MQSeries® Adapter Kernel for Multiplatforms

Quick Beginnings

Version 1 Release 1

GC34-5855-04

MQSeries® Adapter Kernel for Multiplatforms

Quick Beginnings

Version 1 Release 1

GC34-5855-04

Note: Before using this information and the product it supports, read the information in "Notices" on page 97.

Fifth Edition (December 2000)

This edition applies to version 1, release 1 of MQSeries Adapter Kernel for Multiplatforms (product number 5648-D75) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can make comments on this information via e-mail at idrcf@hursley.ibm.com.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Welcome to the MQSeries Adapter Kernel Quick Beginnings

This document describes the MQSeries® Adapter Kernel and explains how to plan for, install, and use it.

To make the kernel ready to use, perform the following general steps:

1. Read "Chapter 1. About MQSeries Adapter Offering" on page 1.
2. Prepare for installation. See "Preparing for installation" on page 31 for details.
3. Install the kernel. See "Installing the kernel" on page 32 for details.
4. Verify the installation. See "Verifying the installation" on page 38 for details.
5. Configure the kernel. See "Configuring the kernel" on page 46 for details.
6. If desired, configure optional software to work with the kernel. See "Configuring MQSeries and MQSeries Integrator" on page 64 for details.
7. Build your adapters by using the MQSeries Adapter Builder, then test and deploy them. See the MQSeries Adapter Builder documentation for details.
8. Start the kernel. See "Starting the kernel" on page 65 for details.

To use this information, you also need to know about prerequisite and optional products. See "Chapter 2. Planning to install the kernel" on page 23. See also "References" on page 73.

Who should use this information

This information is for those who need to plan for, install, or use the MQSeries Adapter Kernel.

Related information

For additional information, see the following:

- The `readme.txt` file. This file possibly contains information that became available after this book was completed. Before installation, the `readme.txt` file is located in the root directory of the product CD-ROM. After installation, the `readme.txt` file is located in the root directory of the MQSeries Adapter Kernel installation.
- The *Problem Determination Guide*, form number SC34-5897, which describes tools, including trace, for solving specific problems with the MQSeries

Adapter Kernel. The *Problem Determination Guide* is available in the MQSeries Adapter Kernel Information Center, which is installed with the product.

- The online application programming interface (API) documentation that is provided in the MQSeries Adapter Kernel Information Center. This information is provided only as an aid to understanding how the kernel functions and an aid to diagnostics. See "Chapter 5. Using MQSeries Adapter Kernel APIs" on page 71.
- MQSeries Adapter Builder information, including books and help system.
- The MQSeries product family Web site at http://www.ibm.com/software/ts/mqseries/.

  By following links from this Web site you can:

  - Obtain the latest information about the MQSeries product family, including MQSeries Adapter Offering.
  - Access MQSeries books in HTML and PDF formats, possibly including a more recent edition of this book.
  - Download MQSeries SupportPacs.

Conventions

MQSeries Adapter Kernel documentation uses the following typographical and keying conventions.

Table 1. Conventions used in this book

Convention	Meaning
Bold	Indicates command names. When referring to graphical user interfaces (GUIs), indicates menus, menu items, labels, and buttons.
`Monospace`	Indicates text you must enter at a command prompt and values you must use literally, such as file names, paths, and elements of programming languages such as functions, classes, and methods. Monospace also indicates screen text and code examples.
Italics	Indicates variable values you must provide (for example, you supply the name of a file for *fileName*). Italics also indicates emphasis and the titles of books.
`%`	Represents the UNIX command-shell prompt for a command that does not require **root** privileges.
`#`	Represents the UNIX command-shell prompt for a command that requires **root** privileges.
`C:\>`	Represents the command prompts on Windows® systems.
>	When used to describe a menu, shows a series of menu selections. For example, “Click **File > New**” means “From the **File** menu, click the **New** command.”
Entering commands	When instructed to “enter” or “issue” a command, type the command and then press Return. For example, the instruction “Enter the **ls** command” means type **ls** at a command prompt and then press Return.
[ ]	Enclose optional items in syntax descriptions.
{ }	Enclose lists from which you must choose an item in syntax descriptions.
|	Separates items in a list of choices enclosed in braces ({ }) in syntax descriptions.
...	Ellipses in syntax descriptions indicate that you can repeat the preceding item one or more times. Ellipses in examples indicate that information was omitted from the example for the sake of brevity.

Note: The term Epic appears in some values and names in the kernel software and in this book. With regard to the MQSeries Adapter Offering, this term has no significance in itself.

Summary of changes

The fifth edition (the current edition) includes the following changes from the fourth edition:

- Information on using the kernel on the Windows® 2000, OS/400®, HP-UX, and Solaris platforms. Support for these platforms is new in MQSeries Adapter Kernel version 1.1. The kernel was previously available only on Windows NT® and AIX®.
- Updates of all installation instructions to reflect MQSeries Adapter Kernel version 1.1.
- Information on using the `aqmconfig.xml` file to configure MQSeries Adapter Kernel. The kernel was previously configured with the `aqmconfig.properties` file. See "The configuration file" on page 47 for details.
- Information on the new MQ and JMS (Java Messaging Service) communication modes. See "Appendix A. Communication modes" on page 75 for details.
- Information on tracing was moved from this document to the new *Problem Determination Guide* document. See the *Problem Determination Guide* for detailed information on using MQSeries Adapter Kernel's tracing capabilities.

The fourth edition included the following changes from the third edition:

- Instructions for installing the Corrective Service Diskette (CSD) release of MQSeries Adapter Kernel version 1.0.
- Information about the transactional capabilities of MQSeries Adapter Kernel; see "Transactional capabilities" on page 22.
- Information about thread-scheduling policies on AIX; see Step 20 on page 18.
- Clarification of the supported C compiler on AIX; see ""Prerequisites for AIX"" on page 24.
- A sample of the `aqmsetup` file; see "Appendix E. Sample of the setup file" on page 95.

Chapter 1. About MQSeries Adapter Offering

IBM MQSeries Adapter Kernel is part of a set of application integration products that together are called IBM MQSeries Adapter Offering. IBM MQSeries Adapter Offering works with MQSeries messaging and other messaging services to enable you to reduce the risk, complexity, and cost of managing the point-to-point integration of your business processes.

In *point-to-point* integration, each application interfaces individually with each of the other applications. Each interface is different and there are many different interfaces. A change in one application typically requires changes to many interfaces. As the number of applications increases, the cost of point-to-point integration rapidly increases. Integrating each new application typically requires more work than integrating the last one.

With MQSeries Adapter Offering, you can evolve from using point-to-point integration to using *one-to-any* integration. Benefits of one-to-any integration include the following:

- All applications can use one common interface.
- Data from a *source application*, in the form of a *message*, is *routed* to one or more *target applications*.
- A change in one application typically affects only that one interface.
- Using a common interface that is application neutral—for example, an industry standard such as extensible markup language (XML)—can be even more cost effective. More applications can be supported with less effort.
- As the number of applications increases, one-to-any integration becomes even more cost effective. Adding each new application typically does not require significant changes to the interfaces of all the other applications.
- Integration work can be automated and can be based on templates.

MQSeries Adapter Offering can be deployed without changing applications or business processes at all. Typically, all integration work is performed in MQSeries Adapter Offering, thus reducing the need to write custom code.

In MQSeries Adapter Offering, the interface to or from one application is provided by an *adapter*. All applications need an adapter to provide the interface between the application environment and the messaging environment. Each adapter is specific to an application.

MQSeries Adapter Kernel can optionally be deployed with MQSeries Integrator to perform brokering and message transformation. MQSeries Adapter Offering can be complemented by service offerings from IBM and others.

Example uses of adapters include the following:

- Add a sales order.
- Synchronize a customer record.
- Synchronize an inventory record.
- Synchronize an item.
- Synchronize a sales order.

Build time and run time

MQSeries Adapter Offering consists of two primary components, the Adapter Builder (also called the builder) and the Adapter Kernel (also called the kernel). This section describes these components, as well as the adapters that are built and run by the Adapter Offering.

adapter

Software that provides an interface to or from an application. Adapters are built in the MQSeries Adapter Builder. Typically, each adapter is built to be specific to *one message type* that is sent from or to an application. Adapters themselves are not part of MQSeries Adapter Offering.

An adapter consists of C source code that compiles to a shared library. When the adapters and the MQSeries Adapter Kernel run together, they perform the run-time functionality of the MQSeries Adapter Offering.

Depending on how it is modeled in the MQSeries Adapter Builder, the adapter can contain a wide variety of functionalities such as controlflow, dataflow, sequential navigation, conditional branching including decision and iteration, data typing, storage of data context, transformation of data elements, transactional capabilities, logical operations, and custom code.

Adapters can be reused.

There are two types of adapters:

- Source adapters, for the application that sends the data.
- Target adapters, for the application that receives the data.

Sending one type of message from one application to a second application typically requires one source adapter and one target adapter. If the second application must send one type of message to the first application, another source adapter and another target adapter are required. Thus, in this case, to send one type of message from the first application to the second application and then to send another type of message from the second application back to the first application, four adapters are typically deployed.

A separate adapter is required for each message type.

MQSeries Adapter Builder

A graphical user interface (GUI) that enables you to build an adapter for virtually any application. The user interface is similar to MQSeries Integrator's user interface. For more information, see the MQSeries Web site at http://www.ibm.com/software/ts/mqseries/.

MQSeries Adapter Kernel

A set of application programming interfaces (APIs), several executable programs in C and Java™, and several configuration files. The kernel enables the deployment and execution of adapters. In addition to directly supporting adapters, the kernel performs related functions, including simple routing of messages and infrastructure services such as message construction, transactional control, tracing, and interfacing with MQSeries or other messaging software.

The kernel is installed on each computer on which a source adapter or a target adapter runs.

With MQSeries Adapter Offering, business processes and each application can remain isolated from the specifics of middleware, message details, and other applications. A common interface for messaging enables the addition of new applications without changing existing applications or business processes.

MQSeries Adapter Kernel can be deployed in two tiers. One tier is the source side of the run time; the other tier is the target side of the run time. Two-tier deployment provides efficient operation and low administrative overhead. A third tier for routing and delivery is not required to reside between the two sides of the run time. However, MQSeries Integrator can optionally be added to perform brokering, such as complex routing, data transformation, and data mediation. Using MQSeries Integrator adds a third tier.

Except where specified, the rest of this document pertains only to MQSeries Adapter Kernel. For detailed information about the MQSeries Adapter Builder, see that product's Information Center.

About the kernel

At its simplest, the run time—that is, the kernel and the adapters that you build—has the following purposes:

1. To transfer data from a source application to a target application.
2. To convert the source application's data to a message, typically in an application-neutral format, that is routed through the kernel, by using MQSeries or other messaging software.
3. To route the message to the target application.
4. To determine how to get the data to the target application.
5. To convert the data from the format of the message that is routed through the kernel through an adapter to the target application's format.

In this section, the kernel's functionality is discussed at a high level. The functionality is discussed in greater detail in "Run-time flow" on page 12.

There are two sides of the kernel:

- The *source side*, which begins when the message is received from the source adapter and ends when the message is put onto a message queue.
- The *target side*, which begins when the message is retrieved from the message queue and ends when the message is sent to the target adapter.

Each side typically resides on a different computer, but they can both reside on the same computer.

See Figure 1 on page 5. It depicts the following sequence.

Source side of the kernel

1. On the source side of the kernel, the source application sends the data in its *source application format,* by using an *application-specific interface,* to a source adapter that was built in the MQSeries Adapter Builder. A different source adapter is required for each message type, for example, for "add a sales order" or for "synchronize a customer record."

   The application-specific interface must be developed outside of the MQSeries Adapter Offering. The exact nature of the application-specific interface depends on the characteristics of the source application or target application. Examples include API calls and user exits, file reads and writes, database triggers, and message queues.

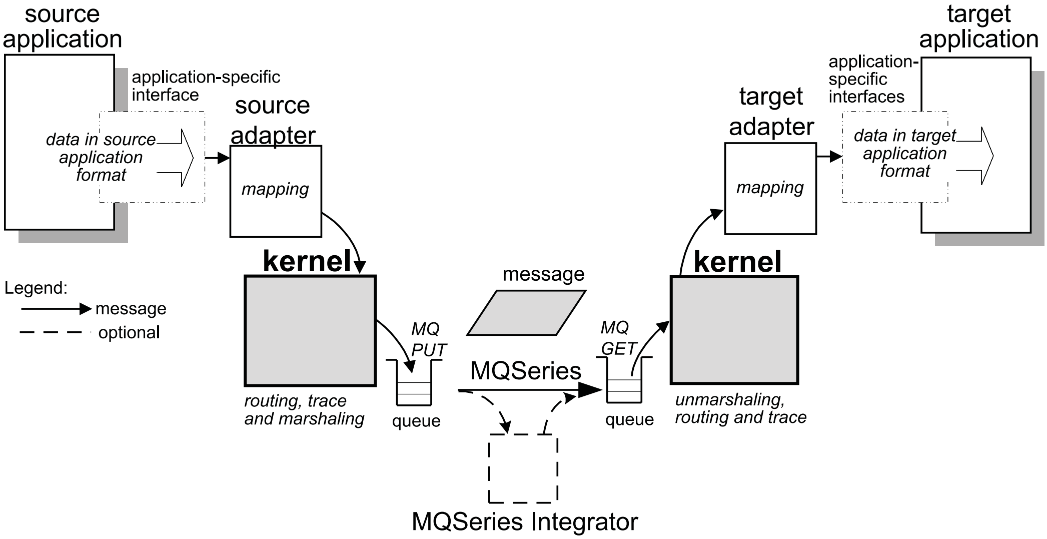
   Note that the source adapter is run in the source application's process. Any daemon or server that contains the source adapter needs to be started.
2. The source adapter performs its function according to how it was built. A typical function is transformation of data elements, that is, mapping elements from the source application format to an integration messaging

data format for body data. The body data and additional metadata representing control values are put into a kernel *message holder object*.

3. When the source adapter passes the message holder object to the kernel by using the native adapter, control values in the message holder object (*message control values*) are used by the kernel to control the marshaling of the message holder object into a communications message format and routing of the communications messages.

   If the message does not contain certain message control values, the kernel can use defaults or message control values obtained from the configuration file. For definitions of message control values, see “Message control values” on page 14.
4. The kernel performs its functions, including *message marshaling*, simple *routing*, and, optionally, *tracing*. See “Message and message format” on page 10, “Routing and delivery” on page 12, and “Tracing” on page 22.

Figure 1. Overview of MQSeries Adapter Offering.


source application
application-specific interface
data in source application format
source adapter
mapping
kernel
routing, trace and marshaling
Legend:
message
optional
MQ PUT
queue
message
MQSeries
MQSeries Integrator
MQ GET
queue
kernel
unmarshaling, routing and trace
target adapter
mapping
application-specific interfaces
data in target application format
target application

Delivery from source side to target side of the kernel

5. The kernel, by using its *native adapter*, puts the message on the appropriate message queue.

   There are two send methods used on the source side:

   - `sendMsg`, which sends the message and returns immediately. The `sendMsg` method can also be used with the `begin`, `commit`, and `rollback` methods to send messages *transactionally*; that is, messages can be sent if (and only if) other operations complete successfully. See “Transactional capabilities” on page 22 for more information.

- `sendRequestResponse`, which sends the message and waits for a response. The `sendRequestResponse` method cannot be issued transactionally.

  Note that a third method, `sendResponse`, is used on the target side of the kernel when the sender requests a response.

MQSeries or other messaging software transports the message. See “Role of MQSeries or other messaging software” on page 7. Note that MQSeries must already be configured to support MQSeries Adapter Offering.

Optionally, if MQSeries Integrator has been configured in the kernel as the destination, MQSeries Integrator can perform brokering functions. See “Role of MQSeries Integrator” on page 7. If the final destination, a message queue, has been configured in MQSeries Integrator’s rules or messageflows, then MQSeries Integrator sends the message to the message queue.

The message arrives on the appropriate message queue.

Target side of the kernel

6. On the target side of the kernel, there are two potential *delivery models* for the interface between the run time and the target application.

   - The most common model is *push*, in which the kernel is responsible for initiating and managing delivery of the message to the target application. The push model typically does not require changing the target application to support MQSeries Adapter Offering.
   - In the *pull* model, the target application is responsible for managing the reception of the message. The pull model requires changing the target application to support MQSeries Adapter Offering. The target application must manage the kernel’s interface to the target application.

   Under the push model, note that on the target side, the kernel’s processes must be started by the user beforehand to get and deliver the message. See “Starting the kernel” on page 65.

   In the push model, the kernel gets the message off the message queue. It performs tracing if tracing is enabled. It continues to route the message by selecting the appropriate target adapter. In general, a different target adapter is required for each message type.

7. The kernel delivers the message to the appropriate target adapter. The target adapter performs the functionality that was built into it. A typical function is mapping elements from the integration messaging data format to elements in the target application format.

8. The target adapter sends the data to the target application in the *target application format* by using an application-specific interface developed outside of MQSeries Adapter Offering.
9. When the target adapter has delivered its message, a commit occurs. This removes the message from the queue.
10. If the source adapter has set a message control value to request an acknowledgment, the kernel delivers either an acknowledgment of message delivery or target adapter output to the source adapter by using the `sendResponse` method.
11. In case of error, the kernel puts the original message on the error queue. If the kernel cannot put the original message on the error queue, the commit does not occur.

Role of MQSeries or other messaging software

MQSeries Adapter Offering's communication messages are transported over message queues. Message queues are provided by messaging software such as MQSeries or the Java Messaging Service (JMS). Messages transported by MQSeries Adapter Offering use the following types of queues:

- *Receive queues*, in the terminology of MQSeries Adapter Offering. These are used as the main input queues to receive messages. There can be multiple receive queues per target application.
- *Error queues*, in the terminology of MQSeries Adapter Offering. These are used when a message that is obtained from a receive queue cannot be processed.
- As an option, *reply queues*. These are used with the `sendRequestResponse` method.

MQSeries Adapter Offering uses certain MQSeries capabilities, such as the following message types:

- Datagrams, used by the `sendMsg` method.
- Request, used by the `sendRequestResponse` method.
- Reply, used by the `sendRequestResponse` method and the `sendResponse` method.

MQSeries can optionally act as an application-specific interface.

See "Appendix B. Validated configurations" on page 81 for a list of validated configurations of MQSeries and MQSeries Adapter Offering. See "Software" on page 24 for a list of supported versions of MQSeries and other software.

Role of MQSeries Integrator

MQSeries Integrator can optionally be deployed with MQSeries Adapter Kernel. It can be used to meet several potential requirements for brokering:

- Complex routing, that is, routing based on the content of the message header or message body. The routing can change dynamically as the content of the message body changes. See "Routing and delivery" on page 12 for information about complex routing and simple routing.
- Data transformation, that is, changing to a different document type.
- Data mediation, that is, changing the content of the message body. For example, if the source application provides the value `each` in a field but the target application expects that field's value to be `ea`, data mediation replaces the provided value with the expected value.

You can use MQSeries Integrator to perform most of the routing in your site; you can also use less of the MQSeries Adapter Kernel's routing functionality.

See "Appendix B. Validated configurations" on page 81 for a list of validated configurations of MQSeries Integrator and MQSeries Adapter Offering. See "Software" on page 24 for a list of supported versions of MQSeries Integrator and other software.

How the kernel works

The following items are discussed in this section:

1. "Components of the kernel run time"
2. "Message and message format" on page 10
3. "Routing and delivery" on page 12
4. "Run-time flow" on page 12

Components of the kernel run time

When the adapters that you build, the custom code that you develop, and MQSeries Adapter Kernel run together, they provide the functionality of MQSeries Adapter Offering.

The major components of the kernel run time are as follows:

source adapter
: Software that is built for a specific application (typically by using MQSeries Adapter Builder) to convert data from that application into an integration messaging format (body data). Source adapters typically run on the same machine as the source application, either within the application's process or as a separate process. Examples of source data include files, C structures, and Java objects. An example of an integration messaging format is XML, typically following an industry standard such as OAG or RosettaNet.

message holder
: A container for metadata used by the kernel to encapsulate the

integration message and other control data used by the kernel. Examples of metadata include application identifiers (logical identifiers) of the source and target applications, the category of the message (for example, OAG), the type of the message (for instance, ″Purchase Order″), and the communications message (body data) being sent or received.

native adapter

Software used for sending and receiving message holder objects. When sending messages, the native adapter provides simple data routing and the ability to support one or more communication transport mechanisms. Simple data routing is based on metadata in the message holder object such as the category of message and type of message. Messages can be sent asynchronously or synchronously. Depending on the communication transport mechanism used, messages can be sent under single-phase transactional control. Transactional support is limited to the capabilities of the transport mechanism used. The message holder object is marshaled into the communications message format used by the transport mechanism. When a communications message is received, the native adapter unmarshals the message back into the message holder object.

adapter daemon

A process that instantiates adapter workers. After it is started, the adapter daemon remains active. For each target application, there can be one adapter daemon for each application receive queue.

adapter worker

A process that delivers each message to the appropriate target adapter. Each worker manages one native adapter. The adapter daemon creates and starts the workers.

The purpose of having multiple workers is to enable *multithreaded message delivery* to target adapters. Each worker, along with its native adapter, can handle one thread. If there is only one worker, then the delivery of messages to the target adapter, and hence to the target application, is single threaded.

In addition to managing a native adapter, the worker also performs the following tasks:

- It instantiates the trace client, if tracing is enabled.
- It instantiates the logon class that is appropriate for each target application.
- It selects the target adapter based on the body type and body category of the message.
- It sends the message to the selected target adapter.

- If it cannot perform a commit, it performs a rollback, sets a flag for all other workers under that adapter daemon, and shuts itself and its native adapter down. This signifies that the message has a problem. Shutting down all workers prevents other workers from reprocessing the same problem message with the same result.
- When it recognizes the flag set by another worker to shut down, it shuts itself and its native adapter down.

target adapter

Software that is built for a specific application (typically by using MQSeries Adapter Builder) to convert data from an integration messaging format (body data) to the native data types required by a target application. The target adapter invokes the necessary APIs on the target application to deliver the message. Target adapters run on the same machine as the application or application client.

configuration component

Data used for resolving logical identifiers into objects such as queue names. The configuration data can be specified either in a file or in the WebSphere Business-to-Business Integrator product's LDAP structure. The data controls the following aspects of the kernel's configuration:

- Marshaling and routing of messages
- Verifying installation
- Communication mode
- Tracing

See "The configuration file" on page 47 for a full description of the configuration file.

tracing component

Its purpose is to write trace messages. Most of the kernel's components use the tracing component. See "Tracing" on page 22 for an overview of tracing and the *Problem Determination Guide* for details about trace.

Message and message format

In MQSeries and MQSeries Adapter Offering, a *message* is a collection of data that is sent by one program and intended for another program. The format of the message at any given point of time depends on the message's location in the message flow at that particular time. MQSeries Adapter Kernel specifies three types of messages, as follows:

- *Integration message*—A message consisting of data from a source application converted into an application-neutral format such as XML for sending to a target application. The integration message is inserted into the message holder object as the message's body data. XML is a standard for the representation of data. When the format is XML, the format is defined by a *Document Type Definition* (DTD). A DTD is one or more files that contain a formal definition of a document—in this case, of the message body. The message body is not required to be in an application-neutral format. Although the format of the message body can be proprietary or otherwise specialized, this type of format is not recommended.

Business Object Documents (BODs) can be used by MQSeries Adapter Offering to define message bodies in its integration formatted messages. A BOD is a representation of a standard business process that flows within an organization or between organizations. Examples are "add purchase order", "show product availability", and "add sales order". BODs are defined in XML by the Open Applications Group (OAG). Use of BODs is recommended but is not required.

- *Message holder object*—An object containing the integration message and additional header metadata representing control values that are specific to MQSeries Adapter Kernel. The source adapter creates the message holder object, sets appropriate control information, and, if there is an integration message to be sent, sets the body data. Target adapters receive message holder objects, get the body data, and convert the body data to data specific to the target application. Source adapters and target adapters are created by using MQSeries Adapter Builder.
- *Communications message*—Any communications transport-specific information plus the message holder object, converted into a messaging format specific to the communications transport being used. Some communications transports support more than one messaging format. Typically, the kernel header metadata values combined with the communications message is considered to be application data by the communications transport. For more information, see "Appendix A. Communication modes" on page 75. Examples for MQSeries transport consist of the MQSeries-specific message header plus the marshaled message holder object. Specific MQSeries formats include the following:
  - The MQSeries message header that is added by MQSeries
  - If MQSeries Integrator is used, the version-specific message header:
    - The MQSeries Integrator version 1 message header, if MQSeries Integrator version 1.1 is used
    - The MQSeries Integrator version 2 message header, if MQSeries Integrator version 2 is used
  - The kernel-specific header metadata representing control values
  - The integration message (body data)

  See "Appendix C. Message headers" on page 83 for a list of relevant fields used in MQSeries Adapter Offering's message headers and their descriptions.

Routing and delivery

The kernel routes each message from the source adapter and delivers it to the appropriate target adapter. Routing is performed in two stages:

1. The source side of the kernel puts the message on the appropriate message queue.
2. The target side of the kernel gets the message from the message queue and invokes the appropriate target adapter.

Routing is determined by several things:

1. Message queues. On the most basic level, message queues must be configured to support MQSeries Adapter Offering's routing.
2. The message control values in the message. They include source logical identifier, destination logical identifier, respond to logical identifier, body category, body type, transaction identifier, message identifier, acknowledgment requested, and time stamps. See "Message control values" on page 14 for details. The destination logical identifier in the message can override the kernel's configuration file. Routing can change dynamically as these message control values in each message header changes. However, the content of the message body data (integration message) cannot determine routing.
3. The message control values in the kernel's configuration file. It can contain destination logical identifier, queue names, and associated target adapters. Determine and modify configuration by editing this file. See "The configuration file" on page 47 for additional information.
4. Optionally, MQSeries Integrator can be used to broker messages, including complex routing. The routing can change dynamically as the content of the message body changes. See "Role of MQSeries Integrator" on page 7. In contrast, by itself MQSeries Adapter Offering can perform only simple routing. Simple routing is based on a combination of message control values in the message and associated message control values in the configuration file. It is not based on the content of the message body.

The kernel can be requested to acknowledge message delivery. This is an application-level acknowledgment.

Run-time flow

This section discusses the run-time flow in detail—how the kernel sends, routes, traces, and delivers a message in a typical production environment. See Figure 2 on page 13 for a diagram of the run-time flow.

Before the kernel can process a message in production, you must prepare for production. See "Preparing for production" on page 45.

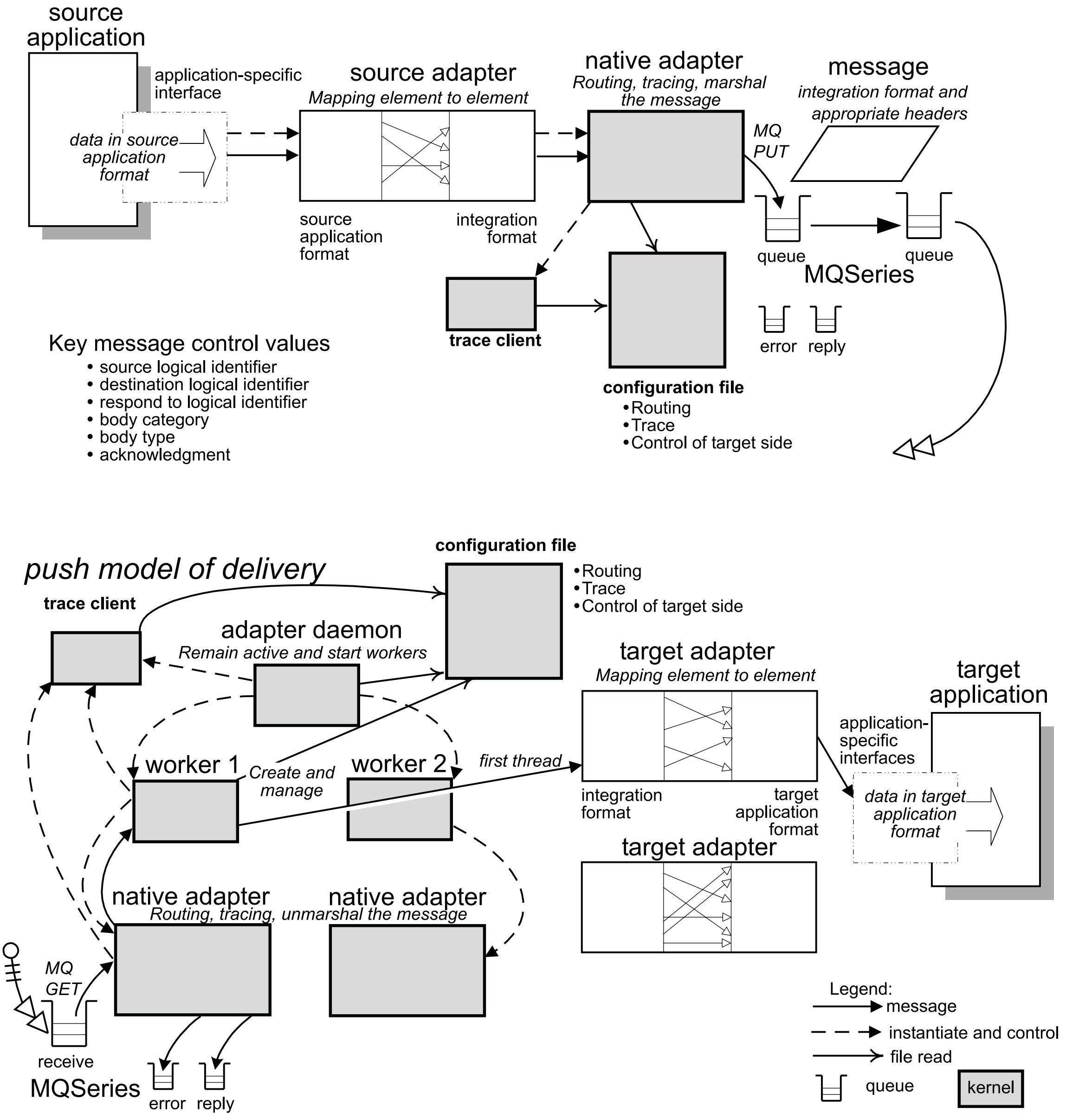


Figure 2. Marshal, send, route, and trace a message — overview.

Source side of the kernel

1. By using an application-specific interface, the source adapter acquires a message from the source application. Typically, the source adapter is invoked by the application-specific interface.

2. The source adapter performs the functionality that was built into it in MQSeries Adapter Builder. Typically, it transforms the data in the source application format into an application-neutral integration format (for the message body).

   As part of its functionality, the source adapter puts several message control values into the MQSeries Adapter Kernel header; it uses these values to envelope the message. The first five message control values determine marshaling and routing, and the last value determines acknowledgment.

   Message control values

   source logical identifier
   Logical identifier of the source application. It is always required in the message.

   destination logical identifier
   Logical identifier of the target application. If it is not present in the message, default values in the configuration file are used instead. In the configuration file, multiple destination logical identifiers can be used in place of values that are not contained in the message.

   respond to logical identifier
   The logical identifier of the application to which replies are to be sent if a reply is requested. It defaults to the source logical identifier in the message.

   body category
   Represents the message's application type—for example, OAG or RosettaNet. It is always required in the message.

   body type
   Represents the specific purpose of the message—for example, "add sales order" or "synchronize inventory". It is always required in the message.

   acknowledgment requested
   Determines whether the source application requests a reply. The reply can be either of the following forms:

   - Reply data from the target application
   - An OAG Confirm BOD message

     Note: The Confirm BOD message is predefined by the OAG. Its body category is `OAG` and its body type is `CONFIRM_BOD_003`. It can also contain data.

   This reply is an application-level acknowledgment.

When the kernel uses the `sendRequestResponse` method to send the message, only the first reply received by the `sendRequestResponse` method is used. If the original message is sent to multiple destinations and requests a reply (which is not recommended), only the first reply is sent back to the source application.

The default is no acknowledgment; thus, no reply is requested or sent.

3. The source adapter initializes the native adapter and passes it the following:
   - The logical identifier of the application under which the source adapter is running.
   - The message object, which contains the message control values and the message body data.
4. The native adapter looks in the configuration file to determine whether trace is enabled for that source logical identifier. If trace is enabled, the native adapter instantiates a trace client.
5. The trace client looks in the configuration file to determine which trace level to use and to obtain other values. The trace client uses the trace level to filter out trace messages. See "Tracing" on page 22 for an overview of tracing and the *Problem Determination Guide* for detailed information about tracing.
6. The native adapter looks in the message object for a destination logical identifier. If present, it is used.
   - If the destination logical identifier is not present, the native adapter looks up the default destination logical identifier in the configuration file, based on the source logical identifier, body category, and body type.
   - Based on the source logical identifier, the native adapter performs a two-stage lookup in the configuration file, in the following order:
     a. For specific body category and body type values that are associated with that source logical identifier.
     b. If no specific values are present, default body category and body type values are used. The default value for body category is DEFAULT. The default value for body type is DEFAULT.

   Note: The kernel uses this two-stage lookup each time it looks up values in the configuration file based on the body category and body type.
7. For each destination logical identifier determined in the previous step, the native adapter looks up the *communication mode*, based on the

destination logical identifier, body category, and body type. The following communication modes are supported:

MQPP The kernel transports messages by using MQSeries base services.

MQRFH1 The kernel transports messages by using MQSeries and brokers messages by using MQSeries Integrator version 1.1.

MQRFH2 The kernel transports messages by using MQSeries and brokers messages by using MQSeries Integrator version 2.

MQBD The kernel transports messages by using MQSeries base services but sends and receives body data only.

MQ The kernel transports messages by using MQSeries.

JMS The kernel transports messages by using the Java Messaging Service (JMS).

FILE The kernel puts messages into a file and gets them from a file. This mode is provided for diagnostic purposes only.

In each communication mode, the message structure is different. See “Message and message format” on page 10. For more information about communication modes, see “Appendix A. Communication modes” on page 75.

Note: If MQSeries Integrator is used, the final destination to which MQSeries Integrator sends the message must also be configured to receive messages by using the same communication mode as MQSeries Integrator.

8. Based on the communication mode, the native adapter instantiates a subclass within itself to handle the message. The subclass is called the *logical message service*. Each communication mode has a different logical message service subclass.

   The native adapter passes the destination logical identifiers, body category, and body type to the logical message service.
9. The logical message service subclass finds the parameters that it needs to send the message. For example, if the communication mode is MQPP, parameters include the format and the receive, reply, and error queue names. Based on the destination logical identifiers, body category, and body type that are passed to it, the logical message service performs a two-stage lookup in the configuration file:
   - For specific parameter values that are associated with the destination logical identifiers, body category, and body type

- If no specific parameter values are present, default parameter values

At this point, the logical message service has all the information that it needs to route and to marshal the message.

10. The logical message service performs the following tasks:
    - Marshals the message as appropriate for the communication mode and format. Each communication mode uses a default format if the format is not otherwise specified. For example, if the communication mode is MQRFH2, the logical message service creates appropriate headers and structures the message for transporting by using MQSeries and brokering by using MQSeries Integrator.
    - Sends the message. For example, if the communication mode is MQRFH2, it puts the message on the appropriate message queue.
11. There are two methods that can be used to send the message:
    - If the native adapter used the `sendMsg` method to send the message, the native adapter simply returns.
    - If the native adapter uses the `sendRequestResponse` method to send the message, the logical message service waits for the reply. The native adapter, by using the logical message service, monitors the reply queue for the *receive timeout period* that is set in the configuration file.

      The receive timeout period is based on the source application identifier, body category, and body type.
      - If an acknowledgment is received, the native adapter returns the message.
      - If an acknowledgment is not received within the receive timeout period, the native adapter does not return a message.
12. MQSeries or other messaging software transports the message according to how it was configured. Optionally, MQSeries Integrator performs brokering services. See “Role of MQSeries Integrator” on page 7.
13. When the source adapter is completely finished with the native adapter—that is, when it no longer needs the native adapter—it closes the native adapter to free resources.

Target side of the kernel

This is the push model of delivery in which the kernel is responsible for initiating and managing delivery of the message to the target application. See “delivery models” on page 102 for a short description of the models.

14. There is one adapter daemon for each target application’s receive queue. The adapter daemon is started.

    At its startup, it is given a name. Typically, each adapter daemon’s name is based on the destination logical identifier, that is, the logical identifier of the target application. For example, if the adapter daemon is servicing a target application whose destination logical identifier is `ABC`, the adapter daemon’s name is `ABCdaemon`.

The adapter daemon determines the destination logical identifier to use by looking in the configuration file for a dependency application identifier. If the daemon finds a dependency application identifier in the configuration file, it uses the dependency application identifier as the destination logical identifier. If the daemon does not find a dependency application identifier in the configuration file, it uses the name with which the daemon was started as the destination logical identifier.

Other parameters that can be passed to the adapter daemon at startup include body category and body type. The native adapter uses them later to determine the communication mode and the receive queue for incoming messages.

See "Starting the kernel" on page 65 for instructions for starting the adapter daemon.

15. At its startup, the adapter daemon looks in the configuration file to determine if trace is enabled for that adapter daemon name. If trace is enabled, the adapter daemon instantiates a trace client.

    See the *Problem Determination Guide* for details on trace.

16. At its startup, the adapter daemon instantiates the first worker and passes it the adapter daemon's name and the body category and body type.

17. The first worker looks in the configuration file to determine whether trace is enabled for that adapter daemon. If trace is enabled, the first worker instantiates a trace client, and the trace client looks in the configuration file to determine the trace level. See the *Problem Determination Guide* for a list of valid trace levels.

18. The first worker looks in the configuration file, based on the adapter daemon's application identifier, for the values that indicate the minimum number of workers that are to be instantiated and started.

    The first worker also looks up the *dependency application identifier.* The dependency application identifier is the name of the application that the worker services. It is later passed to the native adapter.

19. The adapter daemon queries the first worker for the minimum number of workers.

20. The adapter daemon starts the first worker, then instantiates and starts the minimum number of workers.

    The purpose of having multiple workers is to enable multithreaded message delivery to target adapters. Each worker, along with its native adapter, can handle one thread. If there is only one worker, then the delivery of messages to the target adapter, and hence to the target application, is single threaded.

    On AIX systems, two scheduling policies are available for threads: process-based scheduling and system-based scheduling. In process-based scheduling (the default), all user threads are mapped to a pool of operating-system kernel threads and run on a pool of virtual processors. In system-based scheduling, each user thread is mapped to a single OS kernel thread and runs on a single virtual processor. If you are using C source adapters that are called from C executable files on AIX, you must use system-based scheduling. For information on setting the thread-scheduling policy on AIX, see Step 14 on page 36.

Note that only process-based scheduling is supported on Windows systems, HP-UX, Solaris, and OS/400.

The other workers also perform the following steps that the first worker performs:

21. Each worker instantiates its associated native adapter. There is one native adapter associated with each worker. The dependency application identifier, body category, and body type are passed to the native adapter. The native adapter uses these three values to determine the communication mode and, by using the logical message service, the format and the receive queue for incoming messages. This process is similar to the process used for sending messages.
22. The native adapter gets the communications message from the receive queue under commit control and converts it into a message object. All the communication transport-specific headers except the native kernel header are removed from the message.
23. The native adapter passes the message object to the worker, which reads the body category, body type, and requested acknowledgment value from the message’s native kernel header.

    Based on the dependency application identifier, body category, and body type, the worker performs a two-stage lookup in the configuration file for the target command to invoke, in the following order:

    a. It searches for specific body category and body type values that are associated with that dependency logical identifier.
    b. If no specific values are present, it uses default body category and body type values. The default value for body category is DEFAULT. The default value for body type is DEFAULT.

    Together, the body category and body type values determine the *message type*, for example, “update sales order”. Based on the message type, the worker determines the appropriate target adapter command, a Java class that processes that particular message type. It instantiates that particular target adapter.
24. The worker passes the message object to the appropriate command.
25. Each command has three methods that the worker calls. They run in the following order:

    a. The *set message input* method, which sets the message to process into the target adapter.

   b. The *execute* method, which processes the message that was put into the target adapter by using the set message input method, then waits.
      1) The target adapter performs the functionality that was built into it by using MQSeries Adapter Builder. Typically, it transforms the data from the integration formatted message into the target application format. It maps element to element.
      2) The target adapter, by using an application-specific interface, sends the message to the target application.
      3) Depending on the nature of the target application, it sends or does not send a reply back to the target adapter.
   c. The *get message output* method, which gets the reply from the target adapter. The reply can indicate simply that the target application received the message; it can also contain data.

26. If the target adapter command does not throw an exception or if it does not have a Confirm BOD reply (which can indicate an error), the worker commits the received message from the receive queue by using the native adapter.
27. If an acknowledgment was requested, the worker calls the `sendResponse` method on the native adapter.
    - If the target adapter created a reply, it puts the respond to logical identifier of the original message into the destination logical identifier field of the reply message.
    - If the target adapter did not create a reply, then the worker creates a Confirm BOD reply message containing the completion status.
      - If there are no errors, the completion status is success.
      - If there are errors, the completion status is set to an error condition.
28. The reply is sent.
    a. The worker sends the reply message, if one has been created, to the native adapter.
    b. The native adapter puts the reply message into the reply queue.
    c. The native adapter sends the reply message, depending on the original message it received:
       - If it was an MQSeries request message, then the native adapter obtains the queue information for the reply from the MQSeries request message. This queue information overrides the destination logical identifier in the message.
       - If it was not an MQSeries request message, then the native adapter uses the `sendMsg` method to send the reply.
29. In case of exception or a Confirm BOD reply message with an error status, the worker logs an exception message into an exception file called `EpicSystemExceptionFile`*`nnnnnnnn`*`.log` that resides in the same directory as the adapter daemon. See "Exception messages" on page 68.

30. In case of exception or a Confirm BOD reply message with an error status, the worker directs the native adapter to put the original message on the error queue. The name of the error queue is obtained from the configuration file based on the dependency logical identifier, body category, and body type of the original message.

    Based on the dependency application identifier, body category, and body type, the worker performs a two-stage lookup in the configuration file in the following order:

    a. For specific body category and body type values that are associated with that dependency logical identifier.
    b. If no specific values are present, default body category and body type values are used. The default value for body category is DEFAULT. The default value for body type is DEFAULT.

    - If the native adapter is able to put the error message on the error queue, the native adapter is directed to commit the message from the receive queue.
    - If the native adapter is not able to put the error message on the error queue, the following occurs:
      a. The worker directs the native adapter to roll back, that is, not to commit.
      b. The worker sets a flag that directs all workers under that adapter daemon to shut down. This signifies that the message has a problem. Shutting down all workers prevents other workers from reprocessing the same problem message with the same result.
      c. If an out-of-memory error occurs, the exception is treated in the same way as all other exceptions except that the worker sets a flag for itself to stop when it has completed processing the current message. This makes more memory available for other workers.
31. When the native adapter notifies the worker that the work is done, the worker checks two flags:
    - Whether this worker is to stop. This can be caused by a Java out-of-memory condition.
    - Whether all workers are to stop, caused as described in the previous step.
32. If either flag is set, the worker stops. If neither flag is set, the worker processes the next message. The worker requests that the native adapter receive a message.
33. If a reply message is put onto the reply queue or if an error message is put onto the error queue, the following occurs:
    a. MQSeries or other messaging software delivers it back to the source side of the kernel.

b. If the source adapter called its native adapter's `sendRequestResponse` method, then the kernel retrieves the message from the reply queue and returns it to the source adapter. If the source adapter called the `sendMsg` method, then the kernel puts the message into the source application's receive queue.

Transactional capabilities

A *transaction* is a set of operations that must be executed as an indivisible unit of work. If all operations that constitute a transaction are successful, the transaction is *committed*; that is, all of the operations are performed. If one or more of the operations that constitute a transaction fail, the transaction is *rolled back*; that is, none of the operations are performed. By using MQSeries Adapter Kernel's transactional capabilities, a source adapter can perform a series of operations as a single unit, with the assurance that all operations succeed if the transaction is committed or that no operations occur if the transaction is rolled back.

Transactional capabilities can be built into adapters by using the MQSeries Adapter Builder or by using the `begin`, `rollback`, and `commit` methods on the `EpicNativeAdapter` class of the kernel's Java API. If a transactional method is called in an illegal context (for instance, calling the `commit` method without first having called the `begin` method, or calling the `begin` method within the scope of another transaction), the kernel disregards the call and issues a warning to trace. See "Chapter 5. Using MQSeries Adapter Kernel APIs" on page 71 for information on using the API.

Limitations

The following limitations are associated with the kernel's transactional capabilities:

- Transactions are not supported with the `sendRequestResponse` method.
- Nested transactions (that is, transactions that are called within other transactions) are not supported.
- Transactions are not supported by all communication modes; see "Appendix A. Communication modes" on page 75 for details.

Tracing

A trace message contains the state of processing a message at a certain point within the kernel. You can use trace messages to help diagnose problems with the kernel or with your adapters. The MQSeries Adapter Kernel *Problem Determination Guide* discusses using trace with the kernel.

Chapter 2. Planning to install the kernel

This chapter lists the prerequisites for and components of the MQSeries Adapter Kernel.

For latest details, see the MQSeries product family Web site at:

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

IBM reserves the right to update the information shown here. For the latest information regarding levels of supported software, refer to:

http://www.ibm.com/software/ts/mqseries/platforms/supported.html

Hardware

The following hardware is required for the MQSeries Adapter Kernel:

- An IBM PC machine (or compatible) running Windows NT 4.0, Service Pack 5, or Windows 2000, Service Pack 1.
- An IBM RS/6000 machine running AIX version 4.3.2 or 4.3.3.
- An HP Series 9000 machine running HP-UX version 11.0.
- A Sun SPARC or UltraSPARC machine running Solaris version 7 or 8.
- An IBM AS/400 or iSeries machine running OS/400 version 4.4 or 4.5.

Note: The installation of MQSeries Adapter Kernel on OS/400 requires a Windows system to interface with the AS/400 machine. See "Prerequisites for OS/400 installation" on page 25 for details.

MQSeries Adapter Kernel requires a minimum of approximately 25 MB of disk space for product code and data.

Ensure that sufficient disk space is available to hold the adapters. Their size is dependent on the size of the data structures, the complexity of mappings, and the custom code used. Some examples of different adapter sizes on Windows systems follow. Your site's adapters can require more or less disk space. Each example represents adapter source, compiled adapter code, API source, and API compiled code in MB or KB.

- Source adapter for adding a sales order: 1.89 MB
- Target adapter for synchronizing a customer record: 389 KB
- Target adapter for synchronizing an inventory record: 161 KB
- Target adapter for synchronizing an item: 249 KB
- Target adapter for synchronizing a sales order: 579 KB

In addition, allow a minimum of 20 MB for working space for the kernel and adapters. Working space requirements can vary based on a number of factors, such as the number and size of queues and the size of trace files.

Software

This section lists the software that is supported for use with MQSeries Adapter Kernel. Supported levels are shown. See "Appendix B. Validated configurations" on page 81. Note that C compilers are required on development systems but not on production systems. The C compilers listed here were successfully tested with MQSeries Adapter Kernel; other C compilers can possibly work correctly with the kernel but are not officially supported.

For Windows systems:

- Microsoft Windows NT version 4.0, Service Pack 5; or Microsoft Windows 2000, Service Pack 1. To determine the version and service pack of Microsoft Windows, open Windows Explorer, then click **Help > About Windows**.
- Microsoft Visual C++ 6.0 Compiler.
- MQSeries version 5.1, CSD 5, including MQSeries Java support.
- Java Development Kit version 1.1.8 or 1.2.2.

For AIX:

- AIX operating system version 4.3.2 or 4.3.3.
- X Window System (X11R5 or higher). This is required for installation but not at run time.
- IBM C Set++ for AIX version 3.1.3.
- MQSeries version 5.1, CSD 5, including MQSeries Java support.
- Java Development Kit version 1.1.8 or 1.2.2.

For HP-UX:

- HP-UX operating system version 11.0.
- X Window System (X11R5 or higher). This is required for installation but not at run time.
- HP-UX C/ANSI C Compiler. See the `readme.txt` file for details.
- MQSeries version 5.1, CSD 5, including MQSeries Java support.
- Java Development Kit version 1.1.8 or 1.2.2.

For Solaris:

- Solaris operating environment version 7 or 8.
- X Window System (X11R5 or higher). This is required for installation but not at run time.

- Sun Workshop Compilers C/C++. See the `readme.txt` file for details.
- MQSeries version 5.1, CSD 5, including MQSeries Java support.
- Java Development Kit version 1.1.8 or 1.2.2.

For OS/400:

- OS/400 operating system version 4.4 or 4.5, including the following programs:
  - Java Toolkit and Java Developer Kit version 1.1.8 or higher. The Java Toolkit and Java Developer Kit are shipped as licensed program number 5769–JV1. See "Prerequisites for OS/400 installation" for additional details about versions of the Java Developer Kit required for installing MQSeries Adapter Kernel on an AS/400 system.
  - The Host Servers option, which is shipped as licensed program number 5769–SS1, option 12.
  - Qshell Interpreter, which is shipped as licensed program number 5769–SS1, option 30.
  - TCP/IP, which is shipped as licensed program number 5769–TC1.
  - Integrated Language Environment C for AS/400, which is shipped as program number 5769–CX2.
- MQSeries version 5.1, CSD 4, including MQSeries Java support.

See "Prerequisites for OS/400 installation" for additional requirements for installing MQSeries Adapter Kernel on OS/400.

The following products are supported with MQSeries Adapter Kernel:

- MQSeries version 5.1, including MQSeries Java support.

  Note: If MQSeries is not used, another messaging software product such as the Java Messaging Service (JMS) must be used.
- MQSeries Integrator version 1.1
- MQSeries Integrator version 2

See "Appendix B. Validated configurations" on page 81 for a list of validated MQSeries Adapter Kernel, MQSeries, and MQSeries Integrator configurations.

Prerequisites for OS/400 installation

This section describes the prerequisites for installing MQSeries Adapter Kernel on an AS/400 or iSeries system. See Step 2b on page 33 for detailed instructions on installing MQSeries Adapter Kernel on an AS/400 system. Because AS/400 terminals do not natively support Java graphics, a graphics-enabled workstation such as a Windows system is required to run the kernel's Java-based GUI installation program. The workstation can interface with the AS/400 system in one of the following ways:

- Through remote AWT, in which all graphics are processed on the AS/400 system and displayed on the workstation. This is described in more detail in “Using remote AWT”.
- As an attached client, in which the workstation processes and displays the graphics. This is described in more detail in “Using an attached client” on page 27.

This section assumes that you are using a Windows system as the graphics-enabled workstation.

Using remote AWT

When remote AWT is used, Java graphics processing is done on the AS/400 system, and graphics are displayed on a client workstation that is attached to the AS/400 system. This section describes the requirements that must be met to install MQSeries Adapter Kernel on an AS/400 system by using remote AWT.

The following programs must be installed with OS/400:

- Java Toolkit and Java Developer Kit version 1.2.2 or higher. The Java Toolkit and Java Developer Kit are shipped as licensed program number 5769–JV1. Remote AWT capabilities on OS/400 are provided by the Java Developer Kit.
- TCP/IP, which is shipped as licensed program number 5769–TC1. For more information about TCP/IP, see the *AS/400 TCP/IP Fastpath Setup Information* and *AS/400 TCP/IP Configuration* documents, which are available in the AS/400 library at http://www.ibm.com/servers/eserver/iseries/library/.

Requirements for the workstation are as follows:

- An IBM PC machine (or compatible) running Windows 95, Windows 98, Windows NT, or Windows 2000.
- A TCP/IP connection to the AS/400 system.
- JDK 1.2.2 or higher.

To set up and start remote AWT, perform the following steps:

1. Ensure that JDK 1.2.2 or higher is installed on the workstation.
2. Ensure that a TCP/IP connection exists between the AS/400 system and the workstation.
3. Copy the `RAWTGui.jar` file from the `/QIBM/ProdData/Java400/jdk12` directory on the AS/400 system to a directory on the workstation.
4. On the workstation, change to the directory where you copied the `RAWTGui.jar` file and start remote AWT by entering the following command:

   ```
   java -jar RAWTGui.jar
   ```

Note: Because of the resource-intensive nature of processing Java graphics on an AS/400 system, using remote AWT can possibly take much longer than using an attached client to install MQSeries Adapter Kernel.

For more information on remote AWT, see the AS/400 library at http://www.ibm.com/servers/eserver/iseries/library/.

Using an attached client

When an attached client is used to install MQSeries Adapter Kernel on an AS/400 system, Java graphics processing is done on the client workstation, not on the AS/400 system. This section describes the requirements that must be met to install MQSeries Adapter Kernel on an AS/400 system by using an attached client.

The following programs must be installed with OS/400:

- Java Toolkit and Java Developer Kit version 1.1.8 or higher. The Java Toolkit and Java Developer Kit are shipped as licensed program number 5769–JV1.
- The Host Servers option, which is shipped as licensed program number 5769–SS1, option 12.
- TCP/IP, which is shipped as licensed program number 5769–TC1.

Requirements for the workstation are as follows:

- An IBM PC machine (or compatible) running Windows NT 4.0, Service Pack 5, or Windows 2000, Service Pack 1.
- A TCP/IP connection to the AS/400 system.
- JDK 1.1.8 or higher.

Components of the kernel

After installation, MQSeries Adapter Kernel resides in its root directory. It contains subdirectories that in turn can contain other directories. The root and its subdirectories are listed, along with a summary of the files that are most relevant to installation and configuration.

root The default name is `C:\Program Files\MQAK` on Windows systems, `/MQAK` on UNIX, and `/QIBM/ProdData/mqak` on OS/400. It contains the following:

- All other MQSeries Adapter Kernel directories.
- The `aqmsetenv.bat` (Windows systems) or `aqmsetenv.sh` (UNIX) file, which changes system environment variables after installation, if desired.
- The `readme.txt` file.
- The `aqmuninstall.bat` (Windows systems) or `aqmuninstall.sh` (UNIX) file.

bin Contains the following:

- Class libraries and shared libraries.
- Adapters that are provided as part of the kernel, for verification use only.
- The aqmversion.bat (Windows systems) or aqmversion.sh (UNIX and OS/400) file, a script that is run to display the version number of the kernel.
- The aqmcrtmsg.bat (Windows systems) or aqmcrtmsg.sh (UNIX and OS/400) file, a script that is run to create an XML file used to validate the configuration file before it is put into production.
- The aqmsndmsg.bat (Windows systems) or aqmsndmsg.sh (UNIX and OS/400) file, a script that is run to validate the configuration file before it is put into production.
- The aqmstrad.bat (Windows systems) or aqmstrad.sh (UNIX and OS/400) file, a script that is run to start the adapter daemon.
- The aqmstrtd.bat (Windows systems) or aqmstrtd.sh (UNIX and OS/400) file, a script that is run to start the trace server.

documentation
Contains the product documentation, including the Information Center.

runtimefiles
Contains kernel run-time files.

samples

Contains samples of adapters and associated configuration and utility files. You can experiment with and learn from them.

Note: The kernel is intended to be used with adapters built by using the MQSeries Adapter Builder. The kernel is not intended to be used by calls to the kernel APIs from custom code alone. The adapter samples are provided only as an aid to understanding how the kernel functions and in diagnostics.

- Adapter samples.
- The kernel's setup file, aqmsetup, with values that support the samples of adapters. See "The setup file" on page 47 for a discussion of this file.

- The kernel's configuration file, `aqmconfig.xml`, with values that support the samples of adapters, including sample trace values. See "The configuration file" on page 47 for a discussion of this file.

`toolkit`

Contains a software development toolkit (SDK) consisting of the following:

- Header files.
- Library files used during compilation under Windows systems.

`uninstall`

Contains files used to uninstall the kernel.

`verification`

Contains the following files that support verification of the installation of the kernel:

- The `aqmverifyinstall.bat` (Windows systems) or `aqmverifyinstall.sh` (UNIX and OS/400) file, a script that is run to verify installation of the kernel on one computer.
- The `aqmcreateq.bat` (Windows systems) or `aqmcreateq.sh` (UNIX and OS/400) file, a script that creates MQSeries queues for verification. See "Creating MQSeries queues" on page 69.
- The `aqmconfig.xml` file. See "The configuration file" on page 47 for a discussion of this file.
- The `aqmsetup` file. See "The setup file" on page 47 for a discussion of this file.
- The `aqminstalltest.xml` file.

Chapter 3. Installing the kernel

To make the kernel ready to use, perform the following general steps:

Step 1. Read "Chapter 1. About MQSeries Adapter Offering" on page 1.

Step 2. Prepare for installation. See "Preparing for installation" for details.

Step 3. Install the kernel. See "Installing the kernel" on page 32 for details.

Step 4. Verify the installation. See "Verifying the installation" on page 38 for details.

Step 5. Configure the kernel. See "Configuring the kernel" on page 46 for details.

Step 6. Configure messaging software and optional software. See "Configuring MQSeries and MQSeries Integrator" on page 64 for details.

Step 7. Build your adapters by using MQSeries Adapter Builder, then test and deploy them.

Step 8. Start the kernel. See "Starting the kernel" on page 65 for details.

Preparing for installation

You must have administrator or root authority to install MQSeries Adapter Kernel. You must have permission to create and access files in the location where you install MQSeries Adapter Kernel and the location where you put the two kernel configuration files. You must have the current directory in your path. Ensure that all user IDs that run the kernel have read, write, and execute permission.

You must have authority to perform MQSeries operations such as creating queue managers and creating and accessing queues. These operations are performed in different ways on different platforms. Refer to the *MQSeries Administration Guide* for your platform for more information.

The user identifier that starts the kernel's processes must be in the mqm group. There are two kinds of kernel processes:

- Adapter daemon, one for each target application served by the computer
- Trace server (optional)

Note that the source adapter is run in the source application's process. Any daemon or server that contains the source adapter needs to be started.

You must install and configure the kernel to run the adapters that you have built. However, you do not have to install the kernel to install the MQSeries Adapter Builder or to use it to build your adapters.

Installing the kernel

To install MQSeries Adapter Kernel on a Windows system (Windows NT or Windows 2000), on a UNIX platform (AIX, HP-UX, or Solaris), or on OS/400, perform the following steps:

__ Step 1. Read the `readme.txt` file on the CD-ROM or local area network. It possibly contains important information that became available after this book was completed. It is located in the root installation directory.

__ Step 2. Visit the MQSeries Web site at http://www.ibm.com/software/ts/mqseries/. It possibly contains important information that became available after this book was published, possibly including a new edition of this book.

__ Step 3. If you are upgrading from a previous version of MQSeries Adapter Kernel, see "Upgrading the kernel" on page 43 for instructions.

__ Step 4. Ensure that the hardware and software prerequisites are met. See "Hardware" on page 23 and "Software" on page 24 for details. MQSeries must be installed and running before you can verify installation of MQSeries Adapter Kernel. Ensure that MQSeries Java support is installed and configured.

__ Step 5. Ensure that you have prepared for installation, for example, that you are authorized to install and verify the kernel. See "Preparing for installation" on page 31.

__ Step 6. Ensure that JDK 1.1.8 or higher is installed on the machine on which MQSeries Adapter Kernel is to be installed. The installation program checks the version of the JDK before beginning installation. If the JDK or JRE is version 1.1.6 or lower, the installation program fails. If JDK 1.1.6 or JRE 1.1.6 is installed on the machine, uninstall it and install JDK 1.1.8 or higher before proceeding.

__ Step 7. To start the installation program, perform the following operating system-specific steps:

On Windows systems:

a. Start the installation program as follows:

- If you are installing from a local area network, change to the directory that contains the MQSeries Adapter Kernel installation files and run the `install.bat` file.

- If you are installing from CD-ROM, insert the MQSeries Adapter Kernel CD-ROM into the CD-ROM drive. If autorun is enabled, the installation program starts automatically; if autorun is not enabled, run the `install.bat` file in the root directory of the CD-ROM to start the installation program.

Note: On Windows systems, you do not have to copy the `install.bat` file to another location before you run it. During the installation process, you are asked to choose where to install MQSeries Adapter Kernel.

b. Follow the prompts provided by the installation program. Note that if you choose to install MQSeries Adapter Kernel in a location other than the default (on Windows systems, `C:\Program Files\MQAK`), you must specify the installation directory as a fully qualified path name, not as a relative path name.

On UNIX:

a. Start the installation program as follows:
   - If you are installing from a local area network, change to the directory that contains the MQSeries Adapter Kernel installation files and run the `install.sh` script.
   - If you are installing from CD-ROM, insert the MQSeries Adapter Kernel CD-ROM into the CD-ROM drive and, if necessary, mount the CD-ROM drive according to your operating system documentation. Run the `install.sh` script in the root directory of the CD-ROM.

b. Follow the prompts provided by the installation program. Note that if you choose to install MQSeries Adapter Kernel in a location other than the default (on UNIX, `/MQAK`), you must specify the installation directory as a fully qualified path name, not as a relative path name.

On OS/400:

a. Ensure that all prerequisites listed in “Hardware” on page 23, “OS/400 software prerequisites” on page 25, and “Prerequisites for OS/400 installation” on page 25 are met. Note that installing MQSeries Adapter Kernel on OS/400 uses an InstallShield-based program that requires the use of a workstation interfacing with the AS/400 system; see “Prerequisites for OS/400 installation” on page 25 for details.

b. Depending on whether you are using remote AWT or an attached client workstation to perform the installation, perform the following steps:

- If you are using remote AWT to perform the installation, perform the following steps:
  1) Ensure that remote AWT is set up and running. See "Using remote AWT" on page 26 for details.
  2) Ensure that the `installAS400.jar` file is accessible to the AS/400 system. The file must be either in the integrated file system (IFS) or on a device attached to the AS/400 system. If the file is on an attached device, use the Create Link (**CRTLINK**) command to create a symbolic link to the file.
  3) To improve the performance of the installation process, run the Create Java Program (**CRTJVAPGM**) command against the `installAS400.jar` file.
  4) Run the Run Java (**RUNJVA**) command as follows, where *n.n.n.n* represents the TCP/IP address of the workstation that is running remote AWT:

     ```
     RUNJVA CLASS(run)
     CLASSPATH('/installAS400.jar')
     PROP((os400.class.path.rawt 1) (RmtAwtServer 'n.n.n.n')
     (java.version 1.2))
     ```
- If you are using an attached client workstation to perform the installation, perform the following steps:
  1) Ensure that the requirements specified in "Using an attached client" on page 27 are met.
  2) Ensure that the Host Servers option is installed and running on the AS/400 machine. You can start Host Servers by using the Start Host Servers (**STRHOSTSVR**) command at a Control Language (CL) prompt.
  3) Ensure that TCP/IP is installed and running on the AS/400 machine. You can start TCP/IP by using the Start TCP/IP (**STRTCP**) command at a CL prompt.
  4) On the workstation, open a command prompt and change to the `AS400` directory of the MQSeries Adapter Kernel installation media (either local area network or CD-ROM).
  5) If the workstation is running JDK 1.1.8, enter the following command:

     ```
     jre -cp installAS400.jar; run -os400
     ```

     If the workstation is running JDK 1.2.2 or higher, enter the following command:

     ```
     java -classpath installAS400.jar; run -os400
     ```

c. The installation program begins and displays the **Signon to AS/400** panel. Enter the TCP/IP address of the AS/400 machine in the **System:** field and your user ID and password in the corresponding fields. Ignore the **Default User** checkbox. Click **Next**.

d. Follow the prompts provided by the installation program. Depending on the speed of your network and machines, the installation process can take up to one hour to complete. A progress bar displayed on the workstation indicates the status of the installation.

   Note that on OS/400, MQSeries Adapter Kernel is always installed in the `/QIBM/ProdData/mqak` directory in the root of the integrated file system (IFS).

e. Set the CLASSPATH, PATH, and QIBM_MULTI_THREADED environment variables, as follows:

   - Add the `/QIBM/ProdData/mqak/bin` directory to the CLASSPATH environment variable.
   - Add the `/QIBM/ProdData/mqak/bin` directory to the PATH environment variable.
   - Set the QIBM_MULTI_THREADED environment variable to `Y`.

f. Add the library MQAK to the QSYS.LIB library list.

__ Step 8. Kernel installation is complete. As installed, the kernel is configured to support verification, not to support production at your particular site. Verify the installation by performing the steps listed in “Verifying the installation” on page 38. After you have verified the installation, return to the next step in this installation procedure to set environment variables and move several configuration files to support production at your site.

__ Step 9. Decide where you want to put two configuration files, `aqmsetup` and `aqmconfig.xml`, that are used in production. For more information on these files, see “Configuring the kernel” on page 46.

CAUTION:
If you do not create your own configuration files but instead use the configuration files that are provided in the `samples` directory for production, installing a new version of the kernel overwrites them and destroys your production configuration.

__ Step 10. Create a directory for the two configuration files. They do not need to be located in the same directory, but this is recommended for simplicity. If you locate them outside the directory where you installed MQSeries Adapter Kernel, this leaves fewer directories if the kernel is uninstalled at a later time.

The uninstall process leaves directories that contain anything other than the original MQSeries Adapter Kernel files.

__ Step 11. Copy the aqmsetup and aqmconfig.xml files from the samples directory to your desired location. You can put them on a network drive or other central location that is accessible by many computers to make updating them and backing them up easier.

If you rename the aqmconfig.xml file, the kernel does not operate correctly. You can rename the aqmsetup file, provided that you set an environment variable to point correctly to it in a subsequent step.

__ Step 12. Using a text editor, edit the aqmsetup file to point to the desired directory of the aqmconfig.xml file. Use a fully qualified path name (not a relative path name) as the location of the directory. Do not include the file name itself in the path. An example follows:

```
# Location of configuration file aqmconfig.xml.
AQMCONFIG=C:\Program Files\MQAK\Data\
```

Even if your desired location for the aqmconfig.xml file is the same directory where the aqmsetup file resides, you must enter the fully qualified path name here. Save and close the aqmsetup file.

__ Step 13. Set the AQMSETUPFILE environment variable to point to the location of the aqmsetup file (for instance, C:\Program Files\MQAK\Data\aqmsetup on Windows systems, /MQAK/data/aqmsetup on UNIX, or /home/*username*/aqmsetup on OS/400). Note that on OS/400, the aqmsetup file must always be located in the current user's home IFS directory (that is, /home/*username*).

If the kernel is installed on a network drive, perform this step for each computer that accesses it.

__ Step 14. If you are using AIX and plan to use native C-language source adapters that are called from a C program, set the AIXTHREAD_SCOPE environment variable to the value S. To set this environment variable in the Bourne shell or Korn shell, enter the following command:

```
export AIXTHREAD_SCOPE=S
```

To set this environment variable in the C shell, enter the following command:

```
setenv AIXTHREAD_SCOPE S
```

To have the AIXTHREAD_SCOPE variable set automatically when you log in to AIX, add this command to your `.profile` file (if you use Bourne shell or Korn shell) or `.cshrc` file (if you use C shell).

See Step 20 on page 18 for additional information about scheduling policies.

__ Step 15. If necessary, set the THREADS_FLAG environment variable. You must set this variable only if *all* of the following conditions are true:

- Solaris is the operating system being used.
- The version of the Java Development Kit (JDK) being used is 1.2.2.
- MQSeries is being used to transport messages.
- Your source and target adapters are written in C.

If all of these conditions are true, set the THREADS_FLAG environment variable to `native`. To set this environment variable in the Bourne shell or Korn shell, enter the following command:

```
export THREADS_FLAG=native
```

To set this environment variable in the C shell, enter the following command:

```
setenv THREADS_FLAG native
```

To have the THREADS_FLAG variable set automatically when you log in to Solaris, add this command to your `.profile` file (if you use Bourne shell or Korn shell) or `.cshrc` file (if you use C shell).

__ Step 16. Prepare for production. See “Preparing for production” on page 45.

__ Step 17. Edit the configuration file. See “Configuring the kernel” on page 46.

__ Step 18. Configure MQSeries and optional software. See “Configuring MQSeries and MQSeries Integrator” on page 64.

__ Step 19. For production systems, take into account “Performance recommendations” on page 64.

__ Step 20. Start the kernel. See “Starting the kernel” on page 65.

__ Step 21. Set up a kernel maintenance plan. See “Maintaining the kernel” on page 67.

Install the kernel on other computers as required.

Verifying the installation

After you install the kernel, verify that it was installed correctly by running a verification script. It sends a test message from a source application by using a source adapter, then by using the kernel to MQSeries. It then uses the kernel to receive the message from MQSeries and then invoke a target adapter. All of these processes are run on a single computer.

In this verification, the source application is an MQSeries queue named TEST1. The target application is another MQSeries queue named TEST2.

The verification performs the following tasks:

- Verifies that the kernel, with the supplied source adapter and the target adapter, marshaled and routed the test message correctly, using MQSeries as the messaging software, end-to-end within the computer.
- Verifies the `aqmconfig.xml` and `aqmsetup` files that are provided at installation. They determine the kernel configuration. See "Configuring the kernel" on page 46 for information on these files.

You can validate the configuration file before putting it into production. See "Validating the configuration file" on page 60.

The installation verification scripts that are provided with MQSeries Adapter Kernel assume that MQSeries is installed and configured on the machine where the scripts are to be run. If you are using messaging software other than MQSeries, you can edit the installation verification scripts to support your messaging software as follows:

1. Change to the kernel installation's `verification` directory.
2. Open the `aqmconfig.xml` file in a text editor and change the line `<epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>` to `<epicmqppqueuemgr>`*`queue_manager_name`*`</epicmqppqueuemgr>`, where *queue_manager_name* is the name of your queue manager.
3. Edit the `aqmverifyinstall` file as follows:
   - If you are performing installation verification on a Windows system, open the `aqmverifyinstall.bat` file in a text editor and change the line `aqmcreateq TEST2` to `aqmcreateq TEST2` *`queue_manager_name`*, where *queue_manager_name* is the name of your queue manager.
   - If you are performing installation verification on UNIX or OS/400, open the `aqmverifyinstall.sh` file in a text editor and change the line `aqmcreateq.sh TEST2` to `aqmcreateq.sh TEST2` *`queue_manager_name`*, where *queue_manager_name* is the name of your queue manager.

This verification uses some components, such as a target adapter, `com.ibm.epic.adapters.eak.test.InstallVerificationTest`, that are not part of the kernel. They are supplied with the kernel only for the purpose of verifying installation.

When verification is complete, the verification adapter daemon is stopped.

Tracing is not enabled during verification.

Verification procedure

__ Step 1. Verification creates and uses three MQSeries queues. If these queues have messages in them before you perform verification, verification fails. Clear the messages in the following queues:
- TEST2AIQ
- TEST2AEQ
- TEST2RPL

__ Step 2. Check that you have prepared for installation, for example, that you are authorized to install and verify the kernel. See "Preparing for installation" on page 31.

__ Step 3. Start the verification as follows:
- On Windows systems, double-click the `aqmverifyinstall.bat` file in the `verification` directory. Alternately, open a command prompt window, change to the `verification` directory, and run `aqmverifyinstall.bat`.
- On UNIX, open a terminal, change to the `verification` directory, and run the `aqmverifyinstall.sh` file.
- On OS/400, perform the following steps:
  a. Start a **qsh** session by using the **STRQSH** command
  b. Copy the `/QIBM/ProdData/mqak/verification/aqmsetup` file to your home directory (`/home/`*`username`*).
  c. Change to the `/QIBM/ProdData/mqak/verification` directory.
  d. Run the `aqmverifyinstall.sh` file.

The `aqmverifyinstall` file contains comments about how it functions.

__ Step 4. The message `Installation Verification Test completed successfully` indicates success. Close the verification window, if necessary.

__ Step 5. In case of failure, examine the verification window and the log file, `EpicSystemExceptionFile`*`nnnnnnnn`*`.log`, to determine the error.

__ Step 6. See "Common verification problems" on page 40 for common problems that can be encountered during verification and for potential responses.

__ Step 7. If desired, perform optional verification. See "Optional verification" on page 42 for details.

__ Step 8. Return to the installation procedure and configure the kernel to support operation in your particular site. Go to Step 9 on page 35.

Common verification problems

This section lists common problems that can be found during verification, along with potential solutions. Important information in the exception messages is highlighted in **bold**.

Problem: The aqmsetup file was not found.

Response: Make sure the AQMSETUPFILE environment variable is set to the location of the aqmsetup file in the verification directory.

Exception message:

```
com.ibm.epic.adapters.eak.nativeadapter.EpicNativeAdapter::main: caught
throwable with message <AQM0002: com.ibm.epic.adapters.eak.common.
AdapterDirectory::getProperties():
Received exception <com.ibm.epic.adapters.eak.common.AdapterException>
Message information: <AQM0002: com.ibm.epic.adapters.eak.common.
AdapterCfg::readConfig(String):
Received exception <java.io.FileNotFoundException> Message information:
<C:\aqmsetup> Additional program information <>.>
Additional program information <Error Reading Configuration File
[File or Keys in file may not exist]>.>
```

Problem: The aqmconfig.xml file was not found.

Response: Edit the aqmsetup file in the verification directory and make sure the AQMCONFIG= entry points to the verification directory. Use a fully qualified path name. Also ensure the aqmconfig.xml file is in the verification directory.

Exception message:

```
com.ibm.epic.adapters.eak.common.AdapterException: MessageID <AQM0002>
<AQM0002: com.ibm.epic.adapters.eak.common.AdapterDirectory::
getProperties(): Received exception
<java.io.FileNotFoundException> Message information:
<AQMCONFIG.xml> Additional program information <>.>
```

Problem: The queue on which to put the message did not exist.

Response: Use MQSeries to ensure that the queue named in the exception message (TEST2AIQ when installation is being verified) exists and can accept messages. See “Creating MQSeries queues” on page 69.

Exception message:

```
com.ibm.epic.adapters.eak.nativeadapter.EpicNativeAdapter::main: caught
throwable with message
<AQM0107: com.ibm.epic.adapters.eak.nativeadapter.LMSMQbase::
createMQOutputQueue(String):
Received MQException creating queue, QManager name <DEFAULT>
Queue name <TEST2AIQ>:
completion code <2> reason code <2085>.>
```

Problem: The target adapter was not found.

Response: Ensure that the target adapter specified in the message exists: `com.ibm.epic.adapters.eak.test.InstallVerificationTest`. The CLASSPATH environment variable must include the kernel's `bin` directory.

Exception message:

```
com.ibm.epic.adapters.eak.adapterdaemon.EpicAdapterWorker::sendException
(Throwable, String):Thread-2:
Message <<TEST2> <2000.05.18.09.41.43.781> <<Processing Messages.>
<com.ibm.epic.adapters.eak.common.AdapterException: MessageID <AQM0002>
<AQM0002: com.ibm.epic.adapters.eak.adapterdaemon.EpicAdapterWorker:
:instantiateClass(String, Class[], Object[]): Received exception
<java.lang.ClassNotFoundException> Message information:
<com.ibm.epic.adapters.eak.test.InstallVerificationTest>
Additional program information <[Cannot obtain Class for class name
<com.ibm.epic.adapters.eak.test.InstallVerificationTest>]>.>>>>
```

Problem: An adapter was not found to load for delivery of the message. The destination logical identifier does not have an entry in the `aqmconfig.xml` file for the body type and body category specified in the message on the queue.

Response: During verification, the most likely cause of this exception message is the existence of messages on a queue named TEST2AIQ prior to verification. Clear all messages from the TEST2AIQ queue and retry verification. The only entry for a command class name for application TEST2 in the `aqmconfig.xml` file in the `verification` directory is for a body type of TESTBOD and a body category of OAG.

Exception message:

```
com.ibm.epic.adapters.eak.adapterdaemon.EpicAdapterWorker::sendException
(Throwable, String):Thread-2: Message <<TEST2> <2000.05.18.10.28.43.105>
<<Processing Messages.> <com.ibm.epic.adapters.eak.common.
AdapterException:
MessageID <AQM0401> <AQM0401: com.ibm.epic.adapters.eak.
adapterdaemon.EpicAdapterWorker::processMessage(EpicMessage):
Cannot obtain Command class name to load for a received message.>>>>
```

Problem: The verification queue manager was not started.

Response: Ensure that the default MQSeries queue manager was started successfully.

Exception message:

```
com.ibm.epic.adapters.eak.common.AdapterException: Message ID <AQM0104>
<AQM0104: com.ibm.epic.adapters.eak.nativeAdapter.queueCollection::
constructor(String,String,boolean,String,String,int):
```

```
Received MQException creating QManager connection for
QManager name <QMGRNAME>
MQ Message information: completion code <2> reason code <2059>.>
```

Problem: A general MQSeries error occurred.

Response: Ensure that MQSeries is installed and configured correctly and is running on the machine. Examine the MQException reason code and use the *MQSeries Messages* document to determine the cause of the reason code.

Exception message:

```
Received MQException "ACTION ATTEMPTED." Message information:
completion code <completion_code> reason code <reason_code>
```

Optional verification

After you verify that the kernel was installed correctly on the first computer, you can optionally perform the following steps:

1. Verify that the kernel is installed correctly on a second computer, using the same verification.
2. Verify that you can send a test message from a source adapter on one computer to a target adapter on another computer. Manually configure and perform this verification. If you choose to develop this verification by modifying the original verification files that are provided with the kernel, retain a copy of the original verification files for backup purposes.

Removing the kernel

There are several ways to remove the kernel.

- On Windows systems, use one of the following methods:
  - From the Start menu, click **Programs > IBM MQSeries Adapter Kernel > Uninstall MQSeries Adapter Kernel**.
  - Use the Add/Remove Programs utility in the Control Panel.
  - Double-click the `aqmuninstall.bat` file in the kernel's root directory.
- On UNIX, change to the kernel's root directory and enter the following command:

  ```
  aqmuninstall.sh
  ```
- On OS/400, you can use either remote AWT directly on the AS/400 system or an attached client to uninstall the kernel.
  - If you are using remote AWT to perform the installation, perform the following steps:
    1. Ensure that remote AWT is set up and running. See "Using remote AWT" on page 26 for details.

2. To improve the performance of the uninstallation process, run the Create Java Program (**CRTJVAPGM**) command against the `/QIBM/ProdData/mqak/uninstall/uninstall.jar` file.
3. Run the Run Java (**RUNJVA**) command as follows, where *n.n.n.n* represents the TCP/IP address of the workstation that is running remote AWT:

```
RUNJVA CLASS(run)
CLASSPATH('/QIBM/ProdData/mqak/uninstall/uninstall.jar')
PROP((os400.class.path.rawt 1) (RmtAwtServer 'n.n.n.n')
(java.version 1.2))
```

– If you are using an attached client workstation to perform the installation, perform the following steps:

1. Ensure that the requirements specified in “Using an attached client” on page 27 are met.
2. Ensure that the Host Servers option is installed and running on the AS/400 machine. You can start Host Servers by using the Start Host Servers (**STRHOSTSVR**) command at a Control Language (CL) prompt.
3. Ensure that TCP/IP is installed and running on the AS/400 machine. You can start TCP/IP by using the Start TCP/IP (**STRTCP**) command at a CL prompt.
4. Copy the `uninstall.jar` and `uninstall.dat` files from the `/QIBM/ProdData/mqak/uninstall` directory on the AS/400 system to a directory on the client workstation.
5. If the workstation is running JDK 1.1.8, enter the following command:

```
jre -cp uninstall.jar; run -os400
```

If the workstation is running JDK 1.2.2 or higher, enter the following command:

```
java -classpath uninstall.jar; run -os400
```

The uninstall process does not remove any files or directories created after the kernel was installed. This includes all log files and data files copied by the user.

Upgrading the kernel

If you have installed MQSeries Adapter Kernel version 1.0, either with or without the Corrective Service Diskette (CSD), perform the following steps before installing MQSeries Adapter Kernel version 1.1:

__ Step 1. Back up the `aqmsetup` and `aqmconfig.properties` files to a location outside of the MQSeries Adapter Kernel installation directory.

__ Step 2. If the MQSeries Adapter Kernel version 1.0 CSD is installed, uninstall it as follows:

- On Windows NT, use one of the following methods:
  - From the Windows NT Start menu, click **Programs > MQSeries Adapter Kernel > Remove CSD**.
  - Use the Add/Remove Programs utility in the Control Panel.
  - Double-click the `aqmuninstallCSD.bat` file in the kernel's root directory.
  - Open a command prompt, change to the kernel's root directory, and enter the following command:

    ```
    java uninstallCSD
    ```
- On AIX, change to the kernel's root directory and enter one of the following commands:

  ```
  aqmuninstallCSD.sh
  ```

  ```
  java uninstallCSD
  ```

__ Step 3. Uninstall MQSeries Adapter Kernel version 1.0 as follows:

- On Windows NT, use one of the following methods:
  - From the Windows NT Start menu, click **Programs > MQSeries Adapter Kernel > Uninstall MQSeries Adapter Kernel**.
  - Use the Add/Remove Programs utility in the Control Panel.
  - Double-click the `aqmuninstall.bat` file in the kernel's root directory.
  - Open a command prompt, change to the kernel's root directory, and enter the following command:

    ```
    java uninstall
    ```
- On AIX, change to the kernel's root directory and enter one of the following commands:

  ```
  aqmuninstall.sh
  ```

  ```
  java uninstall
  ```

__ Step 4. Install MQSeries Adapter Kernel version 1.1. See "Installing the kernel" on page 32 for details.

__ Step 5. Restore the `aqmsetup` and `aqmconfig.properties` files to their previous locations in the MQSeries Adapter Kernel installation directory. Convert the `aqmconfig.properties` file to an `aqmconfig.xml` file. For more information on the `aqmconfig.xml` file, see "The configuration file" on page 47.

Chapter 4. Using the kernel

This chapter contains the following information about using the kernel:

Preparing for production

Before putting the kernel into production, perform the following tasks:

1. Design the overall system architecture, including MQSeries Adapter Offering, MQSeries or other messaging software, and optionally MQSeries Integrator, based on your site’s requirements and conditions. Typically, the architecture is unique to each site.
2. Build the source adapters and target adapters by using MQSeries Adapter Builder, then test and deploy them.
3. Develop application-specific interfaces outside of MQSeries Adapter Offering for the following purposes:
   - To enable the source adapter to acquire the application data from the source application
   - To enable the target application to acquire the message data from the target adapter

   The exact nature of the application-specific interface depends on the characteristics of the source application and of the target application. Some examples of application-specific interfaces include:
   - API calls and user exits
   - File reads and writes
   - Database triggers
   - Message queues
4. Configure the kernel to support the run-time flow: sending, routing, tracing, and delivering messages. See “Configuring the kernel” on page 46 for information on configuring the kernel.

5. Configure MQSeries or other messaging software and, optionally, MQSeries Integrator to support your overall system architecture. See “Configuring MQSeries and MQSeries Integrator” on page 64.
6. If required, develop Java logon classes to support message delivery. They are specific to each target application. They are needed only if the target adapter requires information for logging on and connecting to the application.
7. Test the whole system—that is, MQSeries Adapter Kernel with your source adapters and target adapters, your application-specific interfaces, and your custom code—before putting the system into production.
8. Deploy the system in the production environment.
9. Turn on the kernel by starting one or more adapter daemons and, optionally, trace servers, based on your unique topology. Ensure that the source application is started. If the source adapter is run in the source application’s process, the source adapter is automatically started with the source application; no extra steps are needed to start the source adapter. Any daemon or server that contains the source adapter needs to be started. See “Starting the kernel” on page 65.

Configuring the kernel

Configuration of the kernel is determined by several customizable files. By using a standard text editor, edit the files to configure the kernel for your site. The following files are involved in configuring the kernel:

- The `aqmsetenv.bat` (Windows systems) or `aqmsetenv.sh` (UNIX) file, which sets environment variables. Edit this file to change system environment variables after installation, if desired. Environment variables set by this file include PATH, CLASSPATH, and LIBPATH. These variables are set automatically by the installation program on Windows systems. To set these variables automatically when you log in to UNIX, add the values specified in the `aqmsetenv.sh` file to your `.profile` file (if you use Bourne shell or Korn shell) or `.cshrc` file (if you use C shell).

  For information on setting the appropriate environment variables on OS/400, see ″Setting system environment variables on OS/400″ on page 35.
- The `aqmsetup` file, which provides several initial setup values for the kernel. See “The setup file” on page 47 for more information.
- The `aqmconfig.xml` file, which configures the kernel. See “The configuration file” on page 47 for additional information. This file contains most of the values that configure the kernel.
- The `aqmcreateq.bat` (Windows systems) or `aqmcreateq.sh` (UNIX and OS/400) file, which is a script that creates MQSeries queues. See “Creating MQSeries queues” on page 69.

All of these files include comments that can help you edit them.

It is recommended that you back up these files. For additional information, see “Maintaining the kernel” on page 67.

The setup file

The setup file, aqmsetup, controls several of the kernel’s initial settings, including the following:

- The location of the configuration file. See “The configuration file”.
- The location of XML DTDs, if not in the current directory.
- Java JNI environment variables for the C interface, for changing the amount of memory used. This applies when a C executable module starts a process and a Java virtual machine is instantiated by that process. Memory use can be controlled in this case by uncommenting and modifying the following lines in the aqmsetup file:

  ```
  #AQM_JNI_NATIVESTACKSIZE=1048576
  #AQM_JNI_JAVASTACKSIZE=4194304
  #AQM_JNI_MINHEAPSIZE=16777216
  #AQM_JNI_MAXHEAPSIZE=268435426
  ```

  All sizes are in bytes.

A sample aqmsetup file is provided in “Appendix E. Sample of the setup file” on page 95 and is also included in the samples directory of the MQSeries Adapter Kernel installation.

If necessary, edit the setup file when MQSeries Adapter Kernel is first installed. After installation, edit the file only if the kernel encounters a Java out-of-memory problem, as discussed in the previous list.

The configuration file

This section discusses the aqmconfig.xml file, which determines the kernel’s configuration. “Syntax and organization of the configuration file” on page 48 provides information on the syntax and organization of the configuration file. “Editing the configuration file” on page 59 provides best-practice suggestions for editing the configuration file.

Configuration of MQSeries Adapter Kernel is determined by an XML file named aqmconfig.xml. A sample configuration file is included in “Appendix D. Sample of the configuration file” on page 89 and is also included in the samples directory of the MQSeries Adapter Kernel installation.

The values specified in the configuration file control the following elements of the kernel:

- Source logical identifiers
- Destination logical identifiers
- Adapter daemons and workers
- Trace clients

- Trace servers
- Marshaling and routing of messages, determined by the following specifications:
  - The names of receive queues, error queues, and reply queues
  - A default destination or default list of destinations to which messages are to be sent
  - The name of the queue manager that gets or sends the message
  - The receive timeout for reply messages
  - The target adapter class on the target side of the kernel that processes each message
  - The minimum number of workers
  - Enabling and disabling trace, and control of trace level
  - Enabling and disabling audit logging
  - The logon class to instantiate on the target side of the kernel
- Communication mode

An understanding of the run-time flow is important for editing the configuration file. See "Run-time flow" on page 12 for more information.

Syntax and organization of the configuration file

Because the configuration of MQSeries Adapter Kernel is based on the Lightweight Directory Access Protocol (LDAP), the structure of the configuration file mirrors LDAP. The top-level XML element, `Epic`, represents the top level of the LDAP directory, and subordinate LDAP objects are represented by XML elements nested within the top-level element. Some of the XML elements have required attributes that represent LDAP information. Values are added to the configuration either as the contents of elements or as attributes of elements. An example of a configuration value assigned as the content of an element is `<epictracelevel>-1</epictracelevel>`, which assigns the value `-1` (all possible messages) to the `epictracelevel` element. An example of a configuration value assigned as an attribute of an element is `<ePICTraceHandler epictracehandler="com.ibm.logging.ConsoleHandler">`, which assigns the `com.ibm.logging.ConsoleHandler` class to be used as the trace handler.

The following is a list and description of the high-level elements used in the configuration file. "XML elements used in the configuration file" on page 51 lists and describes the full set of elements used in the configuration file. See the sample configuration file for examples of how the different elements are used in context.

- `Epic`—The required top-level element for the `aqmconfig.xml` file.
- `ePICApplications`—The required child of the `Epic` element.

- ePICApplication—The required child of the ePICApplications element. It lists and defines the applications to be serviced by the kernel; one fully defined ePICApplication element (including child elements) is required for each application.
- AdapterRouting—An optional child of the ePICApplication element. It defines the queue manager and related information.
- ePICBodyCategory—The required child of the AdapterRouting element. It sets the body category for messages to be routed by the kernel.
- ePICBodyType—The required child of the ePICBodyCategory element. It sets the body type of messages to be routed by the kernel. It contains definitions for items such as message destinations, communication modes for receiving messages, and message formatters.
- ePICAdapterDaemonExtensions—An optional child of the ePICApplication element representing an adapter daemon application. It contains information related to adapter daemons, including application identifiers and number of adapter workers.
- ePICTraceExtensions—An optional child of the ePICApplication element representing a trace client application or trace server element. It defines information related to tracing.

Figure 3 on page 50 shows the high-level structure of the configuration file. This is not a working example of a configuration file; it is simply meant to demonstrate the relationships and dependencies among the high-level elements. See "Appendix D. Sample of the configuration file" on page 89 for a complete example.

```
<?xml version="1.0" encoding="UTF-8"?>
<Epic o="ePIC">
  <ePICApplications o="ePICApplications">
    <!-- The following <ePICApplication> tag configures the kernel to work with
    an application named APP1. -->
    <ePICApplication epicappid="APP1">
      <!-- Tags here specify logging and trace information for the APP1
      application. -->
      <AdapterRouting cn="epicadapterrouting">
        <!-- Tags here specify the queue manager and its attributes. -->
        <ePICBodyCategory epicbodycategory="DEFAULT">
          <ePICBodyType epicbodytype="DEFAULT">
            <!-- Tags here specify the details of transporting and processing messages
             from APP1. -->
          </epicBodyType>
        </ePICBodyCategory>
      </AdapterRouting>
    </ePICApplication>
    <!-- The following <ePICApplication> tag starts an adapter daemon for the
    APP1 application. -->
    <ePICApplication epicappid="APP1Daemon">
      <!-- Specifications for the APP1Daemon adapter daemon, which works with
      the APP1 application. -->
      <ePICAdapterDaemonExtensions cn="epicappextensions">
        <epicdepappid>APP1</epicdepappid>
        <epicminworkers>1</epicminworkers>
      </ePICAdapterDaemonExtensions>
    </ePICApplication>
    <!-- The following <ePICApplication> tag configures the kernel to work with
    an application named APP2. -->
    <ePICApplication epicappid="APP2">
      <!-- Tags here specify logging and trace information for the APP2
      application. -->
      <AdapterRouting cn="epicadapterrouting">
        <!-- Tags here specify the queue manager and its attributes. -->
        <ePICBodyCategory epicbodycategory="DEFAULT">
          <ePICBodyType epicbodytype="DEFAULT">
            <!-- Tags here specify the details of transporting and processing messages
            from APP2. -->
          </epicBodyType>
        </ePICBodyCategory>
      </AdapterRouting>
    </ePICApplication>
    <!-- The following <ePICApplication> tag configures a trace client named
    TraceClient. -->
    <ePICApplication epicappid="TraceClient">
      <ePICTraceExtensions cn="epicappextensions">
        <!-- Tags here specify attributes of the trace client. -->
      </ePICTraceExtensions>
    </ePICApplication>
  </ePICApplications>
</Epic>
```

Figure 3. High-level structure of the configuration file

The following is a list and description of the full set of elements used in the configuration file. If an element is noted as having a default value, the kernel uses that value if an element of the configuration requires a value that is not explicitly specified.

XML elements used in the configuration file

`Epic` Top-level element for the configuration file.

Child elements:

- `context`
- `ePICApplications` (required)

Attributes: `o="ePIC"` (required)

`context`

Specifies the root of the JNDI file system context (FSContext) when JMS objects are used. The default is the current directory. Required if JMS is used. See "Using JMS object storage" on page 78 for information about using JMS objects with MQSeries Adapter Kernel.

Child elements: None

Attributes: None

`ePICApplications`

Contains information about the applications serviced by the kernel.

Child elements: `ePICApplication` (required)

Attributes: `o="ePICApplications"` (required)

`ePICApplication`

Specifies information about an application serviced by the kernel.

Child elements:

- `epiclogging`
- `epictrace`
- `epictracelevel`
- `epictraceclientid`
- `epiclogoninfoclassname`
- `AdapterRouting`
- `ePICTraceExtensions`
- `ePICAdapterDaemonExtensions`

Attributes: `epicappid="`*`application_ID`*`"`, where *application_ID* is a valid application identifier (required)

epiclogging

Determines whether to audit logging. Audit logging requires the WebSphere Business-to-Business Integrator product. The default is `false`.

Child elements: None

Attributes: None

epictrace

Determines whether to use tracing. The default is `false`.

Child elements: None

Attributes: None

epictracelevel

Sets the level of tracing, using the constants specified by the `com.ibm.logging.IRecordType` class. The default is `0` (no messages). See the *Problem Determination Guide* for details about tracing and for a full listing of valid trace levels.

Child elements: None

Attributes: None

epictraceclientid

Specifies the name of the trace client application. The default is `TraceClient`.

Child elements: None

Attributes: None

epiclogoninfoclassname

Specifies the name of the logon class used to connect to an application. The default is `com.ibm.epic.adapters.eak.adapterdaemon.EpicLogonDefault`.

Child elements: None

Attributes: None

AdapterRouting

Contains information about message types and the routing of messages.

Child elements:

- `epicmqppqueuemgr`
- `epicuseremotequeuemanagertosend`
- `epicmqppqueuemgrhostname`
- `epicmqppqueuemgrportnumber`
- `epicmqppqueuemgrchannelname`

- epicjmsconnectionfactoryname
- ePICBodyCategory (required)

Attributes: cn="epicadapterrouting" (required)

epicmqppqueuemgr

If MQSeries is being used as the transport mechanism, specifies the name of the queue manager to be used. If not specified or if specified as DEFAULT, the default queue manager is used.

Child elements: None

Attributes: None

epicuseremotequeuemanagertosend

If MQSeries is being used as the transport mechanism, determines whether to use a remote queue manager to send messages. The default is false.

Child elements: None

Attributes: None

epicmqppqueuemgrhostname

If MQSeries is being used as the transport mechanism, specifies the TCP/IP hostname of the machine on which the queue manager resides. Required if MQSeries Client is being used.

Child elements: None

Attributes: None

epicmqppqueuemgrportnumber

If MQSeries is being used as the transport mechanism, specifies the port number of the queue manager server process. The default is 1414 (the MQSeries default). Used only if MQSeries Client is being used.

Child elements: None

Attributes: None

epicmqppqueuemgrchannelname

If MQSeries is being used as the transport mechanism, specifies the channel name of the queue manager server. Required if MQSeries Client is being used.

Child elements: None

Attributes: None

epicjmsconnectionfactoryname

If JMS is being used as the transport mechanism, specifies the JMS Connection factory name. The value must be specified as *attribute=object*, where *attribute* is the LDAP attribute and *object* is the JMS Connection object. The object is expected to be stored under the AdapterRouting element. For instance, for a JMS connection object named QCFTEST1 with an LDAP attribute of cn, the value specified by this element is cn=QCFTEST1.

Child elements: None

Attributes: None

ePICBodyCategory

Specifies the body category of messages being sent.

Child elements: ePICBodyType (required)

Attributes: epicbodycategory=*body_category*, where *body_category* specifies the body category of messages being sent (required)

ePICBodyType

Specifies the body type of messages being sent.

Child elements:

- epiccommandclassname
- epicdestids
- epicreceivemode
- epicmessageformatter
- epicreceivetimeout
- epicreceivemqppqueue
- epicerrormqppqueue
- epicreplymqppqueue
- epicjmsreceivequeuename
- epicjmserrorqueuename
- epicjmsreplyqueuename
- epicreceivefiledir
- epiccommitfiledir
- epicerrorfiledir

Attributes: epicbodytype=*body_type*, where *body_type* specifies the body type of messages being sent (required)

epiccommandclassname

Specifies the name of a target adapter or command that is invoked to process messages. Required if an adapter daemon is being used to receive messages.

Child elements: None

Attributes: None

epicdestids

Specifies the identifiers of one or more applications to be used as message destinations. Required if the application is sending messages and the destination logical ID is set to NONE.

Child elements: None

Attributes: None

epicreceivemode

Specifies the communication mode to be used. See “Appendix A. Communication modes” on page 75 for a listing and explanation of valid communication modes. Required if the application is receiving messages.

Child elements: None

Attributes: None

epicmessageformatter

Specifies the message formatter to use, dependent on the value of epicreceivemode and on the transport method used. See Table 3 on page 76 and Table 4 on page 77 for details on message formatters and transport methods.

Child elements: None

Attributes: None

epicreceivetimeout

Specifies, in milliseconds, the length of time the receiver waits for messages before it times out. The default is 0. A value of -1 specifies no timeout (wait indefinitely).

Child elements: None

Attributes: None

epicreceivemqppqueue

Specifies the name of the queue from which to receive messages. Required when the epicreceivemode element specifies an MQSeries transport mode. See “Appendix A. Communication modes” on page 75 for a list of MQSeries transport modes.

Child elements: None

Attributes: None

epicerrormqppqueue

Specifies the name of the queue on which to put error messages. Required if error-message queueing is being used and the `epicreceivemode` element specifies an MQSeries transport mode. See "Appendix A. Communication modes" on page 75 for a list of MQSeries transport modes.

Child elements: None

Attributes: None

`epicreplymqppqueue`

Specifies the name of the queue from which to receive reply messages. Required if reply requests are being used and the `epicreceivemode` element specifies an MQSeries transport mode. See "Appendix A. Communication modes" on page 75 for a list of MQSeries transport modes.

Child elements: None

Attributes: None

`epicjmsreceivequeuename`

Specifies the name of the queue from which to receive messages. Required for communication mode JMS. The object is expected to be stored under the `ePICBodyType` element. The value must be specified as *`attribute=object`*, where *attribute* is the LDAP attribute and *object* is the name of the JMS queue object. For instance, for a JMS object named `TEST1AIQ` with an LDAP attribute of `cn`, the value specified by this element is `cn=TEST1AIQ`.

Child elements: None

Attributes: None

`epicjmserrorqueuename`

Specifies the name of the queue on which to put error messages. Required if error-message queueing is being used with communication mode JMS. The object is expected to be stored under the `ePICBodyType` element. The value must be specified as *`attribute=object`*, where *attribute* is the LDAP attribute and *object* is the name of the JMS queue object. For instance, for a JMS object named `TEST1AEQ` with an LDAP attribute of `cn`, the value specified by this element is `cn=TEST1AEQ`.

Child elements: None

Attributes: None

`epicjmsreplyqueuename`

Specifies the name of the queue from which to receive reply messages. Required if reply requests are being used with communication mode JMS. The object is expected to be stored under the `ePICBodyType` element. The value must be specified as *`attribute=object`*, where *attribute* is the LDAP attribute and *object* is the name of the JMS queue object. For instance, for a JMS object named `TEST1RPL` with an LDAP attribute of `cn`, the value specified by this element is `cn=TEST1RPL`.

Child elements: None

Attributes: None

`epicreceivefiledir`

Specifies the name of the directory from which to receive messages. Required for communication mode FILE.

Child elements: None

Attributes: None

`epiccommitfiledir`

Specifies the name of the directory in which to hold received messages until they are committed. Required for communication mode FILE when messages are being received.

Child elements: None

Attributes: None

`epicerrorfiledir`

Specifies the name of the directory into which to put error messages. Required if error-message queueing is being used with communication mode FILE.

Child elements: None

Attributes: None

`ePICAdapterDaemonExtensions`

Contains information about adapter daemon extensions.

Child elements:

- `epicdepappid`
- `epicminworkers`

Attributes: `cn="epicappextensions"` (required)

`ePICTraceExtensions`

Contains information about trace extensions. See the *Problem Determination Guide* for a full discussion of this element and its children.

Child elements:

- `epicdepappid`
- `epictracesyncoperation`
- `epictracemessagefile`
- `epictracehandler`

- `ePICTraceHandler`

Attributes: `cn="epicappextensions"` (required)

`epicdepappid`

Specifies the identifier of the application the adapter daemon is servicing. Defaults to the application ID with which the adapter daemon was started.

Child elements: None

Attributes: None

`epicminworkers`

Specifies the number of adapter workers started by the adapter daemon. The default is 1.

Child elements: None

Attributes: None

Adding adapter information to the configuration

When a new adapter is added to the kernel configuration, several specifications, at a minimum, must be added to the configuration file. For an example of a minimum configuration file, see the `aqmconfig.minimum.xml` file. This file is included in "Appendix D. Sample of the configuration file" on page 89 and is also included in the `samples` directory of the MQSeries Adapter Kernel installation.

The following specifications represent the minimum amount of information that must be added to the configuration when a new adapter is added:

- **Source adapter** (sending messages):
  - The identifier of the application under which the source adapter is running.
  - The default queue manager. If MQSeries is used as the transport mechanism and is installed and running on the same machine as the source adapter, you do not need to specifically configure the queue manager.
  - Destination logical identifiers for messages. If all messages go to the same destination, then use a body category of DEFAULT and a body type of DEFAULT.
  - A receive queue for each destination logical identifier to which the source adapter is sending messages.
- **Target adapter** (receiving messages):
  - The identifier of the application under which the target adapter is running.

  - The default queue manager. If MQSeries is used as the transport mechanism and is installed and running on the same machine as the source adapter, you do not need to specifically configure the queue manager.
  - The receive mode for MQSeries. Typically this is the same for all messages; if so, use a body category of DEFAULT and a body type of DEFAULT.
  - The receive queue. If this is the same for all messages, then use a body category of DEFAULT and a body type of DEFAULT.
  - The error queue, in case an error occurs when the target adapter processes the message. Typically this is the same for all messages; if so, use a body category of DEFAULT and a body type of DEFAULT.
  - The target adapter class name to invoke when a message is received. This is specific to body category and body type.
  - Receive timeout value. This is highly recommended to prevent high CPU usage. Typically this is the same for all messages; if so, use a body category of DEFAULT and a body type of DEFAULT.

  For additional target adapters, the same information can be sufficient if the same receive queue is being used. If this is the case, the only information that needs to be specified differently is the target adapter class name to invoke for the specific body category and body type.

- **Trace specifications**:
  - Whether trace is on or off.
  - The trace level.
  - Additional trace specifications, including trace destination, for source adapters and target adapters. By default, trace is displayed in the command prompt window or terminal where the kernel was started.

Editing the configuration file

Use a text editor or a dedicated XML editor to edit the configuration file. A DTD file named `aqmconfig.dtd` is provided in the `samples` directory of the kernel installation for users of XML editors. An XML editor called Xeena can be downloaded from the IBM alphaWorks Web site at http://www.alphaworks.ibm.com/. The following recommendations apply to editing the configuration file:

- Before you begin editing the configuration file, gather all pertinent information about your desired configuration. This includes the names of applications and queues that are involved in the configuration, the types of messages being exchanged, the communication mode or modes being used, and information about trace programs and other extensions.

- Copy the sample aqmconfig.xml file from the samples directory to your desired location. This needs to be done during installation of the kernel; see "Installing the kernel" on page 32 for details. Do not rename the copy of the configuration file. Edit the copy.
- Use comments to identify different sections of the configuration file and to document the specific values used in your configuration (for instance, application identifiers, message queue names, and timeout values). In XML, comments start with the characters <!-- and end with the characters -->. Comments can span multiple lines, as in the following example:

  ```
  <!--
      Comment text
  -->
  ```

  Note that XML does not permit comments inside other comments.
- Organize the configuration file according to the application identifiers. Keep the entries for each application identifier together.
- If you are not using a dedicated XML editor, use a text editor that preserves the line endings and does not split lines when the file is saved. Examples of this kind of text editor are Notepad on Windows systems and vi or Emacs on UNIX.
- Remember that XML is case sensitive; be extremely careful to use the correct case for all tag (element) names and attributes. Using an incorrect case in the tagging can invalidate the configuration file. Using a dedicated XML editor can help prevent case errors.
- If you want to use default values for body category and body type and the values are not already defaulted, you must configure the value DEFAULT for each in the configuration file. If you do not, the kernel does not use any default values.
- Validate the configuration file before putting it into production. See "Validating the configuration file".
- The changes to the configuration file take effect the next time a process starts. If a process is running when the configuration file is changed, the process must be stopped and then restarted for the changes to take effect. Be extremely careful if you edit the configuration file that is currently in production.
- Back up the configuration file each time you edit it.

Validating the configuration file

After the configuration file is edited and before it is put into production, it is recommended that you validate it. To validate the configuration file, perform the following general steps:

1. Create a configuration file validation directory within which to validate and set up the test.
2. Create a validation XML message.

3. Set up message queues to support the validation test.
4. Set up and then execute a configuration file validation test that sends a message and that receives a message.
5. Examine the results of the test to determine if the configuration file is correct.

The utility that helps to create a validation XML message and the configuration file validation test are both provided as part of the kernel.

The configuration file validation test invokes the `sendMsg` method and sends a validation XML message from a native adapter on the source side of the kernel to an adapter daemon on the target side of the kernel. A source adapter and a target adapter are not required. However, if a target adapter is in place, you can also test sending the message to the target application.

The procedure follows.

Note: Several scripts are provided as a convenience for use in the procedure. If desired, copy the scripts and then edit the copies to make your own versions. If you are using OS/400, note that the UNIX versions of the scripts can be run in a **qsh** session. You can start a **qsh** session by entering the Start QSH (**STRQSH**) command at a Control Language (CL) prompt.

__ Step 1. Open a command prompt window.

__ Step 2. Create a configuration file validation directory. Copy the configuration file and the setup file into it.

__ Step 3. Change to the validation directory.

__ Step 4. Enter the following command to create the validation XML message:

- `aqmcrtmsg.bat` (Windows systems)
- `aqmcrtmsg.sh` (UNIX and OS/400)

__ Step 5. A list of options is displayed. Select an option and press Enter. Enter a value for each. The order in which values are entered is not important. Examples of options are `set sourcelogicalid`, `set msgtype`, and `set bodycategory`. You must enter values for options 20, 21, 22, and 23. You can use options 24 or 241 to provide message body data. Other values are not required.

__ Step 6. Enter option 1 to create the validation XML file. The validation XML file is created in the current directory and is named `EpicMessage`*`nn`*`.xml`, where *nn* is the number of the XML file.

__ Step 7. Enter option 0 to exit from the validation utility.

__ Step 8. Set up the appropriate message queues to support the validation.

__ Step 9. Set the AQMSETUPFILE environment variable to point to the setup file in the validation directory temporarily:

- At a command prompt on Windows systems, enter the following:

  ```
  set AQMSETUPFILE=E:\runtimefiles\aqmsetup
  ```

  where `E:\` represents the correct drive and *runtimefiles* is the validation directory.
- On UNIX and OS/400, enter the following command. The command example assumes that you are using Korn shell; if you are using a different shell, change the command accordingly.

  ```
  export AQMSETUPFILE=root_directory/runtimefiles/aqmsetup
  ```

  where *root_directory* is the kernel's installation directory and *runtimefiles* is the validation directory. On OS/400, the `aqmsetup` file must always be located in your IFS home directory (`/home/`*`username`*).

If necessary, edit the setup file in the validation directory to point to the configuration file that is being validated.

__ Step 10. Choose which of the following to test:

- Only the source side of the kernel.
- Whether the message can be routed all the way to the target application. This test requires a target adapter to be in place already.
- Tracing.

First test the source side, then test the target side. Turn off the adapter daemon to test only the source side. Turn on the adapter daemon to test the target side as well. If a target adapter is not in place already, you can still test whether the adapter daemon processes the message up to the point when it attempts to invoke the command for the appropriate target adapter. It is recommended that you enable tracing, especially if a target adapter is not already in place.

__ Step 11. Execute the validation test. From any directory, enter the following command:

- On Windows systems:

  ```
  aqmsndmsg.bat -a source_logical_identifier -f XML_message_file
  ```
- On UNIX and OS/400:

  ```
  aqmsndmsg.sh -a source_logical_identifier -f XML_message_file
  ```

  where:

source_logical_identifier
indicates the source logical identifier. This value must match the source logical identifier value entered for option 20 in Step 5 on page 61.

XML_message_file
indicates the XML message file.

Note: A list of all options for this test can be displayed by entering the following command:

On Windows systems:

```
aqmsndmsg.bat -?
```

On UNIX and OS/400:

```
aqmsndmsg.sh -?
```

Note that the `-?` works only on Korn shell; if you use another UNIX shell (such as Bourne shell or C shell), escape the question mark by using a backslash (that is, `-\?`).

__ Step 12. Examine the results. The validation message contains the correct body category, body type, and data.

- If you are testing only the source side of the kernel (that is, if the adapter daemon has not been started), examine the queue to which the message was to be routed.
  - If you see your validation message on that queue, those entries in the configuration file are validated.
  - If you do not see your validation message on that queue, check the exception file. If tracing is enabled, check the trace messages.
- If you are testing the target side of the kernel and a target adapter is in place, check the target application.
  - If your validation message is received by the target application, those entries in the configuration file are validated.
  - If your validation message is not received by the target application, check the exception file. If tracing is enabled, check the trace messages.
- If you are testing the target side of the kernel and no target adapter is in place, check the error queue for the validation message and the exception file for an exception message. If tracing is enabled, check the trace messages.

  - If you see your validation message on the error queue and an exception message, those entries in the configuration file are validated.
  - If you do not see your validation message on the error queue, check the exception file. If tracing is enabled, check the trace messages.

__ Step 13. If necessary, modify the configuration file and validate it again.

Configuring MQSeries and MQSeries Integrator

Configure MQSeries and optional software such as MQSeries Integrator to support the kernel as follows.

In MQSeries:

- Several queues are used for verifying the installation. If you use these queues for your test or production environments, you must clear them to verify installation. See "Verification procedure" on page 39 for the queues used for verifying installation.
- Set up queues to support transport of messages according to the routing scheme that you have designed.
- When creating queues, set the MAX_QUEUE_DEPTH environment variable to the maximum queue depth allowed.

In MQSeries Integrator, set up input and output queues in rules (version 1.1) or in messageflows (version 2) that correspond to the queues that are configured in the configuration file.

Performance recommendations

The following performance recommendations apply specifically to MQSeries Adapter Kernel:

- When XML DTDs are parsed, ensure that the DTD files reside in the same directory as the process that parses them. This reduces the effort required by the process to find the DTDs.
- When large messages are being sent and received, using message type RFH2 results in better performance than using message type XML.

See the MQSeries documentation for general recommendations for improving performance.

Starting the kernel

To start the kernel, start the following items:

- Adapter daemon for each target application
- Trace server (optional)

Note that if the source adapter is run in the source application's process, the source adapter is automatically started with the source application; no extra steps are needed to start the source adapter. Any daemon or server that contains source adapters needs to be started. You do not start source adapters directly.

Start each adapter daemon and trace server by performing the following steps:

Note: Several scripts are provided as a convenience for use in the procedure. If desired, copy the scripts and then edit the copies to make your own versions. If you are using OS/400, note that the UNIX versions of the scripts can be run in a **qsh** session. You can start a **qsh** session by entering the Start QSH (**STRQSH**) command at a Control Language (CL) prompt.

__ Step 1. Start MQSeries or other messaging software and optional software such as MQSeries Integrator.

__ Step 2. Start associated other software that your site requires—for example, applications (outside the kernel) to read trace messages from queues.

__ Step 3. Open a command prompt. For each adapter daemon, enter the following command:

- On Windows systems:

```
aqmstrad.bat -a application_identifier [-bc body_category
-bt body_type] [-noretry]
```

- On UNIX and OS/400:

```
aqmstrad.sh -a application_identifier [-bc body_category
-bt body_type] [-noretry]
```

where:

-a *application_identifier*
Identifies the destination logical identifier that the adapter daemon serves.

-bc *body_category*
Specifies the body category that the adapter daemon worker uses for determining the communication mode and related information for receiving messages. If no value is provided, the adapter daemon processes using the value DEFAULT.

-bt *body_type*

Specifies the body type that the adapter daemon worker uses for determining the communication mode and related information for receiving messages. If no value is provided, the adapter daemon processes using the value DEFAULT.

-noretry

Specifies that the worker stops automatically when there are no more messages. If `-noretry` is not specified, then the worker continually polls the queue for messages and the adapter daemon must be stopped manually.

Note: If you need to modify Java startup parameters, edit the `aqmstrad.bat` (Windows systems) or `aqmstrad.sh` (UNIX and OS/400) file. See the comments inside the file for details.

__ Step 4. For each trace server, enter the following command:

- On Windows systems:

  ```
  aqmstrtd.bat -how -a source_application_identifier
  ```

- On UNIX and OS/400:

  ```
  aqmstrtd.sh -how -a source_application_identifier
  ```

where:

-how

Indicates how the trace messages are to be received. Possible values include the following:

- `socket`
- `ena`, that is, native adapter

-a *source_application_identifier*

Source application identifier. If no value is provided, the default value `TraceServer` in the configuration file is used.

See the *Problem Determination Guide* for more information about trace servers.

__ Step 5. After an adapter daemon or trace server is started, a process window remains open until you stop the adapter daemon. The process window can display exceptions. See "Exception messages" on page 68.

Stopping the kernel

To stop the kernel, stop each of the adapter daemons and trace servers. There are several ways to stop them:

- When you start the adapter daemon, set the parameter `-noretry`. See "Starting the kernel" on page 65.

- Go to the command prompt (Windows systems) or terminal (UNIX) from which the adapter daemon or trace server was started and enter **Ctrl-C**. Perform this step for each adapter daemon or trace server.
- On Windows systems, you can use the Task Manager to end the process.
- On UNIX, you can use the **ps** command to determine the number of the process, then use the **kill** command to end the process.

Maintaining the kernel

Set up a kernel maintenance plan. It is recommended that you periodically back up the following items.

- The configuration as specified in the following files:
  - `aqmconfig.xml`
  - `aqmsetup`
- Adapters that you have built and their associated files

Backing up or periodically deleting the contents of trace and other files used by the kernel to support its own processing is not required. Back up these files if desired. If trace messages are being routed to a single file instead of to multiple files, the single trace file can become very large. If the tracing level is set to capture a high level of detail (for instance, all trace messages or information messages), consider deleting the trace files periodically.

Diagnosing problems

You can use exception messages, trace messages, and the MQSeries error queue to help diagnose problems. The MQSeries Adapter Kernel produces exception messages and, if trace is enabled, trace messages. See the *Problem Determination Guide* for information on how to diagnose problems in an MQSeries Adapter Kernel environment.

To understand exception messages and trace messages, you must understand how the kernel works. The kernel uses an error queue to handle some errors. See “How the kernel works” on page 8.

You can identify the message that caused exception messages and trace messages by the combination of the unique message identifier and unique transaction identifier.

There is no identifier that enables you to definitively identify the same message in both the error queue and the kernel. However, you can manually correlate a message on the error queue with the corresponding exception message, trace message, or both. You can compare one or more of the following:

- Approximate time stamp
- Queue for the source logical identifier
- Queue for the destination logical identifier
- Body category
- Body type
- Unique message identifier
- Unique transaction identifier

If they match, then you probably have correlated the message on the error queue with the corresponding exception message or trace message.

Version number

Run `aqmversion.bat` (Windows systems) or `aqmversion.sh` (UNIX and OS/400) in the `bin` directory to display the version number of the kernel.

Exception messages

The kernel produces the following types of exception messages:

- The native adapter on the source side of the kernel throws exceptions to the source adapter. See the MQSeries Adapter Builder documentation for how the source adapter handles these exceptions.
- The native adapter on the target side of the kernel throws exceptions to the worker that manages the native adapter.
- The worker writes exceptions to the `EpicSystemExceptionFile`*`nnnnnnnn`*`.log` file, which resides in the same directory as the worker.
- The adapter daemon writes exception messages to an exception file called `EpicSystemExceptionFile`*`nnnnnnnn`*`.log` that resides in the same directory as the adapter daemon. Because the adapter daemon and its workers reside in the same directory, they all write to the same exception file. The adapter daemon also writes exception messages to the console (that is, the command prompt window or the terminal that was used to start it, if it was started from a window).

The kernel's trace exception messages are different from MQSeries exception messages. The following is an example of an exception message from the kernel:

```
2000.10.26 19:38:20.929 com.ibm.epic.adapters.eak.nativeadapter.LMSMQ
Thread Name=main receiveRequest(ENAService)  ePIC TEST2
TYPE_ERROR_EXC AQM5004: Received exception <com.ibm.epic.adapters.eak.common.
AdapterException> Message information: <AQM0114: com.ibm.epic.adapters.eak.
nativeadapter.MQNMRFH2Formatter::convertMessage(MQMessage): Expecting a message
with an MQHRF2 format and received a message with format <MQSTR   >.>
for <unmarshall Message()> having invalid data <(null)>
```

The values in an exception message depend on the nature of the message, possibly including the following items:

- Time stamp
- Source logical identifier
- Destination logical identifier
- Body category
- Body type
- Unique message identifier
- Unique transaction identifier
- Exception information

See “Common verification problems” on page 40 for common problems that you can encounter during verification of installation and for potential responses.

For control of Java memory utilization, see “Starting the kernel” on page 65.

Trace messages

The kernel can be configured to produce trace messages. For information on tracing, see the *Problem Determination Guide*.

Utilities

Creating MQSeries queues

You can use batch files or shell scripts to automate the creation of MQSeries queues. Run `aqmcreateq.bat` (Windows systems) or `aqmcreateq.sh` (UNIX and OS/400), using the application name as a parameter. These files create the following queues for each application:

- Receive queue, called *application_name*AIQ.
- Error queue, called *application_name*AEQ.
- Reply queue, called *application_ name*RPL.

Chapter 5. Using MQSeries Adapter Kernel APIs

The kernel includes APIs that are used for functions such as sending and receiving messages, creating and parsing XML, and managing the kernel configuration. These APIs are used by adapters created by using the MQSeries Adapter Builder. The MQSeries Adapter Kernel Information Center includes associated online API documentation in Javadoc HTML format.

The kernel is intended to be used with adapters built by the user by using the MQSeries Adapter Builder. The kernel is not intended to be used by calls to the kernel APIs from custom code alone. The online API documentation is provided only as an aid to understanding how the kernel functions and an aid to diagnostics.

The kernel online API documentation is located in the `documentation` directory.

Chapter 6. Obtaining additional information

There are several sources of information that can be useful when you are using MQSeries Adapter Offering. For additional information on MQSeries Adapter Kernel, see the *Problem Determination Guide* document, available from the MQSeries Adapter Kernel Information Center that is installed with the product. The *Problem Determination Guide* provides information on solving specific problems that can arise when using the kernel. For information on MQSeries Adapter Builder, see that product's Information Center and online help system.

Available on the Internet

The MQSeries product family Web site is at http://www.ibm.com/software/ts/mqseries/. By following links from this Web site, you can:

- Obtain latest information about the MQSeries product family, including MQSeries Adapter Offering.
- Access MQSeries books in HTML and PDF formats, possibly including a more recent edition of this book. The direct link to the MQSeries library page is http://www.ibm.com/software/ts/mqseries/library/manualsa/.
- Download MQSeries SupportPacs.

For information on using MQSeries on OS/400, see the OS/400 library at http://www.ibm.com/servers/eserver/iseries/library/. Also see the OS/400–specific books available from the MQSeries library Web site at http://www.ibm.com/software/ts/mqseries/library/manualsa/.

References

The following reference material discusses topics covered in this document:

- The Open Applications Group Web site at http://www.openapplications.org/
- The Extensible Markup Language (XML) 1.0 W3C Recommendation at http://www.w3.org/TR/1998/Rec-xml-19980210

These are not IBM Web sites.

Appendix A. Communication modes

This appendix provides information on the communication modes supported by MQSeries Adapter Kernel and on the Java classes that are used to support them. Some of the communication modes are provided as convenience modes with default formatters. See Table 3 on page 76 for the default formatters that are used with the convenience modes.

The following communication modes are supported:

MQPP
The kernel transports messages by using MQSeries base services. This is a convenience mode.

MQRFH1
The kernel transports messages by using MQSeries and brokers messages by using MQSeries Integrator version 1.1. This is a convenience mode.

MQRFH2
The kernel transports messages by using MQSeries and brokers messages by using MQSeries Integrator version 2. This is a convenience mode.

MQBD
The kernel transports messages by using MQSeries base services but sends and receives body data only. This is a convenience mode. The following characteristics are unique to this mode:

- It can send only body data, not message header values.
- It can receive messages that contain only body data. It uses the following default message header values for received messages:
  - `SourceLogicalApplicationID`—The value in the `ENAService` object used in the receive method call.
  - `BodyCategory`—The value in the `ENAService` object used in the receive method call.
  - `BodyType`—The value in the `ENAService` object used in the receive method call.
  - `Acknowledgment`—If the received MQMessage is an MQSeries REQUEST, then `Acknowledgment` is set to `1`.
  - `BodyData`—The message data received from MQSeries.

  All other header values use the normal defaults.

MQ
The kernel transports messages by using MQSeries base services.

JMS The kernel transports messages by using the Java Messaging Service (JMS). See "Using JMS object storage" on page 78 for information on using JMS objects with MQSeries Adapter Kernel.

FILE The kernel puts messages into a file and gets them from a file. This mode is provided for diagnostic purposes only.

Table 2 lists the communication modes and the Java classes that support them. All Java classes are from the Java package `com.ibm.epic.adapters.eak.nativeadapter`. Note that any Java class that supports the logical message service (LMS) can be specified as a communication mode; in this case, the class itself is used to support communication.

Table 2. Communication modes and supporting Java classes

Communication mode	Java class	Notes
MQPP	`LMSMQBindingMQPP`	Requires installation of MQSeries
MQRFH1	`LMSMQBindingMQRFH1`	Requires installation of MQSeries
MQRFH2	`LMSMQBindingMQRFH2`	Requires installation of MQSeries
MQBD	`LMSMQMQBD`	Requires installation of MQSeries
MQ	`LMSMQBinding`	Requires installation of MQSeries
JMS	`LMSJMS`	Requires installation of JMS
FILE	`LMSFile`	None

Table 3 lists the communication modes and their associated formatter interfaces. Table 4 on page 77 cross-references formatter interfaces, formatter classnames, and their uses. All formatters are from the Java package `com.ibm.epic.adapters.eak.nativeadapter`. Note that any formatter class can be specified for the communication mode; in this case, the specified formatter class is used as the formatter.

Table 3. Communication modes and formatter interfaces

Communication mode	Formatter interface	Default formatter
MQPP	`MQFormatterInterface`	`MQNMXMLFormatter`
MQRFH1	`MQFormatterInterface`	`MQNMRFH1Formatter`
MQRFH2	`MQFormatterInterface`	`MQNMRFH2Formatter`

Table 3. Communication modes and formatter interfaces (continued)

Communication mode	Formatter interface	Default formatter
MQBD	`MQFormatterInterface`	`MQNMBDFormatter`
MQ	`MQFormatterInterface`	`MQNMXMLFormatter`
JMS	`JMSFormatterInterface`	`JMSNMRFH2Formatter`
FILE	`StringFormatterInterface`	`NMXMLFormatter`

Table 4. Formatter interfaces, formatter classnames, and purposes

Formatter interface	Formatter classname	Purpose
`MQFormatterInterface`	`MQNMXMLFormatter`	EpicMessage as XML
	`MQNMRFH1Formatter`	EpicMessage as RFH1
	`MQNMRFH2Formatter`	EpicMessage as RFH2
	`MQNMDBFormatter`	Body data only
`JMSFormatterInterface`	`JMSNMXMLFormatter`	EpicMessage as XML
	`JMSNMRFH2Formatter`	EpicMessage as RFH2
	`JMSBodyDataFormatter`	Body data only
`StringFormatterInterface`	`NMXMLFormatter`	EpicMessage as XML

Table 5 lists the supported LMS classes and their degree of transactional support. See "Transactional capabilities" on page 22 for information about using transactions with MQSeries Adapter Kernel.

Table 5. LMS classes and transactional support

LMS class	Transactional support
`LMSMQBindingMQPP`	Single phase
`LMSMQBindingMQRFH1`	Single phase
`LMSMQBindingMQRFH2`	Single phase
`LMSMQMQBD`	Single phase
`LMSMQBinding`	Single phase
`LMSJMS`	Single phase
`LMSFILE`	No support

Using JMS object storage

The names of JMS objects are stored by using the FSContext file implementation of JNDI, which comes as part of the MQSeries JMS SupportPac. The context (directory structure) that the kernel uses for FSContext follows the LDAP hierarchy by using the distinguishing attribute with the associated value for the directory name. For example, for the LDAP hierarchy `o=ePIC, o=ePICApplications, epicappid=TEST1`, the directory structure is `o-ePIC/o-ePICApplications/epicappid-TEST1`.

To create the context and objects, use the JMS Admin tool that is provided with the JMS installation. The basic steps are defining a context, then changing the context. Changing the context moves you into the context. Create the JMS objects in the appropriate places. Following are example commands for creating the context structure and JMS objects. In this example, the application ID is `TEST1`.

```
#
# This is a script to use with the JMS administration (JMSAdmin) tool
# This tool requires the JMSAdmin.config to be set to either use of
# FSCONTEXT or LDAP.  This script will work with either.
#
#
# Example usage: MQSeries root\java\bin\jmsadmin.bat < aqmjmscreatesample.scp
#
# This script will create contexts for testing using the file system provider
# =UP means return to the parent context
# =INIT means return to root context.  In this example one directory level
#       above o-ePIC
# Always required.
define ctx(o-ePIC)
change ctx(o-ePIC)
# Always required.
define ctx(o-ePICApplications)
change ctx(o-ePICApplications)
# Application id is TEST1, requires a context.
define ctx(epicappid-TEST1)
change ctx(epicappid-TEST1)
# Always required.
define ctx(cn-epicadapterrouting)
change ctx(cn-epicadapterrouting)

# This will hold the JMS QueueConnectionFactory object
define ctx(cn-QCFTEST1)
change ctx(cn-QCFTEST1)

# Create the JMS QueueConnectionFactory object whose name is QCFTEST1
# Using MQSeries in server (bindings) mode.
define qcf(QCFTEST1) qmgr(yourQManagerName) tran(BIND)

change ctx(=UP)

# BodyCategory is DEFAULT
```

```
define ctx(epicbodycategory-DEFAULT)
change ctx(epicbodycategory-DEFAULT)

# BodyType is DEFAULT
define ctx(epicbodytype-DEFAULT)
change ctx(epicbodytype-DEFAULT)

# This will hold the JMS Queue object whose name is TEST1AIQ
define ctx(cn-TEST1AIQ)
change ctx(cn-TEST1AIQ)

# Create the JMS Queue object whose name is TEST1AIQ
# q(JMS Q Object Name) queue(MQSeries Queue name)
define q(TEST1AIQ) queue(TEST1AIQ)

# Can move up and define other contexts and JMS objects.

# Quit the administration tool.
end
```

Appendix B. Validated configurations

There are many potential configurations and combinations of MQSeries, MQSeries Adapter Offering, and MQSeries Integrator. Each of these members of the MQSeries product family is rich in features and configurations. Further, you can combine functionalities in MQSeries, MQSeries Adapter Offering, and MQSeries Integrator. Some functionality in one member of the MQSeries product family can partially overlap with functionality provided by other members of the family. You must determine how to use and combine the different message routing and delivery functionalities in MQSeries, MQSeries Adapter Offering, and MQSeries Integrator.

The following configurations of MQSeries, MQSeries Adapter Offering, and MQSeries Integrator have been validated as of the time of publication. Refer to the MQSeries Web site for the latest validated configurations.

MQSeries Adapter Kernel:

- Sending a message with acknowledgment requested and without acknowledgment requested.
- Using the MQPP communication mode (MQSeries). See “Appendix A. Communication modes” on page 75.
- Message routing and delivery:
  - Sending a message from one source adapter to one target adapter
  - Sending a message from one source adapter to multiple target adapters
  - Multithreaded message delivery, that is, multiple workers
  - With destination logical identifier set to NONE in the message, so that the kernel’s configuration file is used to determine the destination logical identifier based on body category, body type, and source logical identifier
  - Push model of delivery
  - Tracing enabled

  Note: See “Appendix C. Message headers” on page 83. It contains the MQSeries Adapter Kernel message header fields that the kernel populates and processes.
- With the prerequisites shown in “Hardware” on page 23 and “Software” on page 24.
- Using the configuration file, not LDAP, to contain the configuration.

MQSeries:

- Not using MQSeries clusters.

  Note: See “Appendix C. Message headers” on page 83. It contains the MQSeries Adapter Kernel message header fields that the kernel populates and processes.

MQSeries Integrator:

- MQSeries Adapter Kernel and MQSeries can route and deliver the message to MQSeries Integrator. See MQSeries Integrator information to determine its capabilities to broker these messages.
- Sending messages from the source side of the kernel, through MQSeries and MQSeries Integrator version 2, and routing directly to the target side of the kernel. Within MQSeries Integrator, the message flow is configured to route statically. All messages arriving on the MQInput node of the flow are routed directly to a specific MQOutput queue.

  Note: See “Appendix C. Message headers” on page 83. It contains the MQSeries Adapter Kernel message header fields that the kernel populates and processes.

Appendix C. Message headers

MQSeries Adapter Offering uses several message headers. See “Message and message format” on page 10 for which headers are used under which circumstances.

This appendix lists and describes the message header fields.

MQSeries message descriptor header

Content of fields is determined by MQSeries. MQSeries Adapter Offering puts messages onto queues as determined by message control values. See “Message control values” on page 14 for details.

Table 6. MQSeries header

Section or field	**Meaning or usage**
Revision	Fixed.
UniqueID	Each message has a unique identifier.
TransactionID	A message and its reply share the same transaction identifier.
MessageType	Reserved use.
SourceLogicalID	Logical identifier of the source application.
DestinationLogicalID	Logical identifier of the target application.
RespondToLogicalID	A logical identifier to which the reply message is to be sent.
CorrelationID	Reserved use.
GroupStatus	Reserved use.
ProcessingCategory	Reserved use.
QosPolicy	Reserved use.
DeliveryCategory	Reserved use.
AckRequested	Determines whether the source application requests a reply or not.
PublicationTopic	Reserved use.
SessionID	Reserved use.
EncryptionStatus	Reserved use.
TimeStampCreated	Time and date when the message was created.

Table 6. MQSeries header (continued)

TimeStampExpired	Time and date after which the message is no longer meaningful.
Size	Reserved use.
BodyCategory	Represents the message's application type, for example, OAG or RosettaNet.
BodyType	Represents the specific purpose of the message, for example, add sales order or synchronize inventory.
BodySecondaryType	Reserved.
UserArea	General area for user data.
BodyData	Message body data.

MQSeries without MQSeries Integrator

The kernel header values and the body data are put into an XML document. The following is an example of the DTD that describes the XML document:

```
<!ELEMENT EPICHEADER (HEADER, EPICBODY,USERAREA*)>
<!ELEMENT HEADER (#PCDATA)>
<!ATTLIST HEADER Revision CDATA #FIXED "001">
<!ATTLIST HEADER UniqueID CDATA #REQUIRED>
<!ATTLIST HEADER TransactionID CDATA #REQUIRED>
<!ATTLIST HEADER MessageType CDATA #REQUIRED>
<!ATTLIST HEADER SourceLogicalID CDATA #REQUIRED>
<!ATTLIST HEADER DestinationLogicalID CDATA #REQUIRED>
<!ATTLIST HEADER RespondToLogicalID CDATA #IMPLIED>
<!ATTLIST HEADER CorrelationID CDATA #IMPLIED>
<!ATTLIST HEADER GroupStatus CDATA #IMPLIED>
<!ATTLIST HEADER ProcessingCategory CDATA #IMPLIED>
<!ATTLIST HEADER QosPolicy CDATA #IMPLIED>
<!ATTLIST HEADER DeliveryCategory CDATA #IMPLIED>
<!ATTLIST HEADER AckRequested CDATA #IMPLIED>
<!ATTLIST HEADER PublicationTopic CDATA #IMPLIED>
<!ATTLIST HEADER SessionID CDATA #IMPLIED>
<!ATTLIST HEADER EncryptionStatus CDATA #IMPLIED>
<!ATTLIST HEADER TimeStampCreated CDATA #REQUIRED>
<!ATTLIST HEADER TimeStampExpired CDATA #REQUIRED>
<!ATTLIST HEADER Size CDATA #IMPLIED>
<!ELEMENT EPICBODY (#PCDATA)> <!-- The data will be escaped -->
<!ATTLIST EPICBODY Size CDATA #IMPLIED>
<!ATTLIST EPICBODY BodyType CDATA #REQUIRED>
<!ATTLIST EPICBODY BodyCategory CDATA #REQUIRED>
<!ATTLIST EPICBODY BodySecondaryType CDATA #IMPLIED>
<!ELEMENT USERAREA (#PCDATA) >
```

MQSeries Integrator version 1 header

MQSeries Integrator version 1 header, RFH1, consists of the following items:

1. Fixed portion
2. Neon header
3. Data section, which contains the kernel header and message body data

Table 7. MQSeries Integrator version 1 header — RFH1

Section or field	Meaning or usage
Fixed portion	Used as specified in MQSeries Integrator version 1.1.
Neon header	Follows Neon header format.
OPT_APP_GRP	SourceLogicalId value. Taken from the kernel header.
OPT_MSG_TYPE	BodyCategory+BodyType. Derived from the kernel header. Example: If the BodyCategory is OAG and the BodyType is SyncItem, then the value is OAG+SyncItem.
Data section	Consists of the kernel header values followed by the message body data.
Kernel header	Kernel header is enclosed within the tags `<EPICHEADER>`*header*`</EPICHEADER>`. Kernel header values are in XML syntax. Only attributes with values are present. The actual data is not on separate lines. Example of the format of a value: `<MessageType>`*value*`</MessageType>`.
MessageType	Reserved use.
SourceLogicalID	Logical identifier of the source application.
DestinationLogicalID	Logical identifier of the target application.
RespondToLogicalID	Logical identifier to which the reply message is to be sent.
TimeStampCreated	Time and date when the message was created.
TimeStampExpired	Time and date after which the message is no longer meaningful.
TransactionID	A message and its reply share the same transaction identifier.
UniqueID	Each message has a unique identifier.
AckRequested	Determines whether the source application requests a reply.

Table 7. MQSeries Integrator version 1 header — RFH1 (continued)

ProcessingCategory	Reserved.
BodyCategory	Represents the message's application type, for example, OAG or RosettaNet.
BodyType	Represents the specific purpose of the message, for example, add sales order or synchronize inventory.
BodySecondaryType	Reserved.
UserArea	User integration specific application data.
MsgHeaderVersion	Kernel header version (reserved).
CorrelationID	User integration specific.
GroupStatus	User integration specific.
QosPolicy	Reserved.
DeliveryCategory	Reserved.
PublicationTopic	Reserved.
SessionID	Reserved.
EncryptionStatus	Reserved.
Message body data	Message body data.

MQSeries Integrator version 2 header

MQSeries Integrator version 2 header, RFH2, consists of the following items:

1. Fixed portion
2. <mcd> folder — message content descriptor
3. <usr> folder — application (user) defined properties
4. Data section, which contains the kernel header and message body data

Table 8. MQSeries Integrator version 2 header — RFH2

Section or field	**Meaning or usage**
Fixed portion	Used as specified in MQSeries Integrator version 2.
<mcd>	XML if message is XML. Follow MQSeries Integrator version 2 rules.
set	Not used by the kernel.
type	Not used by the kernel.
format	XML if message is XML. Follow MQSeries Integrator version 2 rules.

Table 8. MQSeries Integrator version 2 header — RFH2 (continued)

<usr> folder — application (user) defined properties	Consists of the kernel header values.
Kernel header	Only attributes with values are present. The actual data is not on separate lines.
SourceLogicalID	Logical identifier of the source application.
DestinationLogicalID	Logical identifier of the target application.
MessageType	Reserved use.
RespondToLogicalID	A logical identifier to which the reply message is to be sent.
TimeStampCreated	Time and date when the message was created.
TimeStampExpired	Time and date after which the message is no longer meaningful.
TransactionID	A message and its reply share the same transaction identifier.
UniqueID	Each message has a unique identifier.
ProcessingCategory	Reserved.
BodyCategory	Represents the message’s application type, for example, OAG or RosettaNet.
BodyType	Represents the specific purpose of the message, for example, add sales order or synchronize inventory.
BodySecondaryType	Reserved.
AckRequested	Determines whether the source application requests a reply.
UserArea	User integration specific application data.
MsgHeaderVersion	Kernel header version (reserved).
CorrelationID	User integration specific.
GroupStatus	User integration specific.
QosPolicy	Reserved.
DeliveryCategory	Reserved.
PublicationTopic	Reserved.
SessionID	Reserved.
EncryptionStatus	Reserved.
Data section	Message body data.

Appendix D. Sample of the configuration file

This section lists the version of the aqmconfig.xml file that was current at the time of this publication. “Sample of a minimum configuration file” on page 93 lists the version of the aqmconfig.minimum.xml file that was current at the time of this publication. See the aqmconfig.xml and aqmconfig.minimum.xml files in the kernel installation’s samples directory for the most recent version; the examples listed here are possibly out of date.

See “The configuration file” on page 47 for information on interpreting and editing the configuration file.

Several application identifiers are included in this example configuration file. A set of entries is listed under each application identifier. The sample configuration file contains the following application identifiers:

- TEST1
- TEST1Daemon
- TEST2
- TEST3
- TraceClient
- TraceServer

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- aqmconfig.xml 1.00 00/10/30                                           -->
<!-- Used for MQSeries Adapter Kernel -->
<!-- Sample AQM Configuration.            -->
<!--  -->
<!-- Copyright (c) 2000 International Business Machines. All Rights Reserved. -->
<!--  -->
<!-- This configuration file is as an example only. -->
<!--  -->
<!-- IBM MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS -->
<!-- SAMPLE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE -->
<!-- IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR -->
<!-- PURPOSE, OR NON-INFRINGEMENT. -->
<!--  -->
<!-- CopyrightVersion 1.0 -->
<!--     -->
   <Epic o="ePIC">
      <!-- If getObject is called this indicates the top level directory -->
      <!-- where the JNDI file system context will retrieve objects from -->
      <!-- This defaults to the current directory if this key is not present -->
      <!-- All applications share this context root. -->
      <context>file:///epic/configContext</context>
      <!-- Example using a drive letter 'c' -->
      <!--
      <context>file://c:/E/runtimefiles</context>
      -->
      <ePICApplications o="ePICApplications">
         <!-- The following is for sample Test Application ID: TEST1 with a  -->
         <!-- sample AdapterDaemon named TEST1Daemon  -->
         <ePICApplication epicappid="TEST1">
     <!--  Audit Logging on/off.  Requires WSB2BI product. -->
     <!--  If no entry defaults to false. -->
           <epiclogging>false</epiclogging>
     <!--  Tracing on/off.  If no entry defaults to false. -->
           <epictrace>false</epictrace>
     <!-- Trace levels - Uses the jlog com.ibm.logging.IRecordType constants, -->
     <!-- common constants: -->
```

```
<!-- 0=TYPE_NONE (No messages), 1=TYPE_INFO, 512=TYPE_ERROR_EXC (Exceptions), -->
<!-- 513=TYPE_INFO | TYPE_ERROR_EXC, -1=TYPE_ALL (All possible messages). -->
<!-- No entry defaults to TYPE_NONE -->
       <epictracelevel>-1</epictracelevel>
<!-- Name of the Trace application id.  Will be used for -->
<!-- trace configuration information.  Defaults to TraceClient -->
       <epictraceclientid>TraceClient</epictraceclientid>
<!-- When processing messages into the application. -->
<!-- LogonInfo class name used for connecting to an application. -->
  <!-- Will be used by the AdapterDaemon.  If no entry will default -->
<!-- to com.ibm.epic.adapters.eak.adapterdaemon.EpicLogonDefault. -->
<epiclogoninfoclassname>com.ibm.epic.adapters.eak.adapterdaemon.EpicLogonDefault
</epiclogoninfoclassname>
       <AdapterRouting cn="epicadapterrouting">
          <!-- MQSeries Q Manager for this application use, no entry -->
          <!-- uses the default Q Manager.  A value of DEFAULT means -->
          <!-- use the default Q Manager. -->
          <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
          <!-- Use the remote Q Manager for sending messages.  Remote queue -->
          <!-- definitions are not required.  true - use remote Q Manager, -->
          <!-- false - do not use remote Q Manager.  No entry defaults to false -->
          <epicuseremotequeuemanagertosend>false</epicuseremotequeuemanagertosend>
          <!-- MQSeries Client hostname for where the MQSeries server -->
          <!-- resides for TEST1.  Required if using MQSeries Client -->
          <!--
          <epicmqppqueuemgrhostname>localhost</epicmqppqueuemgrhostname>
   -->
          <!-- MQSeries Client port to use for where the MQSeries server -->
          <!-- resides for TEST1.  No entry defaults to MQSeries default -->
          <!--
          <epicmqppqueuemgrportnumber>1414</epicmqppqueuemgrportnumber>
   -->
          <!-- MQSeries Client channel name to use for the MQSeries server, required -->
          <!--
          <epicmqppqueuemgrchannelname>xyz</epicmqppqueuemgrchannelname>
   -->
          <!-- JMS example for TEST1.  Refers to the JMS Connection factory name. -->
          <!-- Requires the attribute describing the object plus the attributes value. -->
          <!-- For JMS the attribute is 'cn'. -->
          <!--
          <epicjmsconnectionfactoryname>cn=QCFTEST1</epicjmsconnectionfactoryname>
   -->
          <ePICBodyCategory epicbodycategory="DEFAULT">
             <ePICBodyType epicbodytype="DEFAULT">
<!-- Contains the Command selection criteria when processing -->
<!-- a message into an application.  Will be used by the -->
<!-- AdapterDaemon - Command to invoke. -->
                <epiccommandclassname>com.ibm.epic.adapters.eak.samples.SampleCAdapterWrapper
                </epiccommandclassname>
<!-- Default destinations to send messages to. -->
        <!-- Single destination. -->
                <epicdestids>TEST2</epicdestids>
        <!-- Multiple destinations. -->
                <!--
                <epicdestids>
                     <Value>TEST2</Value>
                     <Value>TEST3</Value>
                </epicdestids>
                -->
                <!-- Receive transport communication mode this application -->
                <!-- wants for receiving messages. -->
                <!-- For MQSeries normal mode use MQPP. -->
                <!-- For MQSeries using an RFH1 header format use MQRFH1, when using
                     MQSeries Integrator V1 -->
                <!-- For MQSeries using an RFH2 header format use MQRFH2, when using
                     MQSeries Integrator V2 -->
                <!-- For file normal mode use FILE. -->
                <epicreceivemode>MQPP</epicreceivemode>
                <!-- How to format the message for the receive mode. -->
                <!-- Entry is the class name of the formatter which -->
                <!-- must be for the receive mode -->
                <!-- Receive modes MQPP, MQRFH1, MQRFH2, FILE have -->
                <!-- default receive modes -->
                <epicmessageformatter>com.ibm.epic.adapters.eak.nativeadapter.MQNMBDFormatter
                </epicmessageformatter>
 <!-- JMS formatter for mode for MQSeries provider implementation -->
 <!--
                <epicmessageformatter>com.ibm.epic.adapters.eak.nativeadapter.JMSNMRFH2Formatter
                </epicmessageformatter>
 -->
 <!-- Recieve Time out in milliseconds ie. 1000 = 1 second, -->
 <!-- -1 means never ending.  No entry defaults to 0 -->
 <!-- milliseconds.  Used when receiving messages. -->
                <epicreceivetimeout>30000</epicreceivetimeout>
 <!-- MQSeries queue for this application to receive messages -->
```

```
    <!-- from for receive modes MQPP, MQRFH1, MQRFH2 -->
                   <epicreceivemqppqueue>TEST1AIQ</epicreceivemqppqueue>
    <!-- MQSeries queue required by the AdapterWorker when -->
    <!-- errors encountered processing a message -->
    <!-- for receive modes MQPP, MQRFH1, MQRFH2 -->
                   <epicerrormqppqueue>TEST1AEQ</epicerrormqppqueue>
    <!-- MQSeries reply queue required for synchronous request/replies -->
    <!-- for receive modes MQPP, MQRFH1, MQRFH2 -->
                   <epicreplymqppqueue>TEST1RPL</epicreplymqppqueue>
    <!-- JMS recieve mode, refers to the JMS queue object name for -->
    <!-- this application to receive messages from. -->
                   <!-- Requires the attribute describing the object plus the attributes value. -->
                   <!-- For JMS the attribute is 'cn'. -->
                   <epicjmsreceivequeuename>cn=TEST1AIQ</epicjmsreceivequeuename>
    <!-- JMS recieve mode, refers to the JMS queue object name for -->
    <!-- errors required by the AdapterWorker when errors -->
    <!-- encountered processing a message. -->
                   <!-- Requires the attribute describing the object plus the attributes value. -->
                   <!-- For JMS the attribute is 'cn'. -->
                   <epicjmserrorqueuename>cn=TEST1AEQ</epicjmserrorqueuename>
    <!-- JMS recieve mode, refers to the JMS queue object name for -->
    <!-- the reply queue, required for synchronous request/replies -->
                   <!-- Requires the attribute describing the object plus the attributes value. -->
                   <!-- For JMS the attribute is 'cn'. -->
                   <epicjmsreplyqueuename>cn=TEST1RPL</epicjmsreplyqueuename>
    <!-- In FILE receive mode, directory for this application to receive messages from -->
                   <epicreceivefiledir>./TEST1AID</epicreceivefiledir>
    <!-- In FILE receive mode, interim directory for this application to -->
    <!-- hold received messages until committed. -->
                   <epiccommitfiledir>./TEST1ACD</epiccommitfiledir>
    <!-- In FILE receive mode, directory for this application to put error messages -->
    <!-- File receive mode, directory required by the AdapterWorker when -->
    <!-- errors encountered processing a message -->
                   <epicerrorfiledir>./TEST1AED</epicerrorfiledir>
                </ePICBodyType>
             </ePICBodyCategory>
          </AdapterRouting>
       </ePICApplication>
       <!-- The following is for sample AdapterDaemon 'TEST1Daemon' -->
       <!-- for the 'TEST1' application  -->
       <ePICApplication epicappid="TEST1Daemon">
          <epictrace>false</epictrace>
          <epictracelevel>-1</epictracelevel>
          <ePICAdapterDaemonExtensions cn="epicappextensions">
      <!-- Dependency appid, if no entry then will default -->
      <!-- to the application id of the daemon.     -->
             <epicdepappid>TEST1</epicdepappid>
      <!-- Minimum number of workers.  If no entry defaults to 1. -->
             <epicminworkers>1</epicminworkers>
          </ePICAdapterDaemonExtensions>
       </ePICApplication>
       <!-- The following is for Test Application ID: TEST2 -->
       <!-- Refer to TEST1 for explanations and possible additional entries. -->
       <ePICApplication epicappid="TEST2">
          <epictrace>true</epictrace>
          <epictracelevel>512</epictracelevel>
          <AdapterRouting cn="epicadapterrouting">
             <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
             <ePICBodyCategory epicbodycategory="DEFAULT">
                <ePICBodyType epicbodytype="DEFAULT">
                   <epiccommandclassname>com.ibm.epic.adapters.eak.test.InstallVerificationTest
                   </epiccommandclassname>
                   <epicreceivemode>MQPP</epicreceivemode>
                   <epicreceivemqppqueue>TEST2AIQ</epicreceivemqppqueue>
                   <epicerrormqppqueue>TEST2AEQ</epicerrormqppqueue>
                   <epicreplymqppqueue>TEST2RPL</epicreplymqppqueue>
                </ePICBodyType>
             </ePICBodyCategory>
          </AdapterRouting>
       </ePICApplication>
       <!-- The following is for Test Application ID: TEST3 -->
       <!-- Refer to TEST1 for explanations and possible additional entries. -->
       <ePICApplication epicappid="TEST3">
          <AdapterRouting cn="epicadapterrouting">
             <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
             <ePICBodyCategory epicbodycategory="DEFAULT">
                <ePICBodyType epicbodytype="DEFAULT">
                   <epicdestids>TEST1</epicdestids>
                   <epicreceivemode>MQPP</epicreceivemode>
                   <epicreceivemqppqueue>TEST3AIQ</epicreceivemqppqueue>
                </ePICBodyType>
             </ePICBodyCategory>
          </AdapterRouting>
       </ePICApplication>
<!-- The following is for sample Trace Client Application ID: TraceClient -->
```

```
<!-- Contains the TraceClient configuration information for doing tracing. -->
<!-- This is the application id value in the 'epictraceclientid' element -->
<!-- configured for the application wanting to do tracing -->
       <ePICApplication epicappid="TraceClient">
          <ePICTraceExtensions cn="epicappextensions">
      <!-- Dependency Trace Server application id used for SocketHandler -->
      <!-- and ENAHandler (uses MQSeries), defaults to TraceServer -->
             <epicdepappid>TraceServer</epicdepappid>
             <!-- Write messages synchronously (true) or asynchronously (false), -->
             <!-- defaults to false (write messages asynchronously).  This is    -->
             <!-- used when giving the messages to the handlers. -->
             <epictracesyncoperation>false</epictracesyncoperation>
      <!-- Default Trace message file to use if none passed in to the -->
         <!-- writeTrace method call. Defaults to this file if not indicated -->
             <epictracemessagefile>com.ibm.epic.trace.client.TraceMessage</epictracemessagefile>
      <!-- Handlers to load.  Handlers do the actual processing of the -->
      <!-- Trace message.  If the default trace client id 'TraceClient' -->
      <!-- is used then the handler defaults to the -->
      <!-- com.ibm.logging.ConsoleHandler.  If the default trace client -->
      <!-- id 'TraceClient' is not used, the handler has to be specified. -->
      <!-- A Single Trace Handler -->
             <epictracehandler>com.ibm.logging.ConsoleHandler</epictracehandler>
      <!-- Multiple Trace Handlers -->
      <!--
             <epictracehandler>
                <Value>com.ibm.logging.ConsoleHandler</Value>
                <Value>com.ibm.logging.SocketHandler</Value>
             </epictracehandler>
      -->
             <!-- Handler definitions.  Available definitions depend on the -->
             <!-- handler.  Formatters are used for formatting the trace message.-->
             <ePICTraceHandler epictracehandler="com.ibm.logging.ConsoleHandler">
                <!-- ConsoleHandler formatter to use, defaults to this formatter if none provided. -->
                <epictraceformatter>com.ibm.epic.trace.client.EpicTraceFormatter</epictraceformatter>
             </ePICTraceHandler>
             <ePICTraceHandler epictracehandler="com.ibm.logging.FileHandler">
                <!-- FileHandler formatter to use, defaults to this formatter if none provided. -->
                <epictraceformatter>com.ibm.epic.trace.client.EpicTraceFormatter</epictraceformatter>
                <!-- Trace filename to use, defaults to trc.log in the current directory. -->
                <epictracefilename>trc.log</epictracefilename>
             </ePICTraceHandler>
             <ePICTraceHandler epictracehandler="com.ibm.epic.trace.client.ENAHandler">
                <!-- ENAHandler formatter to use, defaults to this formatter if none provided. -->
                <epictraceformatter>com.ibm.epic.trace.client.EpicXMLFormatter</epictraceformatter>
             </ePICTraceHandler>
             <ePICTraceHandler epictracehandler="com.ibm.logging.SocketHandler">
                <!-- SocketHandler formatter to use, defaults to this formatter if none provided. -->
                <epictraceformatter>com.ibm.epic.trace.client.EpicXMLFormatter</epictraceformatter>
             </ePICTraceHandler>
          </ePICTraceExtensions>
       </ePICApplication>
<!-- The following is for sample Trace Server Application ID: TraceServer -->
<!-- Contains the TraceServer configuration information. -->
<!-- This is the application id pointed to by the trace client -->
<!-- epicdepappid value.  Definitions are similar to TraceClient example. -->
       <ePICApplication epicappid="TraceServer">
          <AdapterRouting cn="epicadapterrouting">
             <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
             <ePICBodyCategory epicbodycategory="DEFAULT">
                <ePICBodyType epicbodytype="DEFAULT">
                   <epicreceivemode>MQPP</epicreceivemode>
                   <epicreceivemqppqueue>TraceServerAIQ</epicreceivemqppqueue>
                </ePICBodyType>
             </ePICBodyCategory>
          </AdapterRouting>
          <ePICTraceExtensions cn="epicappextensions">
      <!-- Write messages synchronously/asynchronously (true/false (default)). -->
             <epictracesyncoperation>false</epictracesyncoperation>
      <!-- Trace message file.  Defaults to this file if not indicated -->
             <epictracemessagefile>com.ibm.epic.trace.server.TraceServerMessage</epictracemessagefile>
             <!-- Handlers to load, for multiple handlers see TraceClient example. -->
             <!-- If the default trace server id 'TraceServer' is used then the handler -->
             <!-- defaults to the com.ibm.logging.MultiFileHandler. -->
             <!-- Note: Do not use SocketHandler or ENAHandler for the trace server. -->
             <epictracehandler>com.ibm.logging.MultiFileHandler</epictracehandler>
      <!-- Handler definitions for com.ibm.logging.SocketHandler -->
      <!-- Formatter to use, defaults to this formatter if none provided.-->
             <ePICTraceHandler epictracehandler="com.ibm.logging.SocketHandler">
 <!-- Entries when using socket handler from the TraceClient and -->
 <!-- starting the Trace Server in socket receive mode. -->
 <!-- SocketHandler host machine, defaults to localhost -->
                <epictracesocketserverhost>localhost</epictracesocketserverhost>
         <!-- SocketHandler port number, defaults to 8181 -->
                <epictraceportnumber>8181</epictraceportnumber>
             </ePICTraceHandler>
```

```
        <!-- Formatter to use, defaults to this formatter if none provided.-->
               <ePICTraceHandler epictracehandler="com.ibm.logging.ConsoleHandler">
           <!-- ConsoleHandler formatter to use, defaults to this formatter if none provided.-->
                  <epictraceformatter>com.ibm.epic.trace.client.ReFormatter</epictraceformatter>
               </ePICTraceHandler>
               <ePICTraceHandler epictracehandler="com.ibm.logging.MultiFileHandler">
   <!-- MultiFileHandler formatter to use, defaults to this formatter if none provided. -->
                  <epictraceformatter>com.ibm.epic.trace.client.ReFormatter</epictraceformatter>
   <!-- MultiFileHandler trace base filename to use, defaults to trc.log in the -->
   <!-- current directory.  The actual filename will be for this -->
   <!-- example trcx.log, where x is a numeric number starting at -->
   <!-- 0 and going up to the number of trace files specified. -->
                  <epictracefilename>trc.log</epictracefilename>
   <!-- MultiFileHandler number of trace files, defaults to 3 -->
                  <epictracefilenumber>3</epictracefilenumber>
   <!-- MultiFileHandler file size in number of bytes, defaults to -->
                  <epictracefilesize>1000000</epictracefilesize>
               </ePICTraceHandler>
            </ePICTraceExtensions>
         </ePICApplication>
      </ePICApplications>
   </Epic>
```

Sample of a minimum configuration file

This section provides an example of a minimum configuration file for use with MQSeries Adapter Kernel. See “Adding adapter information to the configuration” on page 58 for information about the minimum configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- aqmconfig.minimum.xml 1.00 00/11/07                                      -->
<!-- Used for MQSeries Adapter Kernel -->
<!-- Sample AQM Configuration showing a minimum configuration for the         -->
<!-- following conditions:                                                     -->
<!-- 1) Going from applicationid TEST1 to TEST2.  TEST1 is not receiving  -->
<!--    messages. -->
<!-- 2) TEST2 has no special application requirements.  -->
<!-- 3) TEST2 is using 1 worker.                        -->
<!-- 4) Using MQSeries with the default QManager installed on each machine.   -->
<!--    and using default format.                                             -->
<!-- 5) No specific body category and body type being used.                   -->
<!-- 6) Using default tracing to the console.   -->
<!--  -->
<!--  -->
<!--  -->
<!-- Copyright (c) 2000 International Business Machines. All Rights Reserved. -->
<!--  -->
<!-- This configuration file is as an example only. -->
<!--  -->
<!-- IBM MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS -->
<!-- SAMPLE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE -->
<!-- IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR -->
<!-- PURPOSE, OR NON-INFRINGEMENT. -->
<!--  -->
<!-- CopyrightVersion 1.0 -->
<!--      -->
   <Epic o="ePIC">
      <ePICApplications o="ePICApplications">
         <!-- The following is for sample Test Application ID: TEST1  -->
         <ePICApplication epicappid="TEST1">
     <!--  Tracing on/off.  If no entry defaults to false. -->
            <epictrace>false</epictrace>
     <!-- Trace levels - 512=TYPE_ERROR_EXC (Exceptions),-1=TYPE_ALL (All possible messages). -->
            <epictracelevel>0</epictracelevel>
            <AdapterRouting cn="epicadapterrouting">
               <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
               <ePICBodyCategory epicbodycategory="DEFAULT">
                  <ePICBodyType epicbodytype="DEFAULT">
     <!-- Default destinations to send messages to. -->
                     <epicdestids>TEST2</epicdestids>
                  </ePICBodyType>
               </ePICBodyCategory>
            </AdapterRouting>
         </ePICApplication>
         <!-- The following is for Test Application ID: TEST2 -->
         <ePICApplication epicappid="TEST2">
            <epictrace>false</epictrace>
            <epictracelevel>512</epictracelevel>
         <AdapterRouting cn="epicadapterrouting">
            <epicmqppqueuemgr>DEFAULT</epicmqppqueuemgr>
            <ePICBodyCategory epicbodycategory="DEFAULT">
               <ePICBodyType epicbodytype="DEFAULT">
   <!-- AdapterDaemon - Command to invoke. -->
                  <epiccommandclassname>com.ibm.epic.adapters.eak.samples.SampleCAdapterWrapper
                  </epiccommandclassname>
                  <epicreceivemode>MQ</epicreceivemode>
   <!-- Recieve Time out in milliseconds ie. 1000 = 1 second, -->
   <!-- -1 means never ending.  No entry defaults to 0. -->
   <!-- milliseconds.  Used when receiving messages. -->
                  <epicreceivetimeout>30000</epicreceivetimeout>
                  <epicreceivemqppqueue>TEST2AIQ</epicreceivemqppqueue>
                  <epicerrormqppqueue>TEST2AEQ</epicerrormqppqueue>
                  <epicreplymqppqueue>TEST2RPL</epicreplymqppqueue>
               </ePICBodyType>
            </ePICBodyCategory>
         </AdapterRouting>
      </ePICApplication>
   </ePICApplications>
</Epic>
```

Appendix E. Sample of the setup file

The following is an example of the aqmsetup file, which defines several of the kernel’s initial configuration values, including several environment variables. See “The setup file” on page 47 for additional information about this file. The aqmsetup file is located in the samples directory of the kernel’s root installation directory.

```
#
# aqmsetup 1.00 00/08/03
# Sample AQM Adapter runtime parameter configuration file entries.
#
# Copyright (c) 2000 International Business Machines. All Rights Reserved.
#
# This configuration file is as an example only.
#
# IBM MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS
# SAMPLE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
# IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
# PURPOSE, OR NON-INFRINGEMENT.
#
# CopyrightVersion 1.0
#
#
# Pound (#) signs are comments.
#
####################################################################
#
# Use WebSphere Business-to-Business Integrator product 5799-RNK LDAP Directory
# Services or configuration file. No entry defaults to true (use configuration
# file). To use the WebSphere BtoB Integrator directory service set the value
# to false.  Refer to the WebSphere BtoB Integrator documentation for
# specifics on using the directory service.
#AdapterDirectoryUseFileFlag=true
# When using WebSphere Business-to-Business Integrator product 5799-RNK LDAP
# Directory Services this additional entry is required. Refer to the
# WebSphere BtoB Integrator documentation for specifics on using the
# directory service.
#DirectoryServices=G:/mqak1.0.1/bld/TestSetup/DirectoryServices.properties
# Location of configuration file aqmconfig.xml when not using
# the WebSphere Business-to-Business Integrator product 5799-RNK LDAP Directory
# Services.
# No entry defaults to current directory.
#AQMConfig=G:/mqak1.0.1/bld/TestSetup
#
####################################################################
####################################################################
# XML DTD Catalogs and Directories - where to locate DTD's if not
# in the current directory.
# Format: XML_DTD_DIRECTORY_x=ddd where x is a numeric suffix to
# be incremented for each key and ddd is the directory.
# The numeric suffix's must start with 1 and be contiguous.
####################################################################
XML_DTD_DIRECTORY_1=G:/Code-Drop3/runtimefiles/oag
#XML_DTD_DIRECTORY_2=ChangeToDestDir/runtimefiles
#
####################################################################
# Java JNI Environment Variables for C Interface for increasing
# the amount of memory used.  This applies to when a C module
# is instantiating a JVM.  When a C Interface is being called
# from within JAVA the JVM is already established.
####################################################################
# The JDK version number being used. 0x00010001 (version 1.1
# equates to 65537, no entry defaults to version 1.1
# Version 1.2 is not supported.
#AQM_JNI_VERSION=65537
# The stack memory is used for holding local function, function
# parameters, local variable references.
# Native stack is used for non-Java calls from within Java such
# as to C code.  Stack size in bytes to use.
# Default is 128 kilobytes on NT.
#AQM_JNI_NATIVESTACKSIZE=1048576
# Java stack is for Java method calls and local variables.
# Stack size in bytes to use.
# Default is 400 kilobytes on NT.
#AQM_JNI_JAVASTACKSIZE=4194304
# The heap memory is used for storing instantiated Java objects
# Minimum heap size in bytes to start with.
# Default is 1 megabyte on NT.
#AQM_JNI_MINHEAPSIZE=16777216
# Maximum heap size in bytes which can be used.
# Default is 16 megabytes on NT.
#AQM_JNI_MAXHEAPSIZE=268435426
#
####################################################################
#  Designate end of configuration file
####################################################################
*ENDCFG
```

Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

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

AIX
AS/400
IBM
MQSeries
OS/400
RISC System/6000
RS/6000
WebSphere

Lotus and LotusScript are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

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

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

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

Glossary

The glossary contains *key* terms and their meanings as used in MQSeries Adapter Kernel documentation.

If a particular concept or term appears in one section only, it is possibly not included in the glossary. It can, however, possibly be found in the "Index" on page 105.

The glossary does not include terms for other IBM products such as MQSeries.

adapter. The output of MQSeries Adapter Builder. Typically, the user builds each adapter to be specific to *one message type* that is sent from or to an application. Thus, the adapters themselves are not part of MQSeries Adapter Offering. An adapter consists of C source code that compiles to a shared library. When the adapters and MQSeries Adapter Kernel run together, they perform the run-time functionality of MQSeries Adapter Offering. Depending on how it was modeled by the user in MQSeries Adapter Builder, the adapter can contain a wide variety of functionalities such as controlflow, dataflow, sequential navigation, conditional branching including decision and iteration, data typing, storing data context, transformation of data elements, logical operations and custom code. You can reuse adapters that you have created.

See "message type" on page 103, "source application" on page 104, and "target application" on page 104.

adapter daemon. Executable software that is part of the kernel. The adapter daemon is used only in the push delivery model. Its purpose is to instantiate the workers. After it is started, the adapter daemon remains active. For each target application, there can be one or more adapter daemons.

In some cases, the adapter daemon performs the role of a target application. It performs the required functionality, for example, using a target adapter to send an e-mail message or to write a record to a file.

aqmconfig.xml file. See "configuration file" on page 102.

aqmsetup file. See "setup file" on page 103.

application logical identifier. An identifier that represents the application with which an adapter (either a source adapter or a target adapter) is associated. See "source logical identifier" on page 104 and "target logical identifier" on page 104.

application-neutral format. See "integration formatted message" on page 102.

application-specific interface. An interface that is developed outside of MQSeries Adapter Offering for one of the following purposes:

- To enable the source adapter to acquire a message from the source application.
- To enable the target application to acquire a message from the target adapter.

BOD. Business Object Document. A representation of a standard business process that flows within an organization or between organizations. Examples are add purchase order, show product availability, and add sales order. BODs are defined by the OAG using XML. See "OAG" on page 103 and "XML" on page 104.

BODs can be used by MQSeries Adapter Offering to define message bodies in its integration formatted messages.

body category. Data contained in a message that represents the message's application type, for example, OAG or RosettaNet. It belongs to the set of message control values. See "message control values" on page 103.

Body category also helps specify the message type. See "message type" on page 103.

body type. Data contained in a message that represents the specific purpose of the message, for example, add sales order or synchronize inventory. It belongs to the set of message control values. See "message control values" on page 103.

Body type also helps specify the message type. See "message type" on page 103.

configuration file. The `aqmconfig.xml` file, which contains most of the kernel's configuration values. See "The configuration file" on page 47 for details.

communications message. Any communications transport-specific information plus the message holder object, converted into a messaging format specific to the communications transport being used.

communication mode. The mode used by the kernel to transport the message and to perform broker services.

destination logical identifier. A value that represents the target application. It is used, along with other message control values, by the kernel to route messages and to marshal messages. See "message control values" on page 103.

delivery models. There are two models by which the kernel interfaces to the target application. These two models are:

push The kernel is responsible for initiating and managing delivery of the message to the target application. This model typically does not require changing the target application to support MQSeries Adapter Offering.

pull The target application is responsible for managing the delivery of the message. This model requires changing the target application to support MQSeries Adapter Offering. The target application must manage the kernel's interface to the target application.

dependency application identifier. The name of the application that the worker services. The worker gets the dependency application identifier from the configuration file based on the adapter daemon's name.

DTD. Document Type Definition. In XML, usually a file (or several files used together) that contains a formal definition of a particular type of document. It specifies the names that can be used for elements within the DTD, where elements are allowed to occur within the DTD, and how the elements fit together. In MQSeries Adapter Offering, you can use DTDs to define message bodies. See "XML" on page 104 and "integration formatted message".

error queue. In the terminology of MQSeries Adapter Offering, a message queue that is used when a message that is obtained from a receive queue cannot be processed.

integration formatted message. A message consisting of application data in an application-neutral format for integration. An example is an XML document that the source adapter transforms from the source application's format to XML.

kernel. Synonymous with MQSeries Adapter Kernel.

logon class. A Java class that is specific to each target application and that can be used to help deliver the message to the target application. The logon class is required only when the target adapter must log on to the target application before delivering the message. Each logon class is written by the user. The worker instantiates the logon class. The logon class looks in the configuration file to find the values that the target adapter needs to support the application specific interface to the target application. Typically, those values are logon parameters. Thus, the values are made available to the target adapter.

A dummy logon class that does nothing is provided with the kernel.

message. In MQSeries, including MQSeries Adapter Offering, a collection of data that is sent by one program and intended for another program.

message control values. A collective term for a set of values in the messages (body and headers) and in the configuration file that kernel uses to control the marshaling and routing of messages, and that each adapter uses to control, in part, how it performs its functionality.

message holder object. A container for metadata used by the kernel to encapsulate an integration message and other control data.

message type. A message that is specified by a unique combination of body category and body type. See "body category" on page 101 and "body type" on page 102.

MQSeries Adapter Builder. Software that enables a user to build an adapter for virtually any application by using a graphical user interface (GUI).

MQSeries Adapter Kernel. A set of APIs and several executable programs, in C and Java, and several configuration files. The kernel works with and supports adapters. See "adapter" on page 101. In addition to directly supporting adapters, the kernel performs related functions, among the most important: routing of messages and infrastructure services such as message construction, tracing, and interfacing with MQSeries or other messaging software.

MQSeries Adapter Offering. A set of application integration products that consists of MQSeries Adapter Builder and MQSeries Adapter Kernel.

MQSeries Adapter Kernel native adapter. Synonymous with native adapter.

native adapter. Software used for sending and receiving message holder objects.

OAG. Open Applications Group. A nonprofit industry consortium comprising many prominent stakeholders in the business software component interoperability arena. The OAG defines Business Object Documents (BODs).

pull model of delivery. See "delivery models" on page 102.

push model of delivery. See "delivery models" on page 102.

receive queue. In the terminology of MQSeries Adapter Offering, a message queue that is used as the main input queue, to receive messages. There can be multiple receive queues per target application, but only one receive queue for each combination of application identifier, body category and body type.

reply queue. A message queue that is used to receive replies. It is used with the kernel's `sendRequestResponse` method.

respond to logical identifier. The logical identifier of the application to which replies are to be sent when a reply is requested. It defaults to the source logical identifier in the message.

setup file. A file that contains several of the kernel's initial settings. The default name of the file is `aqmsetup`.

source adapter. An adapter that performs the following tasks:

- Accepts or otherwise acquires structured data from a source application (typically by using an application-specific interface that is developed outside the adapter).
- Processes the structured data according to how the adapter had been modeled.
- Transforms the structured data into an integration message format.
- by using the kernel, puts the message onto a message queue, for delivery to one or more target adapters and thence to the target application.

For each message type, there is one source adapter. Typically, a source application can send multiple message types; therefore, in most cases, a source application is supported by multiple source adapters.

See “adapter” on page 101.

source application. A program that is required to send data over a computer network to a program (known as the target application) that typically resides on another computer.

source logical identifier. A value that represents the source application. It is used, along with other message control values, by the kernel to route messages and to marshal messages. See “message control values” on page 103, “application logical identifier” on page 101, and “target logical identifier”.

source side of the kernel. The part of the kernel functionality that begins when the message is received from the source adapter and that ends when the message is put onto a message queue.

target adapter. An adapter that performs the following tasks:

- Receives a message (from the kernel and MQSeries or other messaging software) that had been sent by a source adapter.
- Processes the integration formatted message according to how the adapter had been modeled.
- Transforms the integration formatted message into an application-specific formatted message that the target application can receive.
- Sends the message to the target application by using an application-specific interface.
- Lets the worker know when it has completed sending the message to the target application, to enable the worker to send an acknowledgment.

If the target application can receive the integration formatted message, then a target adapter is possibly not required.

For each message type, there is one target adapter. Typically, a target application can accept multiple message types; in most cases, therefore, a target application is supported by multiple target adapters.See “adapter” on page 101.

target application. A program that is required to receive data over a computer network from a program (known as the source application) that typically resides on another computer.

target logical identifier. A value that represents the target application associated with a target adapter. See “target logical identifier” and “application logical identifier” on page 101.

target side of the kernel. The part of the kernel functionality that begins when the message is gotten from a message queue and that ends when the message is sent to the target adapter.

trace client. A component of the kernel that writes trace messages.

trace messages. Messages that contain the state of processing a message at a certain point within the kernel. You can use trace messages to help diagnose problems with the kernel or with your adapters.

See “tracing”.

tracing. A collection of processes that the kernel uses to write trace messages. See “trace messages”.

transaction. A set of operations that must be executed as an indivisible unit of work. If all operations that comprise a transaction are successful, the transaction is committed; that is, all of the operations are performed. If one or more of the operations that comprise a transaction fail, the transaction is rolled back; that is, none of the operations are performed.

worker. Software that is part of the kernel. The worker is used only in the push delivery model. The adapter daemon starts and creates the workers. Each worker manages one native adapter. The worker delivers each message to the appropriate target adapter.

XML. Extensible Markup Language. A W3C standard for the representation of data.

Index

Printed in U.S.A.

GC34-5855-04