE	NULL
AS	FOR	NOT
ASSYMETRIC	FLOAT	OR
BETWEEN	FROM	ORDER
BIT	GMTTIME	PLACING
BLOB	GMTTIMESTAMP	REPEAT
BOOLEAN	GROUP	ROW
BOTH	HAVING	SECOND
BY	HOUR	SELECT
CASE	IF	SET
CHAR	IN	SOME
CHARACTER	INSERT	SUM
COUNT	INT	SYMMETRIC
CURRENT_DATE	INTEGER	THEN
CURRENT_TIME	INTERVAL	TIME
CURRENT_TIMESTAMP	INTO	TIMESTAMP
CURRENT_GMTDATE	IS	TO
CURRENT_GMTTIME	ITEM	THE
CURRENT_GMTTIMESTAMP	ITERATE	TRAILING
DATE	LAST	TRUE
DAY	LEADING	UNKNOWN
DECLARE	LEAVE	UNTIL
DELETE	LIKE	UPDATE
DO	LIST	VALUES
DISTINCT	LOCAL_TIMEZONE	WHEN
ELSE	LOOP	WHERE
ELSEIF	MAX	WHILE
END	MIN	YEAR

Initial correlation names

For an expression in a Filter node, or for a statement in a Database node, the following correlation names are defined by default:

Root
Identifies the root of the message passing though the Filter node.

Body
Identifies the last child of the root of the message, that is the "body" of the message. This is just an alias for "Root.*[LAST]" (the interpretation of this path is explained below).

Properties
Identifies the standard properties of the input message.

DestinationList
Identifies the structure which contains the destination list for the message passing through the node.

ExceptionList
Identifies the structure which contains the current exception list that the node has access to.

For a Compute node, the initial correlation names are different because there are two messages involved, the input message and the output message. The initial correlation names for a compute name are as follows:

InputRoot
Identifies the root of the input message

InputBody
Identifies the "body" of the input message. Like "Body" in a Filter node this is just an alias for "InputRoot.*[LAST]"

InputProperties
Identifies the standard properties of the input message.

InputDestinationList
Identifies the structure which contains the destination list for the input message.

InputExceptionList
Identifies the structure which contains the destination list for the message passing through the node.

OutputRoot
Identifies the root of the output message.

OutputDestinationList
Identifies the structure which contains the destination list for the output message. For a description of the format of a destination list, see “Exception and destination list structure” on page 373.

Note that whilst this correlation name is always valid, it only has meaning when the "Compute Mode" property of the Compute node indicates that the Compute node is calculating the destination list.

OutputExceptionList
Identifies the structure which contains the destination list which the Compute node is generating.

Note that whilst this correlation name is always valid, it only has meaning when the "Compute Mode" property of the Compute node indicates that the Compute node is calculating the exception list. ("Compute node" on page 108 includes details of the Compute node properties.)

Note that in a Compute node there is no correlation name "OutputBody". New correlation names may be introduced by SELECT expressions (see "Arbitrary repeats: the SELECT expression" on page 314), quantified predicates, and FOR statements.

Case sensitivity of ESQL syntax

The following text describes instances in which the case in which ESQL statements are specified is significant.

- The keyword Database is case sensitive.
- References to the following correlation names are case sensitive:

```
InputRoot
InputBody
InputProperties
InputDestinationList
InputExceptionList
OutputRoot
OutputDestinationList
OutputExceptionList
Properties
```

- References to elements in a path are frequently case sensitive. This depends on the parser. All parsers supplied by MQSeries Integrator are case sensitive.

```
InputRoot.Properties.MessageSet
InputRoot.Properties.MessageType
InputRoot.Properties.MessageFormat
InputRoot.Properties.Encoding
InputRoot.Properties.CodedCharSetId
InputRoot.Properties.Transactional
InputRoot.Properties.Persistence
InputRoot.Properties.CreationTime
InputRoot.Properties.ExpirationTime
InputRoot.Properties.Priority
InputRoot.Properties.Topic
```

Expression parsing and evaluation

The expression or statements used to program a Filter, Compute, or Database node are parsed when it is set into the node by a configuration message. Therefore some syntax error messages may be produced at this time. The parsing and understanding of an expression is done without reference to any kind of message format information. This means that the meaning of the expression is the same for each message that passes through the Filter node, regardless of the message format of the message. This is quite different from the way that standard database SQL expressions are understood. In database systems, a standard SQL expression is interpreted with reference to the schemas of any relevant tables, so ambiguities can be detected and possibly resolved. No such rules are possible in the language described here however.

Note that the reason that it must be possible to understand a filter expression without looking at a message format is that it must be possible to process messages which do not have message formats, such as generic XML. This affects how fields in messages are referred to. This also means that some types of errors such as type mismatch errors cannot always be caught when the expression is parsed at configuration time, but can only be caught when the expression is evaluated against a message, because whether the errors are produced or not can depend on the data in the message.

Expression type checking

It is not always possible to tell the exact datatype that an expression will result in. This is because expressions are "compiled" without reference to any kind of message schema, and so it is not possible to tell what the result of evaluating a field reference will be. Therefore not all type errors can be caught at "compile" time. As many as possible will be caught at compile time, but some may only be caught at run time when the expression is actually evaluated against a message. In these cases, a run-time error is generated.

Examples for generic XML messages

This section provides further details about how you can use ESQL with generic XML messages. The examples here use a default message. Each example illustrates the entire message being copied, with additional ESQL to address, add to, or modify specific parts of the message. You can also create a new message using these examples as a basis: although this is not illustrated, the principle is the same.

| **Note:** The message domains JMSMap and JMSStream are equivalent to the XML domain, and the XML constants used to qualify references also apply. These JMS message domains are not referenced specifically in this section, but the examples are applicable, with the appropriate substitution for the root parser name. For example, you can use `Root.JMSMap.(XML.attr)abc` and similar references.

XML declaration

The XML declaration takes the following form:

```
<?xml version="1.0"?>
```

XmlDecl

You can modify the XML declaration using the following:

```
SET OutputRoot.XML.(XML.XmlDecl)='';
```

Version: The default behavior of the parser causes the version to be automatically included on a call to construct a declaration, and version is set to '1.0'. However, the version constant has been retained to allow for later development of the XML specification.

```
SET OutputRoot.XML.(XML.XmlDecl).(XML.Version)='1.0';
```

Standalone: The standalone option indicates whether or not an external DTD is associated with the XML file.

```
SET OutputRoot.XML.(XML.XmlDecl).(XML.Standalone)='no';
```

Encoding: By default XML supports UTF-8 or UTF-16. It is therefore necessary to inform the parser of any other encoding that the document has been written in.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<A><B C="TODAY IS FRIDAY"><D E="42"/></B></A>
```

To create the above from a message that does not include a declaration, you must code the following ESQL expressions:

```
SET OutputRoot=InputRoot ;
SET OutputRoot.XML.(XML.XmlDecl)='';
SET OutputRoot.XML.(XML.XmlDecl).(XML.Version)='1.0';
SET OutputRoot.XML.(XML.XmlDecl).(XML.Encoding)='ISO-8859-1' ;
SET OutputRoot.XML.(XML.XmlDecl).(XML.Standalone)='no';
```

Document Type Declaration

Document type declarations are optionally found following the XML declaration. The following sample illustrates the basic building blocks of the DTD (element, entity attribute, and notation types).

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
<!ELEMENT doc (header,body)>
<!ELEMENT header (doctitle,byline,pubdate,cpyrt,notes)>
<!ELEMENT doctitle (#PCDATA)>
<!ELEMENT byline (#PCDATA)>
<!ELEMENT pubdate (#PCDATA)>
<!ELEMENT cpyrt (#PCDATA)>
<!ELEMENT notes (#PCDATA)>
<!ELEMENT body (chapter)>
<!ELEMENT chapter (ctitle,formula)>
<!ELEMENT ctitle (#PCDATA)>
<!ELEMENT formula (#PCDATA)>
<!NOTATION TeX PUBLIC "+//ISBN 0-201-13448-9::Knuth//NOTATION The TeXbook//EN">
<!ENTITY XML "eXtensible Markup Language">
<!ATTLIST formula format NOTATION (tex|troff) #REQUIRED>
]>
<doc>
<header>
<doctitle>Sample DTD document</doctitle>
<byline>S P Jones</byline>
<pubdate>Feb 15 2000</pubdate>
<cpyrt>today's sample of XML</cpyrt>
<notes>including a notation of an entity</notes>
</header>
<body>
<chapter>
<ctitle>formula 1</ctitle>
<formula format="tex">$\frac{\sqrt{x+y}} {\pi}$</formula>
</chapter>
</body>
</doc>
```

The following examples illustrate how you can manipulate these message contents.

DocTypeDecl

The document type declaration is a construct that contains all the other declarations that make up the components of an XML message. It can optionally have an external subset, an internal subset, or both.

```
SET OutputRoot.XML.(XML.DocTypeDecl)Note='';
```

IntSubset: The internal subset is a constant that has been added to allow the addressing of that part of the XML-DTD document that has been stored internally. In the sample above, the internal subset includes everything between the [and the]>.

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset)='';
```

The SystemId and PublicId are used to identify the location of the external subset (in a separate DTD file, if specified) that the parser must use in conjunction with the internal subset defined here (if specified). Both internal and external subsets are optional.

SystemId: The SystemId is system specific and normally includes a filename and location. For example:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.SystemId)='note.dtd';
```

PublicId: The PublicId is a non system-specific external entity identifier. For example:

```
SET OutputRoot.XML.(XML.DocTypeDecl).
  (XML.IntSubset).(XML.NotationDecl).(XML.PublicId)
 ='+//ISBN 0-201-13448-9::Knuth//NOTATION The TeXbook//EN';
```

NotationDecl

A notation declaration allows XML documents to access external information. It usually identifies non-XML information.

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).(XML.NotationDecl)TeX='';
```

To construct the following Notation declaration within a message that contains no current example of an XML DTD, use:

```
SET OutputRoot=InputRoot ;
SET OutputRoot.XML.(XML.DocTypeDecl)Note='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.SystemId)='note.dtd';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset)='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.NotationDecl)TeX='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.NotationDecl).(XML.PublicId)
 ='+//ISBN 0-201-13448-9::Knuth//NOTATION The TeXbook//EN';
```

This produces the following output:

```
<?xml version="1.0"?>
<!DOCTYPE Note SYSTEM "note.dtd"[<!NOTATION TeX PUBLIC "
  +//ISBN 0-201-13448-9::Knuth//NOTATION The TeXbook//EN">]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

Entities

Samples are provided for each of the five supported entity element types.

ParameterEntityDecl: The following ESQL illustrates a reference to the ParameterEntityDecl:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset)='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ParameterEntityDecl)test='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ParameterEntityDecl)test.(XML.EntityDeclValue)
 ='#PCDATA|emphasis|link';
```

The ESQL shown generates the following output in the test sample:

```
<?xml version="1.0"?>
<!DOCTYPE Note SYSTEM "note.dtd"[
 <!ENTITY % test "#PCDATA|emphasis|link">
 ]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

ExternalParameterEntityDecl: The following ESQL illustrates a reference to the ExternalParameterEntityDecl:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ExternalParameterEntityDecl)bookdef='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ExternalParameterEntityDecl)bookdef.(XML.SystemId)
 ='bookdef.dtd';
```

The ESQL shown generates the following output in the test sample:

```
<?xml version="1.0"?>
<!DOCTYPE Note SYSTEM "note.dtd"[
 <!ENTITY % bookdef SYSTEM "bookdef.dtd">
 ]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

EntityDecl: The following ESQL illustrates a reference to the EntityDecl:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.EntityDecl)XML='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.EntityDecl)XML.(XML.EntityDeclValue)='eXtensible Markup Language';
```

This creates a general entity within the internal subset of the DTD. Using the standard sample, this gives:

```
<?xml version="1.0"?>
<!DOCTYPE Note SYSTEM "note.dtd"[
 <!ENTITY XML "eXtensible Markup Language">
 ]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

ExternalEntityDecl: The following ESQL illustrates a reference to the ExternalEntityDecl:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ExternalEntityDecl)test='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ExternalEntityDecl)test.(XML.SystemId)='test.txt';
```

The ESQL shown generates the following output in the test sample:

```
<?xml version="1.0"?><!DOCTYPE Note SYSTEM "note.dtd"[
 <!ENTITY test SYSTEM "test.txt">
 ]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

UnparsedEntityDecl: The following ESQL illustrates a reference to the UnparsedEntityDecl:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.UnparsedEntityDecl)pic='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.UnparsedEntityDecl)pic.(XML.SystemId)='scheme.gif';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.UnparsedEntityDecl)pic.(XML.NotationReference)='gif';
```

The ESQL shown generates the following output in the test sample:

```
<?xml version="1.0"?>
<!DOCTYPE Note SYSTEM "note.dtd"[
 <!ENTITY pic SYSTEM "scheme.gif" NDATA gif>
 ]>
<A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

Subcomponents used in Entities

These are constants that exist to help define the various entities completely.

NotationReference: The notation reference is used to add the NDATA section to an UnparsedEntityDecl.

The following example illustrates the addition of the NDATA gif.

```
<!ENTITY pic SYSTEM "scheme.gif" NDATA gif>
```

This form is created using the following ESQL:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.UnparsedEntityDecl)pic.(XML.NotationReference)='gif';
```

EntityDeclValue: You can use EntityDeclValue to set a value element that is a child of any of the five EntityDecl types.

For example:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.EntityDecl)XML.(XML.EntityDeclValue)='eXtensible Markup Language';
```

SystemID and PublicID: SystemID and PublicID can be used as children of ExternalEntityDecl, ExternalParameterEntityDecl, or UnparsedEntityDecl.

ElementDef

ElementDef represents the <!ELEMENT construct. The following ESQL example illustrates the use of ElementDef.

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset)='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.ElementDef)warning='(para+)';
```

The ESQL shown generates the following statement within the internal DTD subset.

```
<!ELEMENT warning (para+)>
```

Attribute definitions

The following examples illustrate the ESQL required to set up the <!ATTLIST construct and its contents.

AttributeList: AttributeList represents the <!ATTLIST construct. The contents of the construct are defined by the following elements:

- AttributeDef

  This name element defines an attribute in the attribute list.

- AttributeDefValue

  This value element gives the default value of the attribute.

- AttributeDefDefaultType

  This value element indicates the default type for the attribute.

- AttributeDefType

  This is a name-value element where the name corresponds to the attribute type, and is one of:

  - CDATA
  - ID
  - IDREF
  - IDREFS
  - ENTITY
  - ENTITIES
  - NMTOKEN
  - NMTOKENS
  - NOTATION

  If the attribute has an enumeration, the value of the name-value element contains the enumerated list.

An attribute list might be coded in XML as follows:

```
<!ATTLIST formula format NOTATION (tex|troff) #REQUIRED>
```

You must code the following ESQL to create this element:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)formula='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)formula.(XML.AttributeDef)format='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)formula.(XML.AttributeDef)format.
 (XML.AttributeDefDefaultType)='REQUIRED';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)formula.(XML.AttributeDef)format.
 (XML.AttributeDefType)NOTATION='(tex|troff)';
```

A second example of an attribute list is:

```
<!ATTLIST report security (public|confidential|secret)"public">
```

This can be constructed using the following ESQL:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)report='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)report.(XML.AttributeDef)security='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)report.(XML.AttributeDef)security.
 (XML.AttributeDefType)='(public|confidential|secret)';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
 (XML.AttributeList)report.(XML.AttributeDef)security.
 (XML.AttributeDefValue)='public';
```

The XML message body

This sections discusses the following components of the XML message body:

- tag
- attr
- pcdata
- ProcessingInstruction
- CDataSection

The examples here use the following XML message structure:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<A>
<B C="TODAY IS FRIDAY">
<D E="42">1234</D>
<D>test</D>
</B>
</A>
```

XML.tag

XML.tag can be used to identify a specific occurrence of a tag within the message body.

If you want to address the second <D> tag in the example message, in most cases you can specify the simplest form:

```
InputRoot.XML.A.B.D[2]
```

You can also specify:

```
InputRoot.XML.A.B.(XML.tag)D[2]
```

If the message is fully validated (that is, it is fully defined in the DTD and the root XML message element matches the DTD name), you must use the following form:

```
InputRoot.XML.(XML.tag)A.B.D[2]
```

Consider a second example:

```
<a><b c="1234"><d d="1234"><d>1234</d><d>2345</d></b></a>
```

To address the first <d> tag, the following forms are valid:

```
InputRoot.XML.a.b.d.d[2]

InputRoot.XML.a.b.d.(XML.tag)d

InputRoot.XML.a.b.d.(XML.tag)d[1]
```

XML.attr

XML.attr is used to reference an attribute of a tag within the message.

To change the attribute with name E on the first D tag you must specify:

```
SET OutputRoot.XML.A.B.D.(XML.attr)E='57';
```

This results in:

```
...<B C="TODAY IS FRIDAY"><D E=57">1234</D><D>test</D></B>...
```

XML.pcdata

You can use XML.pcdata to reference the data associated with a tag within the body.

To change the pcdata field of the second <D> tag (currently set to `test`) you can specify:

```
SET OutputRoot.XML.A.B.D[2](XML.pcdata)='result';
```

The same result can be achieved with the following:

```
SET OutputRoot.XML.A.B.D[2]='result';
```

Both assignments give the following output:

```
...<B C="TODAY IS FRIDAY"><D E="42">1234</D><D>result</D></B>...
```

To change the value of the first D tag (1234), you can specify:

```
SET OutputRoot.XML.A.B.D[1].(XML.pcdata)='5555';
```

This produces:

```
...<B C="TODAY IS FRIDAY"><D E="42">5555</D><D>result</D></B>...
```

ProcessingInstruction and DocTypePI

An XML processing instruction contains information required by a specific application expected to process XML data.

```
SET OutputRoot.XML.A.B.D.(XML.ProcessingInstruction)test='Do this';
```

This statement produces the following output:

```
<A><B C="TODAY IS FRIDAY"><D E="42"><?test Do this?></D></B></A>
```

WhiteSpace and DocTypeWhiteSpace: This adds a series of blanks into the message at the desired location.

```
SET OutputRoot.XML.A.B.(XML.WhiteSpace)='     ';
```

This produces the output:

```
<?xml version="1.0"?><!DOCTYPE Note SYSTEM "note.dtd">
<A><B C="TODAY IS FRIDAY"><D E="42">test</D>     </B></A>
```

Comment and DocTypeComment: These elements refer to comments within the message.

```
SET OutputRoot=InputRoot ;
SET OutputRoot.XML.A.B.D.(XML.Comment)='This is a comment';
```

This changes

```
<A><B C="TODAY IS FRIDAY"><D E="42"></D></B></A>
```

into

```
<A><B C="TODAY IS FRIDAY"><D E="42"><!--This is a comment--></D></B></A>
```

Similarly:

```
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
   (XML.DocTypeComment)='comment';
```

produces the following output

```
<?xml version="1.0"?><!DOCTYPE Note SYSTEM "note.dtd"
[<!--comment-->]><A><B C="TODAY IS FRIDAY"><D E="42">test</D></B></A>
```

The declarations and the internal subset have been added by this example.

CDataSection

A CDATA section is an area of message text that is to remain unparsed. You can therefore include characters that are normally excluded from markup within the CDATA section.

```
SET OutputRoot.XML.A.B.D.(XML.CDataSection)='1122334455';
```

This produces the output:

```
<A><B C="TODAY IS FRIDAY"><D E="42"><![CDATA[1122334455]]></D></B></A>
```

Exception and destination list structure

A tree representation is used within a broker to represent the data contained in the bit-stream representation of a message used outside the broker. Within the broker this tree representation is supplemented by two additional trees:

- The destination list tree

  This represents the destinations to which a message is sent.
- The exception list tree

  This represents the exception conditions that have occurred while processing that message.

A message being processed within the broker consists of three separate syntax element trees:

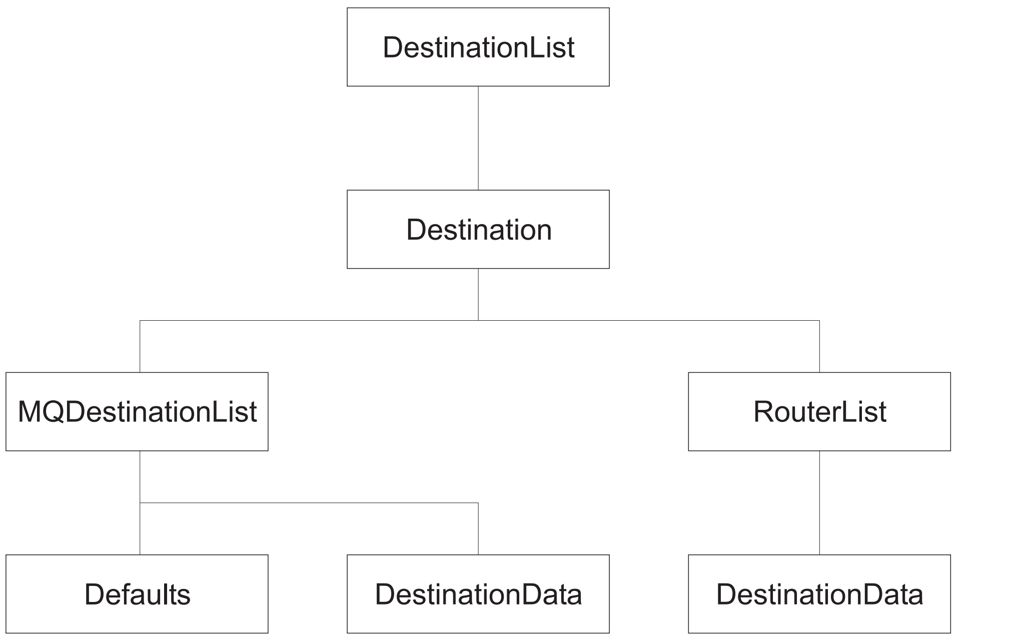
- The destination list tree
- The exception list tree
- The message tree

You can query and manipulate each of these trees in much the same way in Filter, Database, and Compute nodes. Elements can be created, examined, or even copied from one tree to another. Note that the destination and exception list trees only exist within the broker and are not replicated in the bit stream. The following sections describe the structure of the destination and exception lists.

Destination lists

| A destination list tree describes a list of internal and external destinations to which
| a message will be sent. Output nodes can be configured to examine this list and
| send the message to the given destinations. Alternatively, they can be configured to send messages to a fixed destination. In this case, the destination list has no effect on broker operations and can be empty (that is, consist of a Destination List element only).

The destination list tree has a definite structure that is illustrated in the following figure:



The root of the tree is called “Destination List”. The tree consists of a single name element called “Destination”: this is the first and only child of the Destination List. The Destination element consists of a number of children that indicate the transport types to which the message will be directed (the Transport identifiers). Each element is a single name element, for example, MQSeries or RouterList.

The transport name element might contain an element called “Defaults”. If it does, this must be in the first child and contains a set of name-value elements that give default values for the message destination and its put options.

The element that identifies the transport might also contain a number of elements called “DestinationData”. Each of these contains a set of name-value elements that defines a message destination and its put options.

- For MQSeries, the set of elements that define destination comprises:

  queueManagerName
  queueName
  transactionMode
  persistenceMode
  newMsgId
  newCorrelId
  segmentationAllowed
  alternateUserAuthority

  All of these elements have a data type of string. See the description of the MQOutput node in the on-line help for their descriptions and valid values. You can access the online help from the Help menu in the Control Center taskbar or by highlighting an MQOutput node, right clicking, and selecting Help.

- For RouterList, the set of elements that define destination has a single entry:

  labelName

Exception lists

If no exception conditions occur while you are processing a message, the exception list associated with that message consists of a root element only. This is, in effect, an empty list of exceptions.

If an exception condition occurs, message processing is suspended and an exception is thrown. Control is passed back to a higher level, that is, an enclosing catch block. An exception list is built to describe the failure condition, and then the whole message, together with the destination list and the newly-populated exception list, is propagated through an exception handling message flow path.

Exception handling paths start at a failure terminal (most message processing nodes have these), the catch terminal of an MQInput node, or the catch terminal of a TryCatch node, but are no different in principle from a normal message flow path. Such a flow consists of a set of interconnected message flow nodes defined by the designer of message flow. The exception handling paths differ in detail. For example, they might examine the exception list to determine the nature of the error, and so be able to make an appropriate response.

The message and destination list that are propagated to the exception handling message flow path are those in effect at the start of the exception path, not necessarily those in effect when the exception is thrown. Figure 82 illustrates this point:

- A message (M1) and destination list (D1) are being processed by a message flow. They are passed through the TryCatch node to Compute1.
- Compute1 updates the message and destination list and propagates a new message (M2) and destination list (D2) to the next node, Compute2.
- An exception is thrown in Compute2. The exception is propagated back to the TryCatch node, but the message and destination list are not. Therefore the exception handling path starting at point **A** has access to the first message and destination list, M1 and D1.
- If there had been no TryCatch node in the message flow, and the failure terminal of Compute2 had been connected (point **B**), the message and destination list M2 and D2 would have been propagated to the node connected to that failure terminal.

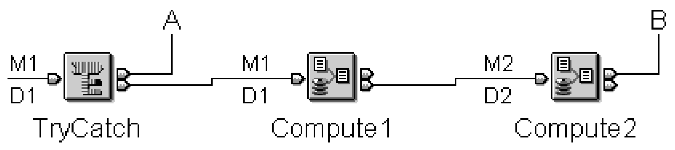


Figure 82. Message and destination list for an exception

The exception list tree has a definite structure. The root of the tree is called “ExceptionList”, and the tree itself consists of a set of one or more exception descriptions. Each exception description consists of a name element whose name is one of the following:

- RecoverableException
- ParserException
- ConversionException
- UserException

These name elements contain children that take the form of a number of name-value elements that give details of the exception and zero or more name elements whose name is “Insert”. The NLS (National Language Support) message number identified in a name-value element in turn identifies an MQSeries Integrator error message. All error messages are defined in detail in the *MQSeries Integrator Messages* book. The Insert values are used to replace the variables within this message, and provide further detail of the precise cause of the exception.

The name-value elements within the exception list are shown in Table 44 on page 377.

Table 44. Exception list name-value elements

Name		Type	Description
File[1]		String	C++ source file name
Line[1]		Integer	C++ source file line number
Function[1]		String	C++ source function name
Type[2]		String	Source object type
Name[2]		String	Source object name
Label[2]		String	Source object label
Text[1]		String	Additional text
Catalog[3]		String	NLS message catalog name[4]
Severity[3]		Integer	1=information 2=warning 3=error
Number[3]		Integer	NLS message number[4]
Insert[3]	Type	Integer	The data type of the value: 0=Unknown 1=Boolean 2=Integer 3=Real 4=Decimal 5=Character 6=Time 7=GMT Time 8=Date 9=Timestamp 10=GMT Timestamp 11=Interval 12=Byte Array 13=Bit Array 14=Pointer
	Text	String	The data value

Notes:

1. The File, Line, Function, and Text elements should not be used for exception handling decision making. These elements ensure that information can be written to a log for use by IBM service personnel.
2. The Type, Name, and Label elements define the object (usually a Message Flow node) that was processing the message when the exception condition occurred.
3. The Catalog, Severity, and Number elements define an NLS message: the Insert elements that contain the two name-value elements shown define the inserts into that NLS message.
4. NLS message catalog name and NLS message number refer to a translatable message catalog and message number.

The exception description structure can be both repeated and nested to produce an exception list tree. In this tree:

- The depth (that is, the number of parent-child steps from the root) represents increasingly detailed information for the same exception.
- The width of the tree represents the number of separate exception conditions that occurred before processing was abandoned. You will find that this number

is usually one, and results in an exception tree that consists of a number of exception descriptions connected as children of each other.

Figure 83 on page 379 illustrates one way in which an exception list can be constructed.

```
ExceptionList {
    RecoverableException = {                          1
        File     = 'f:/build/argo/src/DataFlowEngine/ImbDataFlowNode.cpp'
        Line     = 538
        Function = 'ImbDataFlowNode::createExceptionList'
        Type     = 'ComIbmComputeNode'
        Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
        Label    = 'mf1.Compute1'
        Text     = 'Node throwing exception'
        Catalog  = 'MQSeriesIntegrator2'
        Severity = 3
        Number   = 2230
        RecoverableException = {                      2
            File     = 'f:/build/argo/src/DataFlowEngine/ImbRdlBinaryExpression.cpp'
            Line     = 231
            Function = 'ImbRdlBinaryExpression::scalarEvaluate'
            Type     = 'ComIbmComputeNode'
            Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
            Label    = 'mf1.Compute1'
            Text     = 'error evaluating expression'
            Catalog  = 'MQSeriesIntegrator2'
            Severity = 2
            Number   = 2439
            Insert   = {
               Type = 2
               Text = '2'
            }
            Insert   = {
               Type = 2
               Text = '30'
            }
            RecoverableException = {                  3
                File     = 'f:/build/argo/src/DataFlowEngine/ImbRdlValueOperations.cpp'
                Line     = 257
                Function = 'intDivideInt'
                Type     = 'ComIbmComputeNode'
                Name     = '0e416632-de00-0000-0080-bdb4d59524d5'
                Label    = 'mf1.Compute1'
                Text     = 'Divide by zero calculating '%1 / %2''
                Catalog  = 'MQSeriesIntegrator2'
                Severity = 2
                Number   = 2450
                Insert   = }
                   Type = 5
                   Text = '100 / 0'
                }
            }
        }
    }
}
```

Figure 83. Exception list structure

Notes:

1. The first exception description **1** is a child of the root. This identifies error number 2230, indicating an exception has been thrown. The node that has thrown the exception is also identified (*mf1.Compute1*).
2. Exception description **2** is a child of the first exception description **1**. This identifies error number 2439.
3. Exception description **3** is a child of the second exception description **2**. This identifies error number 2450, which indicates that the node has attempted to divide by zero.

Exception handling paths will base their decisions on the number of exception conditions on:

- The message number, which identifies the type of exception that has occurred.
- The label, which is the known name of the object in which the exception occurred.

Figure 84 illustrates an extract of ESQL to show how you can set up a Compute node to use the exception list. The ESQL loops through the exception list to the last (nested) exception description, and extracts the error number. This error relates to the original cause of the problem and normally provides the most precise information. Subsequent action taken by the message flow can be decided by the error number retrieved in this way.

```
/* Error number extracted from exception list */
DECLARE Error INTEGER;
/* Current path within the exception list */
DECLARE Path CHARACTER;

/* Start at first child of exception list */
SET Path = 'InputExceptionList.*[1]';

/* Loop until no more children */
WHILE EVAL( 'FIELDNAME(' || Path || ') IS NOT NULL' ) DO

  /* Check if error number is available */
  IF EVAL( 'FIELDNAME(' || Path || '.Number) IS NOT NULL' ) THEN
    /* Remember only the deepest error number */
    SET Error = EVAL( Path || '.Number' );
  END IF;

 /* Step to last child of current element (usually a nested exception list */
 SET Path = Path || '.*[LAST]';

END WHILE; /* End loop */
```

Figure 84. Retrieving the exception error code

Appendix D. MQSeries message header parsers

The following topics are discussed:

- “Parser overview.”
- “The properties pseudo parser” on page 382.
- “Maintaining header integrity” on page 382.
- “The MQSeries headers parsers” on page 383.

Parser overview

When a message is received by a message flow, its constituent parts are passed to the correct parser for interpretation (unless interpretation is not required: for example if a whole message is copied).

With the exception of the MQMD, which must be the first header, the order of the headers preceding the message body is not important: the parser for each header processes that header independently. However, the fields are parsed in a particular order that is governed by the parser: you cannot predict or rely on the order chosen.

Fields within the message tree have the following datatypes that are equivalent to the corresponding fields MQSeries structure description, except where noted:

- MQLONG types are represented as INTEGER.
- MQCHAR and MQCHARn are represented as CHARACTER.
- MQBYTE and MQBYTEn are represented as BYTEARRAY.
- Date and time fields are represented as:
  - TIMESTAMP if the field can be converted to a valid TIMESTAMP
  - CHARACTER if the field cannot be converted to a valid TIMESTAMP
- The Expiry field in the MQMD is a special case:
  - If it is set to -1 (unlimited) it is converted to an integer
  - If it is not set to -1, it is converted to a TIMESTAMP

Fields in MQSeries structures, excluding MQRFH2 folders, are represented by name and value elements. MQRFH2 folder names are represented by name only elements. Structure length fields and structure identifier fields are not visible, and are filled in with appropriate values by the broker.

Other fields that are updated by the broker include all Format fields and domain fields in those parsers that support them. See “Maintaining header integrity” on page 382 for details of this process.

The properties pseudo parser

Every message has a set of standard properties that you can manipulate in the message flow nodes in the same way as any other property. The majority of these fields map to fields in the supported MQSeries headers and are passed to the appropriate parser when a message is delivered from one node to another.

If no parser is capable of receiving the property, the property is kept in the property parser until such time as a suitable parser can be found. If the message is converted to a bitstream, for example in an output node, any properties remaining solely in the property parser are discarded.

All messages generated by IBM supplied nodes provide a properties folder for the message as the first child of the root. It is not a requirement for a message to have a properties folder, although it is recommended. If you are using your own (plug-in) nodes, the interface provided does not automatically generate the properties folder for a message: if you want one in a message, you must create the folder yourself.

Having transmitted properties to each appropriate parser, the properties parser requests the values back from the owning parser. This ensures that the cached values of the properties are consistent with the message on entry and exit from each node. The state within any given node is dependant on the behavior of the node.

The Root name for this parser is "Properties". The standard properties are:

- MessageSet
- MessageType
- MessageFormat
- Encoding
- CodedCharSetId
- Transactional
- Persistence
- CreationTime
- ExpirationTime
- Priority
- Topic (this field contains a list)

Maintaining header integrity

The broker ensures that the integrity of the headers that precede a message body is maintained. The format of each part of the message is defined by the Format field in the immediately preceding header:

- The format of the first header is known, because this must be MQMD.
- The format of the next (second) part of the message, which might be another header, or the message body, is set in the Format field in the MQMD.
- If a third part of the message exists, its format is defined in the format field of the second part of the message.

This process is repeated as many times as is required by the number of headers that precede the message body. You do not have to populate these fields yourself: the broker handles this sequence for you.

If the body parser is not understood by MQSeries, the current Format field is checked. If it currently contains a registered parser name, it is set to MQFMT_NONE. The domain field is always updated. These actions might result in information explicitly stored by an SQL expression being replaced by the broker.

The MQSeries headers parsers

The following sections define the element names, types, and attributes for each of the supported MQSeries headers.

The following parsers are described:

- "The MQCIH parser" on page 384
- "The MQDLH parser" on page 386
- "The MQIIH parser" on page 387
- "The MQMD parser" on page 388
- "The MQMDE parser" on page 389
- "The MQRFH parser" on page 390
- "The MQRFH2 parser" on page 391
- "The MQRMH parser" on page 392
- "The MQSAPH parser" on page 393
- "The MQWIH parser" on page 394
- "The SMQ_BMH parser" on page 395
- "The BLOB parser" on page 396

The MQCIH parser

The Root name for this parser is "MQCIH". Table 45 lists the elements native to the MQCIH header.

Table 45 (Page 1 of 2). MQCIH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ReturnCode	INTEGER	Name Value
CompCode	INTEGER	Name Value
Reason	INTEGER	Name Value
UOWControl	INTEGER	Name Value
GetWaitInterval	INTEGER	Name Value
LinkType	INTEGER	Name Value
OutputDataLength	INTEGER	Name Value
FacilityKeepTime	INTEGER	Name Value
ADSDescriptor	INTEGER	Name Value
ConversationalTask	INTEGER	Name Value
TaskEndStatus	INTEGER	Name Value
Facility	BYTEARRAY	Name Value
Function	CHARACTER	Name Value
AbendCode	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
Reserved1	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
RemoteSysId	CHARACTER	Name Value
RemoteTransId	CHARACTER	Name Value
TransactionId	CHARACTER	Name Value
FacilityLike	CHARACTER	Name Value
AttentionId	CHARACTER	Name Value
StartCode	CHARACTER	Name Value
CancelCode	CHARACTER	Name Value
NextTransactionId	CHARACTER	Name Value
Reserved2	CHARACTER	Name Value
Reserved3	CHARACTER	Name Value
CursorPosition	INTEGER	Name Value
ErrorOffset	INTEGER	Name Value
InputItem	INTEGER	Name Value

Table 45 (Page 2 of 2). MQCIH parser element names, types, and attributes

Element Name	**Element Data Type**	**Element Attributes**
Reserved4	INTEGER	Name Value

The MQDLH parser

The Root name for this parser is "MQDLH". Table 46 lists the elements native to the MQDLH header.

Table 46. MQDLH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Reason	INTEGER	Name Value
DestQName	CHARACTER	Name Value
DestQMgrName	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

The MQIIH parser

The Root name for this parser is "MQIIH". Table 47 lists the elements native to the MQIIH header.

Table 47. MQIIH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
LTermOverride	CHARACTER	Name Value
MFSMapName	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
TranInstanceId	BYTEARRAY	Name Value
TranState	CHARACTER	Name Value
CommitMode	CHARACTER	Name Value
SecurityScope	CHARACTER	Name Value
Reserved	CHARACTER	Name Value

The MQMD parser

The Root name for this parser is "MQMD". Table 48 lists the orphan elements adopted by the MQMD header.

Table 48. MQMD parser orphan element names, types, and attributes

Element Name	Element Data Type	Element Attributes
SourceQueue	CHARACTER	Name Value
Transactional	CHARACTER	Name Value

Table 49 lists the elements native to the MQMD header.

Table 49. MQMD parser native element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Report	INTEGER	Name Value
MsgType	INTEGER	Name Value
Expiry	INTEGER/TIMESTAMP	Name Value
Feedback	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Priority	INTEGER	Name Value
Persistence	INTEGER	Name Value
MsgId	BYTEARRAY	Name Value
CorrelId	BYTEARRAY	Name Value
BackoutCount	INTEGER	Name Value
ReplyToQ	CHARACTER	Name Value
ReplyToQMgr	CHARACTER	Name Value
UserIdentifier	CHARACTER	Name Value
AccountingToken	BYTEARRAY	Name Value
ApplIdentityData	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value
ApplOriginData	CHARACTER	Name Value
GroupId	BYTEARRAY	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

The MQMDE parser

The Root name for this parser is "MQMDE". Table 50 lists the elements native to the MQMDE header.

Table 50. MQMDE parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
GroupId	BYTEARRAY	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

The MQRFH parser

The Root name for this parser is "MQRFH". Table 51 lists the elements native to the MQRFH header.

Table 51. MQRFH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value

Other name value elements might be present that contain information as parsed from or destined for the option buffer. See the Rules and Format header documentation for specific names and values.

The MQRFH2 parser

The Root name for this parser is "MQRFH2". Table 52 lists the elements native to the MQRFH2 header.

Table 52. MQRFH2 parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
NameValueCCSID	INTEGER	Name Value

Other name and child name value elements might be present that contain information as parsed from or destined for the option buffer. See the Rules and Format header section in the MQSeries Integrator Documentation for further details.

The MQRMH parser

The Root name for this parser is "MQRMH". Table 53 lists the elements native to the MQRMH header.

Table 53. MQRMH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ObjectType	CHARACTER	Name Value
ObjectInstanceId	BYTEARRAY	Name Value
SrcEnv	CHARACTER[1]	Name Value
SrcName	CHARACTER[2]	Name Value
DestEnv	CHARACTER[3]	Name Value
DestName	CHARACTER[4]	Name Value
DataLogicalLength	INTEGER	Name Value
DataLogicalOffset	INTEGER	Name Value
DataLogicalOffset2	INTEGER	Name Value

Notes:

1. This field represents both SrcEnvLength and Offset
2. This field represents both SrcNameLength and Offset
3. This field represents both DestEnvLength and Offset
4. This field represents both DestNameLength and Offset

The MQSAPH parser

The Root name for this parser is "MQSAPH". Table 54 lists the elements native to the MQSAPH header.

Table 54. MQSAPH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
Client	CHARACTER	Name Value
Language	CHARACTER	Name Value
HostName	CHARACTER	Name Value
UserId	CHARACTER	Name Value
Password	CHARACTER	Name Value
SystemNumber	CHARACTER	Name Value
Reserved	BYTEARRAY	Name Value

The MQWIH parser

The Root name for this parser is "MQWIH". Table 55 lists the elements native to the MQWIH header.

Table 55. MQWIH parser element names, types, and attributes

Element Name	**Element Data Type**	**Element Attributes**
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ServiceName	CHARACTER	Name Value
ServiceStep	CHARACTER	Name Value
MsgToken	BYTEARRAY	Name Value
Reserved	CHARACTER	Name Value

The SMQ_BMH parser

The Root name for this parser is "SMQ_BMH". Table 56 lists the elements native to the SMQ_BMH header.

Table 56. SMQ_BMH parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
ErrorType	INTEGER	Name Value
Reason	INTEGER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

The BLOB parser

The Root name for this parser is "BLOB". Table 57 lists the elements native to the BLOB header.

Table 57. BLOB parser element names, types, and attributes

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
BLOB	BYTEARRAY[1]	Name Value
UnkownParserName	CHARACTER[2]	Name Value

Notes:

1. This field contains the remaining unparsed bitstream from the message. It is represented as a BYTEARRAY and may be manipulated as such.
2. This field (if present) contains the class name of the parser that would have been chosen in preference to the BLOB parser. This information is used by the header integrity routine (described in “Maintaining header integrity” on page 382) to ensure that the semantic meaning of the message is preserved.

Appendix E. NEON Rules and Formatter

The NEON rules and formatter are inherited by MQSeries Integrator from Version 1 of the product. MQSeries Integrator Version 2 message flow nodes do not read directly from an input queue but instead use the MQInput node. If you are migrating from Version 1 and need to incorporate NEON messages into your message flows, you can wire the out terminals on your MQInput nodes into the in terminal on a NEON Rules node. However, rules and formatting operations will run unchanged on Version 2 and you will not have to change any of your client applications.

MQSeries Integrator Version 2 supplies a message flow that provides function equivalent to the MQSeries Integrator Version 1.1 daemon. This message flow is described in “Using the IBM-supplied message flows” on page 175.

NEONFormatter and NEONRules nodes

If you are not using MQSeries Integrator Version 1 you do not have any migration tasks to complete. Instead, the nodes will appear in the list of IBM supplied message flow nodes and behave as described in “The Message Sets view” on page 56.

Further information about the NEON nodes is in this appendix and in the MQSeries Integrator help, accessible by highlighting either node, right clicking, and selecting Help from the drop down list.

NEON formatter and rules engine

The NEON formatter and rules engines define a set of formats and rules that govern how an incoming message is processed.

They are both inherited from MQSeries Integrator Version 1. They enable you to receive messages from Version 1 and so act as an aid to migration, but the tasks you have to perform to complete migration from Version 1 involve the broker only.

Combining NEON rules with MQSeries Integrator

When you design message flows that combine the use of the NEONRules and NEONFormatter nodes in conjunction with other message flow nodes, the following conditions apply:

- A NEONRules or NEONFormatter node can only process messages defined using the NEONFormatter interface (they cannot process messages defined using the MRM).
- A message flow node can parse a message that has been defined as an input format using the NEONFormatter interface.
- A message flow node cannot create or modify a message whose format has been defined using the NEONFormatter tool unless it is a NEON node.

According to these conditions, the procedures in the following table should be carried out when you implement message flows that include NEON nodes.

The NEON Rules node has no properties. However, it does have to access the database in which the NEON rules are defined. Because the NEON message parser has to access the same database, all the NEONRules nodes in a message flow, and any NEON parsers, use the same set of database connection parameters.

These database connection parameters are retrieved from a configuration file that matches the format of the MQSeries Integrator Version 1 rules engine configuration file. For migration purposes the MQSI_PARAMETERS_FILE environment variable can be pointed at that file. You must create and initialize this system environment variable after installation of MQSeries Integrator Version 2. If you use this parameter on Windows NT, you must restart your system for it to take effect.

The minimum configuration file that is necessary is as follows:

```
[Rules Database Connection]
#
# rules and formatter database connection information (mandatory)
# exceptions/notes:
#   - leave "DatabaseInstance" as "???" (Oracle and DB2 only)
#   - enter the database name as the value of "ServerName" (DB2 only)
#
ServerName       = ???
UserId           = ???
Password         = ???
DatabaseInstance = ???
#
# DatabaseType is a numeric with these values:
#    SYBASE (CTLIB bindings) = 1
#    SYBASE (DBLIB bindings) = 2
#    MSSQL                   = 4
#    DB2                     = 5
#    ODBC                    = 6
#    ORACLE (version 7.x)    = 8
#    ORACLE (version 8.x)    = 9
#
DatabaseType = 5

#
# end of file!
#
```

A message flow node in MQSeries Integrator will accept messages made up of an MQMD (full details of which are in the on-line help), optionally followed by an MQRFH or an MQRFH2 header, and a message body that can be parsed by the NEON parser.

An MQSeries message flow node takes the portion of the bit-stream representation of the message that is being parsed by the NEON parser. It passes that representation to the NEON rules processor using the message type and application group parameters retrieved from the values of the Type and Set standard properties.

The MQSeries message flow node then processes the actions returned from the rules processor. Three actions are recognized. These are reformat, putqueue, and propagate. If no rule is hit then the original message is propagated to the noHit terminal. If any errors occur whilst evaluating the rules, the original message is propagated to the failure terminal.

Note: Errors that occur in message flow nodes further down the message flow than any of the output terminals are not caught by the NEON rules engine. They are caught by the input node or last TryCatch node instead.

Table 58. Procedures for implementing message flows with NEON nodes

Circumstance	Action
Nodes preceding the NEON node(s) only examine the content of the message and do not modify the message content in any way.	No action required. The message format only needs to be defined as an input format to NEON components.
Node following the NEON node(s) only examine the properties of the message or the message headers. They do not examine the content of the message.	No action required.
Message flow nodes that follow the NEON nodes in a message flow examine and modify the content of a message.	The format of the message passed out of the NEON node must be defined either as both an input and output format in the NEON dictionary, or as an output format in the NEON dictionary and as any format in the MRM. In this second case, the interchange message must be a format that can be built by the NEON formatter and parsed by the IBM parser.

To make sure that interchange messages are defined in both dictionaries, you must describe the message format as a COBOL copybook and then import the record format into both dictionaries.

NEON rules engine

The NEON rules engine works by evaluating a set of rules against an incoming NEON message. The set of rules applied is selected based on the Application group and message type of the message. (These values are replaced respectively by message set and message type properties in Version 2.)

Each rule in the set has a list of subscriptions attached to it, that are executed if the rule evaluates to true. The possible actions that can be performed as part of this subscription are:

- Reformat
- Putqueue
- Propagate

These actions result in the following:

- The Reformat action allows a message to be formatted between a NEON input format and a NEON output format.
- The Putqueue action puts a copy of the message to the *putqueue* terminal and attaches the queue given in the action to the message's destination list. The Putqueue action has several options that govern how the headers are copied

across (see the *MQSeries Integrator User's Guide Version 1.1* for further details).

- The Propagate action puts a copy of the message to the *propagate* terminal. It has identical options to the Putqueue action but does not attach a queue name to the destination list. The Control Center online help has further information about the use of the Propagate action.

If no rules evaluate to true, the message is propagated to the *noHits* terminal unchanged.

If a failure occurs whilst in the rules node the message is propagated to the *failure* terminal.

For full details of how to use and configure the NEON Rules engine, see the *MQSeries Integrator User's Guide Version 1.1*).

Appendix F. Notices

Trademarks

Glossary of terms and abbreviations

This glossary defines MQSeries Integrator terms and abbreviations used in this book. If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute. Copies may be ordered from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

Access Control List (ACL). The list of principals that have explicit permissions (to publish, to subscribe to, and to request persistent delivery of a publication message) against a topic in the topic tree. The ACLs define the implementation of topic-based security.

ACL. Access Control List.

AMI. Application Messaging Interface.

Application Messaging Interface (AMI). The programming interface provided by MQSeries that defines a high level interface to message queuing services. See also **MQI** and **JMS**.

B

blob. Binary Large OBject. A block of bytes of data (for example, the body of a message) that has no discernible meaning, but is treated as one solid entity that cannot be interpreted. Also written as BLOB.

broker. See **message broker**.

broker domain. A collection of brokers that share a common configuration, together with the single Configuration Manager that controls them.

C

callback function. See *implementation function*.

category. An optional grouping of messages that are related in some way. For example, messages that relate to a particular application.

| **check in**. The Control Center action that stores a new or updated resource in the configuration or message respository.

| **check out**. The Control Center action that extracts and locks a resoource from the configuration or message respository for local modification by a user. Resources from the two repositories can only be worked on when they are checked out by an authorized user, but can be viewed (read only) without being checked out.

collective. A hyperconnected (totally connected) set of brokers forming part of a multi-broker network for publish/subscribe applications.

configuration. In the broker domain, the brokers, execution groups, message flows and message sets assigned to them, topics and access control specifications.

Configuration Manager. A component of MQSeries Integrator that acts as the interface between the configuration repository and an executing set of brokers. It provides brokers with their initial configuration, and updates them with any subsequent changes. It maintains the broker domain configuration.

configuration repository. Persistent storage for broker configuration and topology definition.

connector. See **message processing node connector**.

content-based filter. An expression that is applied to the content of a message to determine how the message is to be processed.

context tag. A tag that is applied to an element within a message to enable that element to be treated differently in different contexts. For example, an element could be mandatory in one context and optional in another.

Control Center. The graphical interface that provides facilities for defining, configuring, deploying, and monitoring resources of the MQSeries Integrator network.

D

datagram. The simplest form of message that MQSeries supports. Also known as *send-and-forget*. This type of message does not require a reply. Compare with *request/reply*.

deploy. Make operational the configuration and topology of the broker domain.

| **destination list**. A list of internal and external destinations to which a message is sent. These can be nodes within a message flow (for example, when using the RouteToLabel and Label nodes) or MQSeries queues (when the list is examined by an MQOutput node to determine the final target for the message).

distribution list. A list of MQSeries queues to which a message can be put using a single statement.

| **Document Type Definition**. The rules that specify the structure for a particular class of SGML or XML documents. The DTD defines the structure with elements, attributes, and notations, and it establishes constraints for how each element, attribute, and notation can be used within the particular class of documents. A DTD is analogous to a database schema in that the DTD completely describes the structure for a particular markup language.

| **DTD**. Document Type Definition

E

e-business. A term describing the commercial use of the Internet and World Wide Web to conduct business (short for electronic-business).

element. A unit of data within a message that has business meaning, for example, street name

element qualifier. See **context tag**.

| **ESQL**. Extended SQL. A specialized set of SQL statements based on regular SQL, but extended with statements that provide specialized functions unique to MQSeries Integrator.

| **exception list**. A list of exceptions that have been generated during the processing of a message, with supporting information.

execution group. A named grouping of message flows that have been assigned to a broker. The broker is guaranteed to enforce some degree of isolation between message flows in distinct execution groups by ensuring that they execute in separate address spaces, or as unique processes.

Extensible Markup Language (XML). A W3C standard for the representation of data.

F

filter. An expression that is applied to the content of a message to determine how the message is to be processed.

format. A format defines the internal structure of a message, in terms of the fields and order of those fields. A format can be self-defining, in which case the message is interpreted dynamically when read.

G

graphical user interface (GUI). An interface to a software product that is graphical rather than textual. It refers to window-based operational characteristics.

I

implementation function. Function written by a third-party developer for a plug-in node or parser. Also known as a *callback function*.

input node. A message flow node that represents a source of messages for the message flow.

installation mode. The installation mode can be Full, Custom, or Broker only. The mode defines the components of the product installed by the installation process on Windows NT systems.

J

Java Database Connectivity (JDBC). An application programming interface that has the same characteristics as **ODBC** but is specifically designed for use by Java database applications.

Java Development Kit (JDK). A software package that can be used to write, compile, debug, and run Java applets and applications.

Java Message Service (JMS). An application programming interface that provides Java language functions for handling messages.

Java Runtime Environment. A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes and supporting files.

JDBC. Java Database Connectivity.

JDK. Java Development Kit.

JMS. Java Message Service. See also **AMI** and **MQI**.

JRE. Java Runtime Environment.

L

local error log. A generic term that refers to the logs to which MQSeries Integrator writes records on the local system. On Windows NT, this is the Event log. On UNIX systems, this is the syslog. See also system log. Note that MQSeries records many events in the log that are not errors, but information about events that occur during operation, for example, successful deployment of a configuration.

M

message broker. A set of execution processes hosting one or more message flows.

messages. Entities exchanged between a broker and its clients.

message dictionary. A repository for (predefined) message type specifications.

message domain. The source of a message definition. For example, a domain of MRM identifies messages defined using the Control Center, a domain of NEON identifies messages created using the NEON user interfaces.

message flow. A directed graph that represents the set of activities performed on a message or event as it passes through a broker. A message flow consists of a set of message processing nodes and message processing node connectors.

message flow component. See **message flow**.

message parser. A program that interprets a message bitstream.

message processing node. A node in the message flow, representing a well defined processing stage. A message processing node can be one of several primitive types or can represent a subflow.

message processing node connector. An entity that connects the output terminal of one message processing node to the input terminal of another. A message processing node connector represents the flow of control and data between two message flow nodes.

message queue interface (MQI). The programming interface provided by MQSeries queue managers. The programming interface allows application programs to access message queuing services. See also **AMI** and **JMS**.

message repository. A database holding message template definitions.

message set. A grouping of related messages.

message template. A named and managed entity that represents the format of a particular message. Message templates represent a business asset of an organization.

message type. The logical structure of the data within a message. For example, the number and location of character strings.

metadata. Data that describes the characteristic of stored data.

MQI. Message queue interface.

MQRFH. An architected message header that is used to provide metadata for the processing of a message. This header is supported by MQSeries Publish/Subscribe.

MQRFH2. An extended version of MQRFH, providing enhanced function in message processing.

multi-level wildcard. A wildcard that can be specified in subscriptions to match any number of levels in a topic.

N

node. See **message processing node**.

O

ODBC. Open Database Connectivity.

Open Database Connectivity. A standard application programming interface (API) for accessing data in both relational and non-relational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group.

output node. A message processing node that represents a point at which messages flow out of the message flow.

P

plug-in. An extension to the broker, written by a third-party developer, to provide a new message processing node or message parser in addition to those supplied with the product. See also *implementation function* and *utility function*.

point-to-point. Style of messaging application in which the sending application knows the destination of the message. Compare with *publish/subscribe*.

POSIX. Portable Operating System Interface For Computer Environments. An IEEE standard for computer operating systems (for example, AIX and Sun Solaris).

predefined message. A message with a structure that is defined before the message is created or referenced. Compare with *self-defining message*.

primitive. A message processing node that is supplied with the product.

principal. An individual user ID (for example, a log-in ID) or a group. A group can contain individual user IDs and other groups, to the level of nesting supported by the underlying facility.

property. One of a set of characteristics that define the values and behaviors of objects in the Control Center. For example, message processing nodes and deployed message flows have properties.

publication node. An end point of a specific path through a message flow to which a client application subscribes. A publication node has an attribute, subscription point. If this is not specified, the publication node represents the default subscription point for the message flow.

publish/subscribe. Style of messaging application in which the providers of information (publishers) are decoupled from the consumers of that information (subscribers) using a broker. Compare with *point-to-point*. See also *topic*.

publisher. An application that makes information about a specified topic available to a broker in a publish/subscribe system.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that programs can access messages on the queues that the queue manager owns.

R

retained publication. A published message that is kept at the broker for propagation to clients that subscribe at some point in the future.

request/reply. Type of messaging application in which a request message is used to request a reply from another application. Compare with *datagram*.

rule. A rule is a definition of a process, or set of processes, applied to a message on receipt by the broker. Rules are defined on a message format basis, so any message of a particular format will be subjected to the same set of rules.

S

self-defining message. A message that defines its structure within its content. For example, a message coded in XML is self-defining. Compare with *pre-defined message*.

send and forget. See *datagram*.

setup type. The definition of the type of installation requested on Windows NT systems. This can be one of **Full**, **Broker only**, or **Custom**.

shared. All configuration data that is shared by users of the Control Center. This data is not operational until it has been deployed.

signature. The definition of the external characteristics of a message processing node.

single-level wildcard. A wildcard that can be specified in subscriptions to match a single level in a topic.

subscriber. An application that requests information about a specified topic from a publish/subscribe broker.

subscription. Information held within a publication node, that records the details of a subscriber application, including the identity of the queue on which that subscriber wants to receive relevant publications.

subscription filter. A predicate that specifies a subset of messages to be delivered to a particular subscriber.

subscription point. An attribute of a publication node that differentiates it from other publication nodes on the same message flow and therefore represents a specific path through the message flow. An unnamed publication node (that is, one without a specific subscription point) is known as the default publication node.

| **system log**. A generic term used in the MQSeries Integrator messages (BIPxxx) that refers to the local error logs to which records are written on the local system. On Windows NT, this is the Event log. On UNIX systems, this is the syslog. See also local error log.

T

terminal. The point at which one node in a message flow is connected to another node. Terminals enable you to control the route that a message takes, depending whether the operation performed by a node on that message is successful.

topic. A character string that describes the nature of the data that is being published in a publish/subscribe system.

| **topic based subscription**. A subscription specified by a subscribing application that includes a topic for filtering of publications.

| **topic security**. The use of ACLs applied to one or more topics to control subscriber access to published messages.

topology. In the broker domain, the brokers, collectives, and connections between them.

transform. A defined way in which a message of one format is converted into one or more messages of another format.

U

| **Uniform Resource Identifier**. The generic set of all names and addresses that refer to World Wide Web resources.

| **Uniform Resource Locator**. A specific form of URI that identifies the address of an item on the World Wide Web. It includes the protocol followed by the fully qualified domain name (sometimes called the host name) and the request. The Web server typically maps the request portion of the URL to a path and file name. Also known as Universal Resource Locator.

| **URI**. Uniform Resource Identifier

| **URL**. Uniform Resource Locator

User Name Server. The MQSeries Integrator component that interfaces with operating system facilities to determine valid users and groups.

utility function. Function provided by MQSeries Integrator for the benefit of third-party developers writing plug-in nodes or parsers.

W

warehouse. A persistent, historical datastore for events (or messages). The **Warehouse** node within a message flow supports the recording of information in a database for subsequent retrieval and processing by other applications.

wildcard. A character that can be specified in subscriptions to match a range of topics. See also *multilevel wildcard* and *single-level wildcard*.

wire format. This describes the physical representation of a message within the bit-stream.

W3C. World Wide Web Consortium. An international industry consortium set up to develop common protocols to promote evolution and interoperability of the World Wide Web.

X

XML. Extensible Markup Language.

Index

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5602-01