MQSeries® Adapter Kernel for Multiplatforms

Quick Beginnings

Version 1 Release 1

GC34-5855-06

MQSeries® Adapter Kernel for Multiplatforms

Quick Beginnings

Version 1 Release 1

GC34-5855-06

Note: Before using this information and the product it supports, read the information in "Notices" on page 115.

Seventh Edition (January 2002)

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

This edition replaces GC34-5855-05.

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 29 for details.
3. Install the kernel. See "Installing the kernel" on page 30 for details.
4. Verify the installation. See "Verifying the installation" on page 34 for details.
5. Configure the kernel. See "Configuring the kernel" on page 44 for details.
6. If desired, configure optional software to work with the kernel. See "Configuring MQSeries and MQSeries Integrator" on page 69 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 71 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 79.

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 potentially 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 GC34-5897, 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 as an aid to diagnostics. See "Chapter 5. Using MQSeries Adapter Kernel APIs" on page 77.
- MQSeries Adapter Builder information, including books and help system.
- The MQSeries product family Web site at 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, potentially 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 seventh edition (the current edition) includes the following additions and changes from the sixth edition:

- MQSeries Adapter Kernel now provides national language support for both, Java and C adapters.
- MQSeries Adapter Kernel now supports OS/400 version 5.1. See “Software” on page 24.
- MQSeries Adapter Kernel now supports JDK 1.3 on all platforms. For OS/400, version 5.1 is required.
- MQSeries Adapter Kernel now supports MQSeries Integrator processing for the communication mode MQRFH2. See “XML elements used in the configuration file” on page 51 for details.
- MQSeries Adapter Kernel now provides a configuration checker utility. See “Using the configuration checker utility” on page 68 for details.
- MQSeries Adapter Kernel now contains a command-line utility for stopping an adapter daemon. Therefore, the new parameter `-q qid` is introduced for the start utility. See “Using the start and stop command-line utilities” on page 70 for details.
- New messages have been added.
- How to use a J2EE connector for MQSeries Adapter Kernel microflow is described in “Appendix F. Using a J2EE connector in an MQAK microflow” on page 101.

The sixth edition includes the following additions and changes from the fifth edition:

- Update to the discussion of run-time flow to reflect several changes. See “Run-time flow” on page 10.
- Information on using MQSeries Adapter Kernel with WebSphere® Business Integrator. See “Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server” on page 21 for details.
- Information on the level of national language support provided with different kinds of adapters. See “National language support” on page 22 for details.
- Clarification of installation instructions. See “Installing the kernel” on page 30.
- Information on silent installation. See “Using silent installation” on page 38 for details.
- A conceptual overview of configuration to assist with configuring the kernel. See “Overview of configuration” on page 44 for details.
- Information on new header values. See “MQSeries Adapter Kernel message descriptor header” on page 87 for details.

The fifth edition included the following additions and 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 was 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 49 for details.
- Information on the new MQ and JMS (Java Message Service) communication modes. See "Appendix A. Communications modes" on page 81 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.

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. 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 communicates 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 previous one.

With MQSeries Adapter Offering, you can migrate from using point-to-point integration to using *one-to-any* integration. There are many benefits of one-to-any integration, including 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 at least one adapter to provide the interface between the application environment and the messaging environment. Each adapter is specific to an application and to a message type.

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 by using 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 or Java™ 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 control flow; data flow; sequential navigation; conditional branching, including decision and iteration; data typing; storage of data context; transformation of data elements; transactional control; logical operations; and custom code.

Adapters can be reused.

There are two primary types of adapters:

- Source adapters, for applications that send data.
- Target adapters, for applications that receive 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, 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.

A third type of adapter, the Java service session bean adapter, is used when IBM WebSphere Application Server and enterprise beans are used on the target side of the kernel. WebSphere Application Server's implementation of the Sun Microsystems Enterprise JavaBeans (EJB) specification enables the use of Java service session bean adapters and other enterprise beans. See "Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server" on page 21 and the MQSeries Adapter Builder documentation for more information.

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 Adapter Builder Information Center.

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. It also provides 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.

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 10.

There are two sides of the kernel:

- The *source side*, which begins when the message is received from the source application 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.

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

See Figure 1 on page 4. 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.”

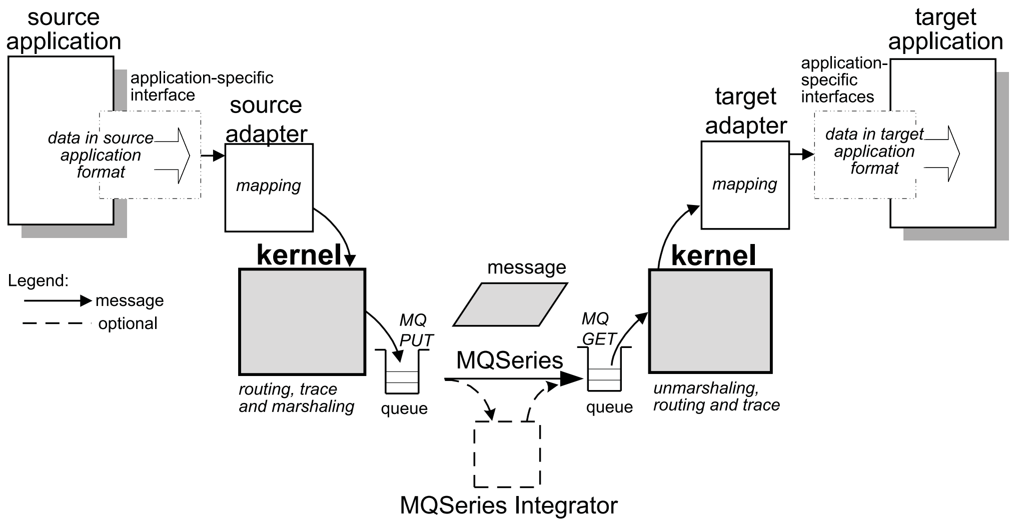
The application-specific interface must be developed outside of the MQSeries Adapter Offering. The exact nature of the application-specific interface depends on the characteristics of the source application or target application. Examples include API calls and user exits, file reads and writes, database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

2. The source adapter performs its function according to how it was built. A typical function is the transformation of data elements, that is, mapping elements from the source application format to an *integration-messaging 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 12.
4. The kernel performs its functions, including *message marshaling*, simple *routing*, and, optionally, *tracing*. See “Message and message format” on page 9, “Routing and delivery” on page 10, and “Tracing” on page 21.

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 20 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 6. Note that the messaging software 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 6. If the final destination, a message queue, has been configured in MQSeries Integrator's rules or message flows, 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.

   In 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 71.

   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 format to elements in the *target application format*.

   Target adapters can be hosted either by an MQSeries Adapter Kernel adapter daemon or by WebSphere Application Server. See "Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server" on page 21 for a discussion of the latter.
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, the message is committed from the message queue. 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 Message 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 85 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 10 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 85 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:

- “Components of the kernel run time”
- “Message and message format” on page 9
- “Routing and delivery” on page 10
- “Run-time flow” on page 10

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 communications 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. If the underlying communication transport mechanism supports transactional messaging, 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.

Having multiple workers enables *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 indicates 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 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.

Java service session bean adapter

A type of Java-language EJB adapter that is hosted in an EJB server such as WebSphere Application Server.

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 Integrator product's LDAP structure. The data controls the following aspects of the kernel's configuration:

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

See "The configuration file" on page 49 for a full description of the configuration file. See the WebSphere Business Integrator documentation for information on configuring that product to work with the kernel.

tracing component

Software that writes trace messages. Most of the kernel's components use the tracing component. See "Tracing" on page 21 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 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 another 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. Although it is strongly recommended, the message body is not required to be in an application-neutral format. The format of the message body can be proprietary or otherwise specialized; however, 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 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 that is 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 are considered to be application data by the communications transport. For more information, see "Appendix A. Communications modes" on page 81. 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 87 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 factors:

- Message queues. On the most basic level, message queues must be configured to support MQSeries Adapter Offering's routing.
- The message-control values in the message. They include the 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 12 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 change. However, the content of the message body data (integration message) cannot determine the routing.
- The message-control values in the kernel's configuration file. The file can specify destination logical identifiers, queue names, and associated target adapters. Determine and modify the configuration by editing this file. See "The configuration file" on page 49 for additional information.
- Optionally, MQSeries Integrator, which 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 6. 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 11 for a diagram of the run-time flow.

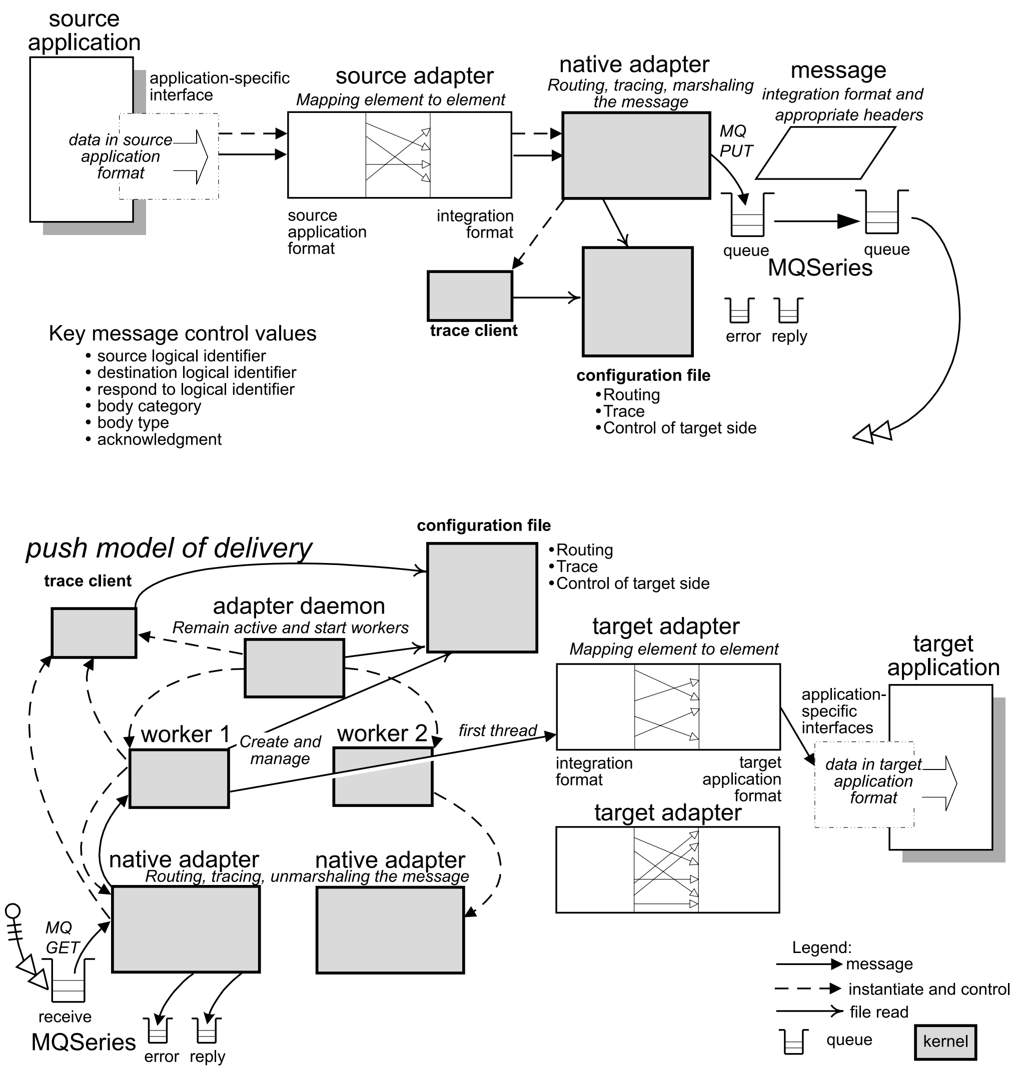


Figure 2. Marshal, send, route, and trace a message — overview.

Source side of the kernel

This section discusses the run-time flow on the source side of the kernel; that is, how data is moved from source application through a source adapter and onto a communications transport. "Target side of the kernel" on page 14 discusses how data is moved from the communications transport to the target.

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 envelop 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-holder 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 21 for an overview of tracing and the *Problem Determination Guide* for detailed information about tracing.

6. The native adapter looks in the message-holder 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 multistage lookup of body category and body type values in the configuration file, in the following order:
     a. For specific body category and body type values.
     b. For a specific body category value and a default body type value.
     c. For a default body category value and a specific body type value.
     d. For default body category and body type values.

   Note: The kernel uses this multistage lookup each time it looks up values in the configuration file.
7. For each destination logical identifier determined in the previous step, the native adapter looks up the *communications 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 Message 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 communications mode, the message structure is different. See "Message and message format" on page 9. For more information about communications modes, see "Appendix A. Communications modes" on page 81.

   Note: If MQSeries Integrator is used, the final destination to which MQSeries Integrator sends the message must use the same communications mode as MQSeries Integrator to receive messages.
8. Based on the communications 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 communications mode is MQPP, parameters include the format and the names of the receive, reply, and error queues. Based on the destination logical identifiers, body category, and body type that are passed to it, the logical message service performs a multistage lookup in the configuration file:
   a. For specific body category and body type values.
   b. For a specific body category value and a default body type value.
   c. For a default body category value and a specific body type value.
   d. For default body category and body type 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 communications mode and format. Each communications mode uses a default format if the format is not otherwise specified. For example, if the communications 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 version 2.
    - Sends the message. For example, if the communications mode is MQRFH2, it puts the message on the appropriate MQSeries message queue.
11. There are two methods that can be used to send the message:
    - If the native adapter uses the `sendMsg` method to send the message, the native adapter does not wait for a response.
    - 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 is configured. Optionally, MQSeries Integrator performs brokering services. See “Role of MQSeries Integrator” on page 6.
13. When the source adapter no longer needs the native adapter, it closes the native adapter to free resources.

Target side of the kernel

This section discusses using stand-alone MQSeries Adapter Kernel to receive and process messages on the target side, and provides a high-level description of using the kernel with WebSphere Application Server. See “Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server” on page 21 for a discussion of using the kernel with JMS, WebSphere Business Integrator’s JMS Listener component, and WebSphere Application Server on the target side of the kernel. This section describes 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 119 for a short description of the models.

Overview of the adapter worker

This section describes the structure and behavior of MQSeries Adapter Kernel adapter workers. One of the assumptions of the MQSeries Adapter Kernel architecture is that target applications do not actively participate in integration data flows with other applications; that is, that applications normally do not actively poll for messages to process. In this case, message data needs to be actively pushed to the target application. Adapter workers push message data to an application or other service by selecting and invoking a range of service interface types.

MQSeries Adapter Kernel can host adapter workers that run in either a stand-alone daemon (adapter daemon) or an Enterprise JavaBeans application server (currently, IBM WebSphere Application Server Advanced Edition). Messages arrive at the adapter worker by different means, depending on which type of target environment is used. If a stand-alone adapter daemon is used, it hosts one or more stand-alone adapter workers that use the native adapter to receive messages. If an EJB server is used, the JMS Listener component receives messages and passes them to a worker message bean (sometimes referred to as a message-driven-bean adapter worker).

Regardless of the target environment used, after the adapter worker receives the message, it then forwards the message to the appropriate target adapter. The target adapter then performs the necessary work to deliver the message to the target application. Target adapters are created to work with specific target applications. The adapter daemon, application server, stand-alone adapter worker, and worker message bean are not specific to any given source or target application.

The adapter worker handles two types of target adapter interfaces: Enterprise Access Builder (EAB) command adapters and EJB service session beans. Each adapter type includes a handler that sets up the appropriate environment, accesses any additional configuration information required for the adapter, and performs other low-level tasks required for the adapter's operation. The handler that is used depends on the adapter type listed in the configuration file. The two types of handlers perform the following additional tasks:

- The EAB handler obtains a logon class, which is used for providing connection information to the target adapter, and initializes the IBM Common Connector Framework (CCF) run time. The logon class is passed the target application's logical identifier, which it uses to obtain the application-specific logon information.
- The EJB handler obtains a Java Naming and Directory Interface™ (JNDI) connection, then obtains the service session bean's remote interface and other information required to access the service session bean.

The basic process flow of an adapter worker within a stand-alone adapter daemon is as follows:

1. On startup, the adapter daemon instantiates one or more stand-alone adapter workers, according to the information provided in the kernel's configuration file. The application's logical identifier and optional body category and body type values are passed to the adapter daemon. The body category and body type values are used to obtain additional configuration values.
2. Each stand-alone adapter worker performs the following tasks:
   a. The adapter worker instantiates a native adapter and starts receiving messages. Each message is received under transactional control and returned to the adapter worker as a message-holder object.

   b. For each message received, the adapter worker retrieves the target adapter command type for processing the message from the configuration file, and obtains the appropriate handler for that command type.
   c. The handler obtains from the configuration file any additional information it needs to instantiate the target adapter instance. It instantiates the target adapter and passes the message to it.
   d. If the message is processed successfully (that is, with no exceptions, errors, or bad return data), the message is committed from the incoming message queue. If the message is not processed successfully, it is put onto an error queue. If the message is not processed successfully and cannot be put onto an error queue, then the message is rolled back and all workers are shut down.

The basic process flow of an adapter worker within WebSphere Application Server is as follows:

1. A JMS Listener process operating with WebSphere Application Server Advanced Edition's EJB server receives a JMS message. It then obtains a worker message bean to process the message. The application's logical identifier and optional body category and body type values are part of the worker message bean's environment. The body category and body type values are used to obtain additional configuration values.
2. Each worker message bean performs the following tasks:
   a. The worker message bean instantiates a native adapter and uses the `receiveMsg` method on the native adapter, passing it the JMS message. The native adapter converts the JMS message into a message object and returns the message-holder object.
   b. For each message-holder object received, the adapter worker retrieves the target adapter command type for processing the message-holder object from the configuration file, and obtains the appropriate handler for that command type.
   c. The handler obtains from the configuration file any additional information it needs to instantiate the target adapter instance. It instantiates the target adapter and passes the message-holder object to it.
   d. If the message-holder object is processed successfully (that is, with no exceptions, errors, or bad return data), the message is committed from the incoming message queue. If the message-holder object is not processed successfully, it is put onto an error queue. If the message is not processed successfully and cannot be put onto an error queue, then the message is rolled back and all workers are shut down.

The run-time flow on the target side with an adapter daemon and a stand-alone adapter worker is as follows:

1. 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 that serves as an application identifier. 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`.

   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 communications mode and the receive queue for incoming messages.

See “Starting the kernel” on page 71 for instructions for starting the adapter daemon.

2. When it starts, the adapter daemon looks in the configuration file to determine whether 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.

3. When it starts, the adapter daemon instantiates the first worker and passes it the adapter daemon’s name and the message’s body category and body type.

4. The first worker looks in the configuration file to determine whether trace is enabled for that adapter daemon name. 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.

5. 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.

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

7. 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 (OS) 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 6 on page 33.

   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:

8. 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.

9. The native adapter gets the communications message from the receive queue under commit control and converts it into a message-holder object. It removes all headers specific to the communications transport except for the native kernel header.

10. The native adapter passes the message-holder 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 multistage lookup in the configuration file for the target command type to invoke, in the following order:

    a. For specific body category and body type values.
    b. For a specific body category value and a default body type value.
    c. For a default body category value and a specific body type value.
    d. For default body category and body type values.

    Based on the target command type, the worker determines the appropriate target adapter type handler, a Java class that processes that particular adapter type. It instantiates that particular target adapter.
11. There are two kinds of adapter type handlers: EAB command target adapter handlers and EJB service session bean target adapter handlers. The different kinds of adapter handlers work as follows:

    Note: The EJB service session bean target adapter handler is supported with WebSphere Business Integrator running with WebSphere Application Server on the Windows NT platform, and on AIX.

    - If an EAB command target adapter handler is invoked, it initiates the Common Connector Framework (CCF) environment, sets a logon class with a name obtained from the configuration file, and invokes the EAB target adapter with the name obtained from the configuration file.
    - An EJB service session bean target adapter must interact with WebSphere Business Integrator and WebSphere Application Server to obtain the appropriate configuration information and invoke the EJB service session bean. See "Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server" on page 21 for a discussion of using the kernel with JMS, WebSphere Business Integrator's JMS Listener component, and WebSphere Application Server on the target side of the kernel.
12. Each adapter type has a different interface and required supporting classes, as follows:
    - An EAB target adapter command has three methods to call; these methods 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 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, the target application 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.

- An EJB service session bean calls one method, which requires a `TerminalDataContainer` object. The data returned by the method is considered to be the reply data and must be a `TerminalDataContainer` type object.

13. 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.
14. 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.
15. 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.
16. 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, where *nnnnnnnn* is the number of the log file. In addition, if the WebSphere Business Integrator classes are installed, these classes send an exception to the WebSphere Business Integrator Solution Management component. See “Exception messages” on page 75.
17. 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 multistage lookup in the configuration file in the following order:

    a. For specific body category and body type values.
    b. For a specific body category value and a default body type value.
    c. For a default body category value and a specific body type value.
    d. For default body category and body type values.

    - 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.
18. 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.
19. 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.
20. 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 77 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 communications modes; see "Appendix A. Communications modes" on page 81 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.

Using MQSeries Adapter Kernel with WebSphere Business Integrator and WebSphere Application Server

This section discusses the use of MQSeries Adapter Kernel with the WebSphere Business Integrator and WebSphere Application Server products. For additional details, see the WebSphere Business Integrator documentation.

JMS Listener

WebSphere Business Integrator provides a component called the JMS Listener that works with MQSeries Adapter Kernel and WebSphere Application Server Advanced Edition to provide an alternative way of delivering messages to target applications. The JMS Listener runs inside WebSphere Application Server's Enterprise JavaBeans (EJB) server. This section provides an overview of the JMS Listener's functionality. For additional information, including details on configuring WebSphere Business Integrator and the JMS Listener, see the WebSphere Business Integrator documentation. See "Configuring the kernel" on page 44 for information on configuring MQSeries Adapter Kernel to recognize the JMS Listener as a target. Using the JMS Listener as a target is equivalent to sending a message to an adapter daemon.

Before the JMS Listener can be used, you must deploy an MQSeries Adapter Kernel message bean adapter worker and either Java service session bean adapters or EAB adapters for the target side of the kernel. Perform these tasks by using MQSeries Adapter Builder. In a WebSphere Business Integrator environment, the operation of the kernel within WebSphere Application Server is similar to its operation with a stand-alone adapter daemon, except that the JMS Listener receives the message on behalf of the adapter worker and invokes the appropriate adapter worker. In a stand-alone MQSeries Adapter Kernel environment, the adapter daemon starts adapter workers, which in turn receive messages directly.

The sequence of events when MQSeries Adapter Kernel works with the JMS Listener is as follows:

1. The JMS Listener, monitoring a JMS queue, receives a JMS message object, either from an EJB client or a non-EJB application.
2. The JMS Listener instantiates a *worker message bean* and passes the message object to it. The worker message bean is an instance of a *session bean*, a type of enterprise bean that encapsulates temporary data associated with a specific client.
3. The worker message bean converts the JMS message object into an MQSeries Adapter Kernel message-holder object.
4. Based on the message's header values, the kernel invokes either an EAB adapter or an EJB adapter. If the adapter type to be invoked is an EAB adapter, the data flow is as in the stand-alone case. If the adapter type to be invoked is an EJB adapter, an EJB handler is called and performs the following tasks:

- It determines the correct service session bean (home interface) to invoke, the appropriate method to call, and the method's input parameter type for the `TerminalDataContainer` object.
- It converts the application data contained within the message-holder object to the appropriate `TerminalDataContainer` data structure for the service session bean by using a `Mapper` class. The `TerminalDataContainer` object contains the message-holder object's metadata plus the application objects. In many cases, the application object is the message-holder object's body data XML document string.
- It invokes the service session bean, passing the `TerminalDataContainer` object to the appropriate method on the service session bean. The service session bean, which is part of the Java service adapter, is the target of the message.

5. If a reply was requested, the worker message bean converts the reply `TerminalDataContainer` object to a message-holder object and sends the reply by using the native adapter.
6. If an error occurs, the worker message bean puts the message-holder object onto an error queue by using the native adapter.

National language support

MQSeries Adapter Kernel provides national language support for both, Java and C adapters.

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:

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:

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

Hardware

MQSeries Adapter Kernel runs on the following hardware:

- An IBM PC (or compatible) machine running Windows NT 4.0, Service Pack 5 or later, 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 8.
- An IBM AS/400® or iSeries machine running OS/400 version 4.4, 4.5, or 5.1.

  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 compiled API 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 85. 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 potentially work correctly with the kernel but are not officially supported.

For Windows systems:

- Microsoft Windows NT version 4.0, Service Pack 5 or later; 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 5.2 with SupportPac MA88.
- IBM Java Development Kit (JDK) version 1.2.2 or 1.3.

  Note: MQSeries Adapter Kernel supports JDK version 1.3 on all platforms. For the OS/400 operating system, version 5.1 is required.

For AIX:

- AIX operating system version 4.3.2 or 4.3.3.
- VisualAge® C++ for AIX version 5.0.
- MQSeries version 5.2 with SupportPac MA88.
- Java Development Kit version 1.2.2. JDK 1.3 is not supported.
- X Window System (X11R5 or higher). This is required for installation but not at run time.

For HP-UX:

- HP-UX operating system version 11.0.
- HP-UX C/ANSI C Compiler. See the `readme.txt` file for details.
- MQSeries version 5.2 with SupportPac MA88.
- Java Development Kit version 1.2.2. JDK 1.3 is not supported.
- X Window System (X11R5 or higher). This is required for installation but not at run time.

For Solaris:

- Solaris operating environment version 8.
- Sun Workshop Compilers C/C++. See the `readme.txt` file for details.
- MQSeries version 5.2 with SupportPac MA88.
- Java Development Kit version 1.2.2. JDK 1.3 is not supported.
- X Window System (X11R5 or higher). This is required for installation but not at run time.

For OS/400:

- OS/400 operating system version 4.4, 4.5, or 5.1 including the following programs:
  - Java Toolkit and Java Developer Kit version 1.2.2. JDK 1.3 is supported for OS/400 operating system version 5.1. 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 licensed program number 5769–CX2.
- MQSeries version 5.2 with SupportPac MA88.

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 5.2 with SupportPac MA88

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

See "Appendix B. Validated configurations" on page 85 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 3 on page 31 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 26.

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. For OS/400 operating system version 5.1 you can also use JDK 1.3. 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 from the AS/400 library at 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 potentially 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 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 1.2.2. For OS/400 operating system version 5.1 you can also install JDK 1.3. 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 1.2.2 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, `/usr/lpp/mqak` on AIX, `/MQAK` on HP-UX, `/opt/MQAK` on Solaris, 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 `aqmstpad.bat` (Windows systems) or aqmstpad.sh (UNIX and OS/400) file, a script that is run to stop 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.
- The `aqmchk.bat` file, (Windows systems) or `aqmchk.sh` (UNIX and OS/400) file, a script that is run to start the configuration checker utility.

`documentation`
Contains the product documentation, including the Information Center.

`runtimefiles`
Contains kernel run-time files.

`samples`

Contains the following:

- Samples of adapters and associated configuration and utility files. You can experiment with and learn from them.
- The mqak_model_q.mqsc. You can use this sample file for creating a model queue that is necessary for using the stop command-line utility.

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 48 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 49 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 76.
- The `aqmconfig.xml` file. See "The configuration file" on page 49 for a discussion of this file.
- The `aqmsetup` file. See "The setup file" on page 48 for a discussion of this file.
- The `aqminstalltest.xml` file.

Chapter 3. Installing the kernel

This chapter discusses the steps necessary to install and verify MQSeries Adapter Kernel. Installation consists of the following general steps:

Step 1. Prepare for installation. See "Preparing for installation" for details.

Step 2. Install the kernel. See "Installing the kernel" on page 30 for details.

Step 3. Complete several post-installation steps. See "Completing the post-installation" on page 32 for details.

Step 4. Verify the installation. See "Verifying the installation" on page 34 for details.

This chapter also discusses the following topics:

- Using silent installation to install MQSeries Adapter Kernel. See "Using silent installation" on page 38 for details.
- Upgrading MQSeries Adapter Kernel from an earlier version. See "Upgrading the kernel" on page 39 for details.
- Removing an installation of MQSeries Adapter Kernel. See "Removing the kernel" on page 40 for details.

After installing the kernel, perform the following additional tasks to prepare it for use:

1. Configure the kernel. See "Configuring the kernel" on page 44 for details.
2. Configure messaging software and optional software. See "Configuring MQSeries and MQSeries Integrator" on page 69 for details.
3. Build your adapters by using MQSeries Adapter Builder, then test and deploy them.
4. Start the kernel. See "Starting the kernel" on page 71 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 executable path. Ensure that all user IDs that run the kernel have read, write, and execute permissions.

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 for the source adapter to run.

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.

Perform the following steps before beginning installation:

- Read the `readme.txt` file on the CD-ROM or local area network. It potentially contains important information that became available after this book was completed. It is located in the root installation directory.
- Visit the MQSeries Web site at www.ibm.com/software/ts/mqseries/. It potentially contains important information that became available after this book was published, potentially including a new edition of this book.
- If you are upgrading from a previous version of MQSeries Adapter Kernel, see "Upgrading the kernel" on page 39 for instructions.
- 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.

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 operating system-specific steps:

On Windows systems:

Step 1. 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.

Step 2. 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:

Step 1. 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.

Step 2. 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, you must specify the installation directory as a fully qualified path name, not as a relative path name. The default installation directories on UNIX are as follows:

- AIX: `/usr/lpp/mqak`
- HP-UX: `/MQAK`
- Solaris: `/opt/MQAK`

On OS/400:

Step 1. Ensure that all prerequisites listed in “Hardware” on page 23, “OS/400 software prerequisites” on page 24, 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.

Step 2. Create a user profile named MQAKSRV on the AS/400 system by using the **CRTUSRPRF** command at a Control Language (CL) prompt.

Step 3. 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:
  a. Ensure that remote AWT is set up and running. See “Using remote AWT” on page 25 for details.
  b. 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.
  c. To improve the performance of the installation process, run the Create Java Program (**CRTJVAPGM**) command against the `installAS400.jar` file.
  d. 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:
  a. Ensure that the requirements specified in “Using an attached client” on page 26 are met.

b. 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 CL prompt.

c. 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.

d. 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).

e. Enter the following command:

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

f. 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. Do not check the **Default User** checkbox. Click **Next**.

Step 4. 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).

Step 5. 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`.

Step 6. Add the library MQAK to the QSYS.LIB library list.

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 34. After you have verified the installation, follow the steps in “Completing the post-installation” to set environment variables and move several configuration files to support production at your site.

Install the kernel on other computers as required.

Completing the post-installation

After installing the kernel, perform the following steps:

Step 1. Decide where to put the `aqmsetup` and `aqmconfig.xml` files, which are used to configure the kernel. For more information on these files, see “Configuring the kernel” on page 44.

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 2. 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. Locating them outside the directory where you installed MQSeries Adapter Kernel 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 3. 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 Step 5.

Step 4. 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 5. 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/`*`user_name`*`/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/`*`user_name`*`).`

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

Step 6. 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 the 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 the Bourne shell or the Korn shell) or `.cshrc` file (if you use the C shell).

See Step 7 on page 17 for additional information about scheduling policies.

Step 7. 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 the 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 the Bourne shell or the Korn shell) or `.cshrc` file (if you use the C shell).

After completing the post-installation steps, perform the following tasks to prepare the kernel for use:

1. Prepare for production. See "Preparing for production" on page 43.
2. Edit the configuration file. See "Configuring the kernel" on page 44 for details.
3. Configure MQSeries and optional software. See "Configuring MQSeries and MQSeries Integrator" on page 69.
4. For production systems, take into account "Performance recommendations" on page 69.
5. Start the kernel. See "Starting the kernel" on page 71.
6. Set up a kernel maintenance plan. See "Maintaining the kernel" on page 74.

Verifying the installation

After you install the kernel, verify that it was installed correctly by running a verification script. The script sends a test message from a source application by using a source adapter, then to MQSeries by using the kernel. 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 44 for information on these files.

You can validate the configuration file before putting it into production. See "Validating the configuration file" on page 66.

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 name `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. Ensure that you are authorized to install and verify the kernel. See "Preparing for installation" on page 29.

Step 3. Start the verification as follows:
- On Windows systems, double-click the `aqmverifyinstall.bat` file in the `verification` directory. Alternatively, open a command prompt, 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 entering the **STRQSH** command.
  b. Copy the `/QIBM/ProdData/mqak/verification/aqmsetup` file to your home directory (`/home/`*`user_name`*).
  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 36 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 37 for details.

Step 8. Return to the installation procedure and configure the kernel to support operation in your particular site. Go to Step 1 on page 32.

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 that the `aqmconfig.xml` file is located 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 76.

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`. Ensure that the CLASSPATH environment variable includes 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.

Using silent installation

MQSeries Adapter Kernel can be installed on all platforms by using *silent installation*. Silent installation enables you to bypass the MQSeries Adapter Kernel installation program, where you must manually select the installation options you want. Silent installation is useful when you want to install the default configuration on multiple machines.

To install the kernel silently, perform the following operating system-specific steps:

On Windows systems:

Step 1. Open a command prompt and change to the directory that contains the MQSeries Adapter Kernel installation files.

Step 2. Enter the following command:

```
java -cp install.jar run -P product.installLocation="install_location"
-silent
```

where *install_location* is the desired installation location (for instance, `D:\mqak`).

On UNIX:

Step 1. At a terminal, change to the directory that contains the MQSeries Adapter Kernel installation files. 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.

Step 2. Enter the following command:

```
java -cp install.jar run -P product.installLocation="install_location"
-silent
```

where *install_location* is the desired installation location (for instance, `/opt/mqak`).

On OS/400:

If you are not using an attached client to access the AS/400 machine, perform the following steps:

Step 1. 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.

Step 2. To improve the performance of the installation process, run the Create Java Program (**CRTJVAPGM**) command against the `installAS400.jar` file.

Step 3. Depending on whether you are using a CL prompt or a **qsh** session, enter one of the following commands:

- If you are using a CL prompt, enter the following command:

```
RUNJVA CLASS(run)
CLASSPATH('/installAS400.jar')
PROP((java.version 1.2)) PARM('-silent')
```

- If you are using a **qsh** session, enter the following command:

  ```
  java -Djava.version=1.2 -classpath installAS400.jar run -silent
  ```

If you are using an attached client to interface with the AS/400 system, perform the following steps:

Step 1. Ensure that the requirements specified in “Using an attached client” on page 26 are met.

Step 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.

Step 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.

Step 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).

Step 5. Enter the following command:

```
java -cp installAS400.jar run -silent -os400 machine_name user_ID
password
```

where *machine_name* is the TCP/IP address of the AS/400 system, *user_ID* is your user ID, and *password* is your password.

Upgrading the kernel

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

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

Step 2. If an MQSeries Adapter Kernel 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.
  - Run 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 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.
  - Run 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 30 for details.

Step 5. Restore the `aqmsetup` and `aqmconfig` files to their previous locations in the MQSeries Adapter Kernel installation directory. If necessary, convert the `aqmconfig.properties` file to an `aqmconfig.xml` file. For more information on the `aqmconfig.xml` file, see "The configuration file" on page 49.

Removing the kernel

There are several ways to remove the kernel. Note that 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.

- 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.
  - Run the `aqmuninstall.bat` file in the kernel's root directory.
  - To uninstall the kernel silently (that is, without having the uninstallation program prompt you for details or confirmation), open a command prompt, change to the kernel's installation directory, and enter the following command:

    ```
    java -cp uninstall.jar run -silent
    ```
- On UNIX, change to the kernel's root directory and enter the following command:

  ```
  aqmuninstall.sh
  ```

  To uninstall the kernel silently (that is, without having the uninstallation program prompt you for details or confirmation), change to the kernel's root directory and enter the following command:

  ```
  java -cp uninstall.jar run -silent
  ```
- On OS/400, use one of the following methods to uninstall the kernel.
  - If you are using remote AWT to uninstall the kernel, perform the following steps:

    Step 1. Ensure that remote AWT is set up and running. See "Using remote AWT" on page 25 for details.

    Step 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.

Step 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 uninstall the kernel, perform the following steps:

Step 1. Ensure that the requirements specified in "Using an attached client" on page 26 are met.

Step 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.

Step 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.

Step 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.

Step 5. Enter the following command:

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

To uninstall the kernel silently (that is, without having the uninstallation program prompt you for details or confirmation), enter the following command:

```
java -cp uninstall.jar run -silent -os400 machine_name user_ID
password
```

where *machine_name* is the TCP/IP address of the AS/400 system, *user_ID* is your user ID, and *password* is your password.

– If you are working directly on the AS/400 system at a CL prompt or **qsh** session and want to uninstall the kernel silently (that is, without having the uninstallation program prompt you for details or confirmation), enter one of the following commands:

- At a CL prompt:

```
RUNJVA CLASS(run)
CLASSPATH('/uninstall.jar')
PROP((java.version 1.2)) PARM('-silent')
```

- At a **qsh** session:

```
java -Djava.version=1.2 -classpath uninstall.jar run -silent
```

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 required 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 44 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 69.
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. 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 71.

Configuring the kernel

This section discusses configuring the kernel for use in your environment. "Overview of configuration" provides a conceptual overview of kernel configuration. "Files involved in startup and configuration" on page 48 discusses the various files that together define an MQSeries Adapter Kernel configuration. "The setup file" on page 48 discusses the `aqmsetup` file, which defines several of the kernel's initial settings. "The configuration file" on page 49 discusses the `aqmconfig.xml` file, which provides the kernel with specific configuration information such as the names of source and target applications, source and target adapters, queues and queue managers, communication modes, and logging and tracing specifications.

Overview of configuration

This section provides a conceptual overview of kernel configuration. It is important to understand the kernel's run-time flow before configuring the kernel. This section discusses the run-time flow at a simplified level. See "Run-time flow" on page 10 for detailed information on the run-time flow.

At the most basic level, the configuration of MQSeries Adapter Kernel is driven by the data that flows between applications. Configuration must also take the following factors into account:

- The applications that receive the data.
- The adapters that are required on the source side, and the target adapters, adapter daemons, and workers that are required on the target side.
- The communications modes, marshaling formats, and transport mechanisms that are used.

Data structures and data format are different for each application in the configuration. For example, if a configuration includes two applications, A and B, that each send purchase-order data to application C, the data from application A is likely to be in a different format and have different tag meanings from application B's data. To prevent application C from needing to recognize and parse the two different data flows from the two different applications, each application's data is converted into an *integration message* that is in an *integration-neutral format*. Normally the integration-neutral format is an industry standard based on XML. The only data format that application C needs to be able to recognize and parse is the integration-neutral format.

Figure 3 on page 45 shows the flow of data from applications A and B to applications C and D. Following the figure is an explanation of the various data flows it depicts.

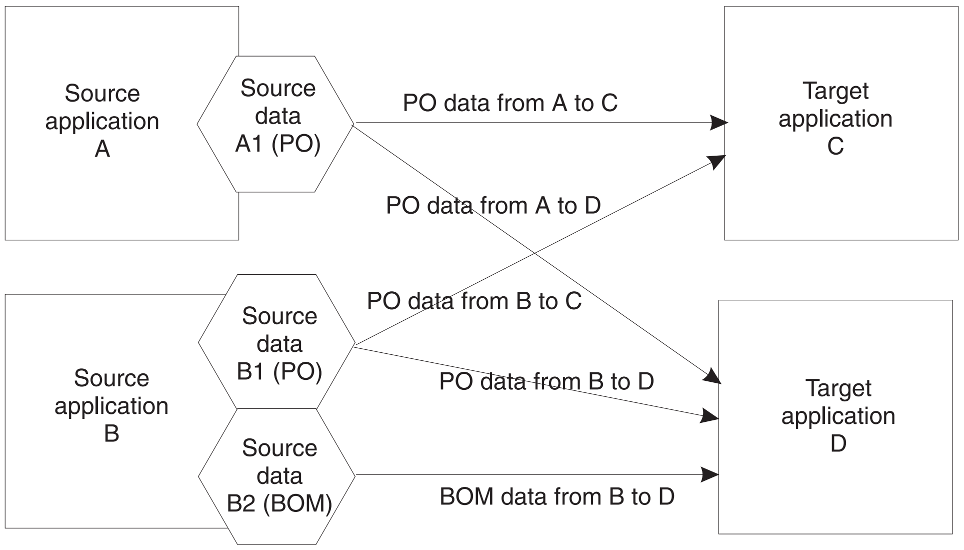


Figure 3. Applications connected by data flows in a simple configuration

In Figure 3, purchase-order data from application A flows to applications C and D, purchase-order data from application B also flows to applications C and D, and bill-of-material data from application B flows only to application D. In each case, data is converted to an industry-standard format before being sent to the target applications. In the case of purchase-order data coming from applications A and B, the data is converted to a standard XML format representing purchase-order data. In the case of bill-of-material data coming from application B, the data is converted to a standard XML format representing bill-of-material data.

A communications transport such as MQSeries or an implementation of the Java Message Service (JMS) is used to send the data to the target application or applications. The integration message is converted into the *marshaling format* required by the specific communications transport, then delivered to the communications transport (for example, an MQSeries queue). Each target application can use a different communications transport and marshaling format to receive messages. For instance, application C can use MQSeries to receive messages and application D can use JMS, as shown in Figure 4 on page 46. In this case, all integration messages going to application C (that is, the purchase-order data from applications A and B) are converted to an MQSeries marshaling format, and all integration messages going to application D (that is, purchase-order data from applications A and B as well as bill-of-material data from application B) are converted to a JMS marshaling format. MQSeries Adapter Kernel uses the following mechanisms to perform these conversions:

- A *source adapter* is used to convert application data to an integration message. Source adapters are created in MQSeries Adapter Builder.
- The *native adapter* is used to convert the integration message into a *communications message*. The native adapter uses the *logical message service* (LMS) to convert the message for transportation by the communications transport; the LMS is specific to the communications transport being used. The LMS then uses a *formatter* to marshal the message onto the transport.

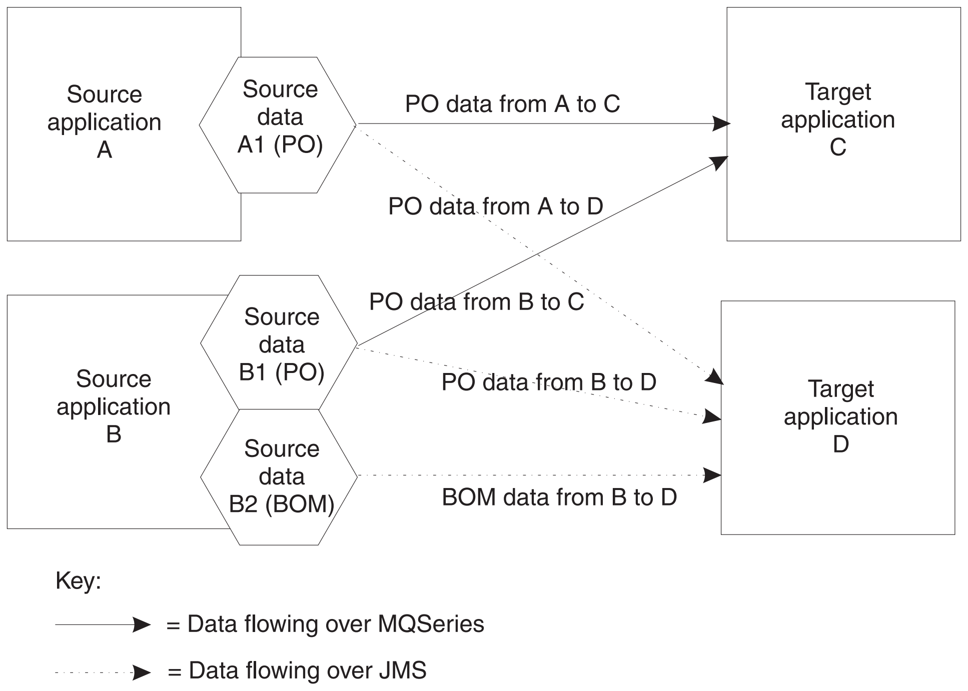


Figure 4. Applications connected by different communications transports in a simple configuration

The flow of data from application to integration message to communications message is shown in Figure 5 and Figure 6 on page 47. When a communications message is received at the target, the conversions are reversed: the native adapter converts the communications message back into the integration message; then, if necessary, the target adapter converts the integration message into the data format required by the target application.

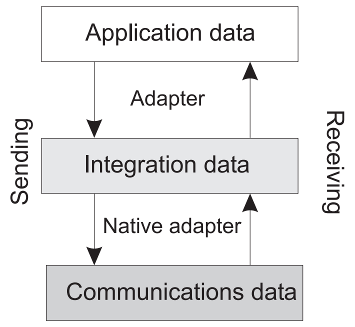


Figure 5. Conversion of data

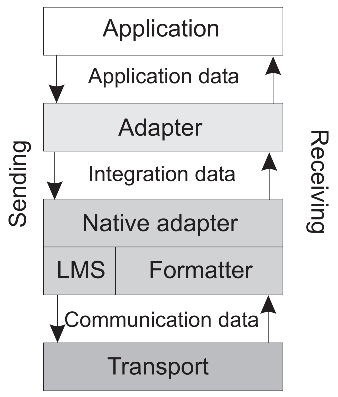


Figure 6. Flow of data

Each point that the data passes through must be represented in the kernel’s configuration file. There are three logical configuration requirements that are divided by application identifier (source or target). An application identifier can be for either a source application or a target application, depending on whether messages are going into the application (target) or coming from the application (source). The configuration file must include the following types of information:

- Communications
  - Where the data needs to go
  - The communications transport to use
  - The communications marshaling method (formatter) to use
  - The underlying communications requirements, such as an MQSeries queue manager and MQSeries queue names, or a JMS queue connection factory and JMS queue names
- Adapters (only for the target side)
  - The adapters that are required to process the data
  - The type of adapters used (EAB or EJB)
  - Additional information for the specific adapter type used
  - For stand-alone MQSeries Adapter Kernel, the adapter daemon and worker information
- Other
  - Tracing specifications
  - Logging specifications

The flow of data as it relates to the different parts of the configuration is shown in Figure 7 on page 48.

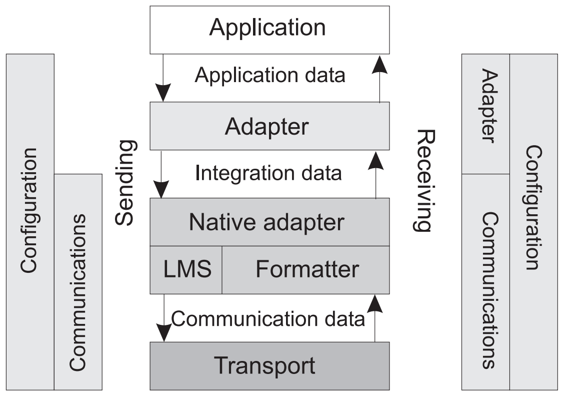


Figure 7. Flow of data related to configuration

See "Syntax and organization of the configuration file" on page 50 for details on mapping these configuration requirements to XML elements in the `aqmconfig.xml` file, which controls these aspects of the kernel's configuration. "Common configurations" on page 59 lists several common configurations.

Files involved in startup and configuration

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 the Bourne shell or the Korn shell) or `.cshrc` file (if you use the C shell).

  For information on setting the appropriate environment variables on OS/400, see Step 5 on page 32.
- The `aqmsetup` file, which provides several initial setup values for the kernel. See "The setup file" for more information.
- The `aqmconfig.xml` file, which configures the kernel. See "The configuration file" on page 49 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 76.

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 74.

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" on page 49.
- The location of XML DTDs, if not in the current directory.

- Java Native Interface (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 99 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 50 provides information on the structure of the configuration file. "Editing the configuration file" on page 65 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 93 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 worker information on the target side
- Trace clients
- Trace servers
- Marshaling and routing of messages, determined by the following specifications:
  - The names of receive queues, error queues, and reply queues
  - One or more default destinations to which messages are to be sent
  - The name of the MQSeries queue manager or JMS queue connection factory that gets or sends the message
  - The timeout for receiving messages or replies
  - The target adapter class on the target side of the kernel that processes each message
  - Additional information specific to the target adapter
  - The minimum number of workers on the target side (if running stand-alone MQSeries Adapter Kernel)
  - Enabling and disabling trace, and control of trace level
  - Enabling and disabling audit logging

- Communications mode

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, communications modes for receiving messages, and message formatters.
- `ePICAdapterDaemonExtensions`—An optional child of the `ePICApplication` element representing an adapter daemon. 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 8 on page 51 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 93 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. -->
<epicrfh2messageset>true</epicrfh2messageset>
          </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 8. 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 Java Naming and Directory Interface (JNDI) file system context (FSContext) when Java Message Service (JMS) objects are used. The default is the current directory. Required if JMS is used. See "Using JMS object storage" on page 83 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 perform audit logging. Audit logging requires the WebSphere 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 list 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 a logon class used to connect to the application when using an EAB adapter. 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, specifies 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 host name of the machine on which the queue manager resides. Required only 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 server process of the queue manager. The default is `1414` (the MQSeries default). Required 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 only 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`
- `epiccommandtype`
- `epiccommandejbmapper`
- `epiccommandejbmethod`
- `epiccommandejbmethodparmtype`
- `epiccommandejburl`
- `epiccommandejbinitialcontext`
- `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)

`epicrfh2messageset`
Enables the <mcd> folder (message content descriptor) processing for MQSeries Integrator if set to "true". The default value is "false". This setting is optional and only relevant if the `epicreceivemode` is MQRFH2. Note that you cannot use this setting within LDAP. For a description of the <mcd folder, see “MQSeries Integrator version 2 header” on page 91. For further details, refer to the MQSeries Integrator documentation.

`epiccommandclassname`
Specifies the name of an EAB target adapter or EJB command that is invoked to process messages. Required if an adapter daemon or WebSphere Application Server is being used to receive messages.

Child elements: None

Attributes: None

`epiccommandtype`
Specifies the type of the target adapter. Possible values include `MQAKEAB` and `MQAKEJB`. `MQAKEAB` specifies a standard MQSeries Adapter Kernel EAB target adapter; `MQAKEJB` specifies that enterprise beans are used on the target side of the kernel in WebSphere Application Server. The default is `MQAKEAB`. A value of `MQAKEJB` is required when the target adapter is an enterprise bean.

Child elements: None

Attributes: None

`epiccommandejbmapper`
Specifies the name of the `TDCMapper` class used to map input data. The default is `TDCGenericMapper`. Required when the target adapter is an enterprise bean.

Child elements: None

Attributes: None

`epiccommandejbmethod`
Specifies the name of the method to invoke on an enterprise bean. The method must accept a `TerminalDataContainer` object as input and return a `TerminalDataContainer` object. The default is `execute`. Required when the target adapter is an enterprise bean.

Child elements: None

Attributes: None

`epiccommandejbmethodparmtype`
Specifies the class name of the object that is being used as the parameter for the method being invoked on the enterprise bean. The default is the class name of the object returned by `TDCMapper`. Required when the target adapter is an enterprise bean.

Child elements: None

Attributes: None

epiccommandejburl

Specifies the uniform resource locator (URL) of a deployed enterprise bean, in the form `IIOP://`*`host_name`*`:`*`port`*, where *host_name* is the host name of the EJB server and *port* is the port at which the name server listens (by default, `900`). The default is `IIOP:///`. Required when the target adapter is an enterprise bean.

Child elements: None

Attributes: None

epiccommandejbinitialcontext

Specifies the name of the initial context factory used to look up the home name of the enterprise bean. The default is `com.ibm.ejs.ns.jndi.CNInitialContextFactory`. Required when the target adapter is an enterprise bean.

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 communications mode to be used. See "Appendix A. Communications modes" on page 81 for a list and explanation of valid communications 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 10 on page 82 and Table 11 on page 82 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 communications mode. See "Appendix A. Communications modes" on page 81 for a list of MQSeries communications 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 communications mode. See "Appendix A. Communications modes" on page 81 for a list of MQSeries communications 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 communications mode. See "Appendix A. Communications modes" on page 81 for a list of MQSeries communications modes.

Child elements: None

Attributes: None

epicjmsreceivequeuename
Specifies the name of the queue from which to receive messages. Required for the JMS communication mode. 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 the JMS communications mode. 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 the JMS communications mode. 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 the FILE communications mode.

  Child elements: None

  Attributes: None

epiccommitfiledir
: Specifies the name of the directory in which to hold received messages until they are committed. Required for the FILE communications mode 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 the FILE communications mode.

  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 that the adapter daemon is servicing. It 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

Common configurations

This section lists the configuration values for several common configuration scenarios, including values for sending and receiving messages by using various communications transports. When a message is sent, configuration values are obtained from both the source side and the target side; when a message is received, configuration values are obtained only from the target side. The source and target are represented by their respective logical identifiers. These examples assume that the source and target are on two different machines. If the target application identifier is not already set, it is determined from the value of the `epicdestids` element in the source's configuration.

Note: The configuration scenarios list the element configuration values that are applicable and that can be set. Refer to the element's listing in "XML elements used in the configuration file" on page 51 for any defaults that apply to that element.

MQSeries common configurations: This section provides common configurations when MQSeries is used as the communications transport. The `epicreceivemode` element specifies an MQSeries communications mode (for example, MQPP or MQRFH2). The following scenarios are listed:

- Table 2 shows configuration elements that need to be set when sending a message from an MQSeries server to an MQSeries server.
- Table 3 on page 60 shows configuration elements that need to be set when sending a message from an MQSeries server that is using a remote queue manager to an MQSeries server.
- Table 4 on page 60 shows configuration elements that need to be set when sending a message from an MQSeries client that is using a host server to an MQSeries server.
- Table 5 on page 61 shows configuration elements that need to be set when receiving a message on an MQSeries server.
- Table 6 on page 62 shows configuration elements that need to be set when receiving a message on an MQSeries client that is using a host server.

Table 2. Common configuration: Sending a message from an MQSeries server to another MQSeries server

Source configuration	**Target configuration**
	The `epicreceivemode` element specifies an MQSeries communications mode.
The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the source application's machine.	
	The `epicreceivemqppqueue` element specifies the name of the receive queue. This queue must be an MQSeries remote queue on the target application's machine or part of an MQSeries cluster.

Table 2. Common configuration: Sending a message from an MQSeries server to another MQSeries server (continued)

Source configuration	**Target configuration**
The `epicreplymqppqueue` element specifies the name of the reply queue. This queue must be an MQSeries local queue on the sender's machine or part of an MQSeries cluster. Used only for synchronous requests and replies.	
	The `epicmessageformatter` element specifies the name of the formatter to use.
The `epicreceivetimeout` element specifies the time the receiver waits for a reply before it times out.	

Table 3. Common configuration: Sending a message from an MQSeries server to an MQSeries server via a remote queue manager

Source configuration	**Target configuration**
	The `epicreceivemode` element specifies an MQSeries communications mode.
The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the source application's machine.	The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the target application's machine. The name must be specified; a default value cannot be used.
The `epicremotequeuemanagertosend` element specifies that a remote queue manager is being used to send messages.	
	The `epicreceivemqppqueue` element specifies the name of the receive queue. This queue must be an MQSeries local queue on the target application's machine or part of an MQSeries cluster.
The `epicreplymqppqueue` element specifies the name of the reply queue. This queue must be an MQSeries local queue on the sender's machine or part of an MQSeries cluster. Used only for synchronous requests and replies.	
	The `epicmessageformatter` element specifies the name of the formatter to use.
The `epicreceivetimeout` element specifies the time the receiver waits for a reply before it times out.	

Table 4. Common configuration: Sending a message from an MQSeries client that is using a host server to an MQSeries server

Source configuration	**Target configuration**
	The `epicreceivemode` element specifies an MQSeries communications mode.

Table 4. Common configuration: Sending a message from an MQSeries client that is using a host server to an MQSeries server (continued)

Source configuration	Target configuration
The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the sender client's host machine.	
The `epicmqppqueuemgrhostname` element specifies the host name of the MQSeries server machine.	
The `epicmqppqueuemgrportnumber` element specifies the port number of the server process of the queue manager on the server machine.	
The `epicmqppqueuemgrchannelnumber` element specifies the channel number of the queue manager server.	
	The `epicreceivemqppqueue` element specifies the name of the receive queue. This queue must be an MQSeries remote queue on the target application's machine or part of an MQSeries cluster.
The `epicreplymqppqueue` element specifies the name of the reply queue. This queue must be an MQSeries local queue on the sender client's host machine or part of an MQSeries cluster. Used only for synchronous requests and replies.	
	The `epicmessageformatter` element specifies the name of the formatter to use.
The `epicreceivetimeout` element specifies the time the receiver waits for a reply before it times out.	

Table 5. Common configuration: MQSeries server receiving a message

Source configuration	Target configuration
Not applicable.	The `epicreceivemode` element specifies an MQSeries communications mode.
	The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the target application's machine.
	The `epicreceivemqppqueue` element specifies the name of the receive queue. This queue must be an MQSeries local queue on the target's machine.
	The `epicerrormqppqueue` element specifies the name of the error queue. This queue must be an MQSeries local queue on the target's machine or part of a cluster. Required only if using an adapter worker.
	The `epicmessageformatter` element specifies the name of the formatter to use.

Table 5. Common configuration: MQSeries server receiving a message (continued)

Source configuration	Target configuration
	The `epicreceivetimeout` element specifies the time the receiver waits for a message to receive before it times out.

Table 6. Common configuration: MQSeries client that is using a host server receiving a message

Source configuration	Target configuration
Not applicable.	The `epicreceivemode` element specifies an MQSeries communications mode.
	The `epicmqppqueuemgr` element specifies the name of the queue manager. This queue manager must exist on the receiver's client host machine.
	The `epicmqppqueuemgrhostname` element specifies the host name of the MQSeries server machine.
	The `epicmqppqueuemgrportnumber` element specifies the port number of the server process of the queue process on the server machine.
	The `epicmqppqueuemgrchannelnumber` element specifies the channel number of the queue manager server.
	The `epicreceivemqppqueue` element specifies the name of the receive queue. This queue must be an MQSeries local queue on the receiver's client host machine.
	The `epicerrormqppqueue` element specifies the name of the error queue. This queue must be an MQSeries local queue on the receiver's client host machine or part of a cluster. Required only if using an adapter worker.
	The `epicmessageformatter` element specifies the name of the formatter to use.
	The `epicreceivetimeout` element specifies the time the receiver waits for a message to receive before it times out.

JMS common configurations: This section provides common configurations when JMS is used as the communications transport. The `epicreceivemode` element specifies JMS.

If the MQSeries JMS implementation is being used, the appropriate MQSeries objects must exist. For example, a JMS queue connection factory must be related to a queue manager on an MQSeries server, and a JMS queue must be related to an MQSeries queue. MQSeries objects do not need to be listed in the configuration, but the supporting MQSeries objects must exist.

The following scenarios are listed:

- Table 7 shows configuration elements that need to be set when sending a message via JMS.
- Table 8 shows configuration elements that need to be set when receiving a message via JMS.

Table 7. Common configuration: Sending a message via JMS

Source configuration	Target configuration
	The `epicreceivemode` element specifies JMS communications mode.
The `epicjmsconnectionfactoryname` element specifies the name of the JMS queue connection factory. The referenced object must exist in the configuration.	
	The `epicjmsreceivequeuename` element specifies the name of the JMS receive queue. The referenced object must exist in the configuration.
The `epicjmsreplyqueuename` element specifies the name of the JMS reply queue. The referenced object must exist in the configuration. Used only for synchronous requests and replies.	
	The `epicmessageformatter` element specifies the name of the formatter to use.
The `epicreceivetimeout` element specifies the time the receiver waits for a reply before it times out.	

Table 8. Common configuration: Receiving a message via JMS

Source configuration	Target configuration
Not applicable.	The `epicreceivemode` element specifies JMS communications mode.
	The `epicjmsconnectionfactoryname` element specifies the name of the JMS queue connection factory. The referenced object must exist in the configuration.
	The `epicjmsreceivequeuename` specifies the name of the JMS receive queue. The referenced object must exist in the configuration.
	The `epicjmserrorqueuename` element specifies the name of the JMS error queue. The referenced object must exist in the configuration.
	The `epicmessageformatter` element specifies the name of the formatter to use.
	The `epicreceivetimeout` element specifies the time the receiver waits for a message to receive before it times out.

Adapter common configurations: This section provides common configurations when an adapter is invoked on the target side. Different configuration values are used depending on whether the target is an Enterprise Access Builder (EAB) target adapter (specified by an `epiccommandtype` value of `MQAKEAB`) or an EJB service session bean target adapter (specified by an `epiccommandtype` value of `MQAKEJB`).

Note: EJB service session bean target adapters are supported only in WebSphere Application Server.

For an `epiccommandtype` value of `MQAKEAB`, specify values for the following additional elements:

- `epiclogoninfoclassname`
- `epiccommandclassname`

For an `epiccommandtype` value of `MQAKEJB`, specify values for the following additional elements:

- `epiccommandclassname`
- `epiccommandejbmethod`
- `epiccommandejbmethodparmtype`
- `epiccommandejburl`
- `epiccommandejbinitialcontext`
- `epiccommandejbmapper`

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 shown in “Appendix D. Sample of the configuration file” on page 93 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. It is recommended that an appropriate value be set 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 www.alphaworks.ibm.com. The following recommendations apply to editing the configuration file:

- Before you begin editing the configuration file, gather all the 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 communications 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. 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 break 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 value 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 kernel 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 whether 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 temporarily to the setup file in the validation directory:

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

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

  where `E:\` represents the correct drive and *run_time_files* 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/run_time_files/aqmsetup
  ```

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

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 value of the source logical identifier entered for option 20 in Step 5 on page 66.

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 the Korn shell; if you use another UNIX shell (such as the Bourne shell or the C shell), use the backslash character before the question mark (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.

Using the configuration checker utility

The MQSeries Adapter Kernel provides a configuration checker utility that checks the active configuration of the kernel, as well as LDAP configurations for:

- Syntactical correctness
- Completeness
- Availability of configured MQ resources
- Availability of configured JMS objects

The configuration checker utility performs the following steps:

1. It checks the setup file as specified via the AQMSETUPFILE environment variable.
2. It scans the configuration that is specified in the setup file. The configuration contains a set of epicApplication specifications. The utility checks those epicApplications first that are used for adapter communications. These applications have an epicBodyCategory and an epicBodyType specified. In the output file of the configuration checker these specifications are referred to as ″fully qualified″ applications. The EpicApplications that do not have specified an epicBodyCategory or an epicBodyType are checked only if they are referenced by a fully qualified application.
3. It checks all settings of each fully qualified applications, for example, trace settings, log settings, dependent ApplIDs, or communication settings.
4. It checks each application that is referenced by a fully qualified application for existence and consistency, for example the TraceClient.
5. It checks each application that is referenced indirectly by a fully qualified application for existence and constistency..
6. It checks each resource that is needed for an accessible application for existence and accessibility as, for example, MQ queues or JMS queue objects.

To invoke the configuration checker utility open a command prompt and enter the following command: `aqmchk [-q]`.

If you specify `-q`, the utility runs in quiet mode, that is, only error messages are displayed or the message informing you that the checking process finished successfully. If you do not specify `-q`, the utility reports all steps that are performed on the screen, including the results of the checking process. Independent from the `-q` option, the utility writes all results of the checking process to the aqmchk.log file, which is located in the directory that you are currently using.

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, clear them to verify installation. See “Verification procedure” on page 35 for the queues used for verifying installation.
- Set up queues to support the 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 message flows (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.

Using the start and stop command-line utilities

When an adapter daemon process is started, the required number of adapter daemon worker threads is started according to the specification in the kernel's configuration file. If you use the command-line utility for starting the adapter daemon **aqmstrad**, you can use a temporary administration queue. The adapter daemon creates the temporary queue according to the administration model queue, and then listens to this queue and waits for a stop request. You must create an administration model queue before the adapter deamon can create a temporary administration queue.

In addition to the start utility, MQSeries Adapter Kernel provides a utility for stopping an adapter daemon, called **aqmstpad**. The following sections describe how to start and stop an adapter daemon by using the command-line utilities, and how to create an administration model queue.

Creating an administration model queue

Before the adapter deamon can create a temporary administration queue, you must create an administration model queue. You can use the script input file mqak_model_q.mqsc to create an administration model queue. It is defined as follows:

```
DEFINE QMODEL (MQAK.ADMIN.QUEUE) +
DESCR('The MQAK administration model queue (to stop AdapterDaemons).') +
PUT(ENABLED) +
GET(ENABLED) +
DEFTYPE(TEMPDYN) +
DEFSOPT(SHARED) +
DEFPSIST(NO) +
SHARE +
USAGE(NORMAL);
```

This script input file is located in the samples directory of the MQSeries Adapter kernel installation.

You must have administration rights to create an administration model queue. To execute the script, open a command prompt and change to the samples directory of the kernel. Then, enter the following command:

```
runmqsc [QueueManager Name] < mqak_model_q.mqsc
```

where [QueueManager Name] is the name of the queue manager you have configured in the configuration file of the kernel. If you have configured more than one queue manager, you must specify an administration queue for each queue manager that is configured.

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 any associated 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] [-timing] [-q qid]
```

- On UNIX and OS/400:

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

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 communications mode and the related information for receiving messages. If no value is provided, the adapter daemon uses the value DEFAULT.

-bt *body_type*
: Specifies the body type that the adapter daemon worker uses for determining the communications mode and related information for receiving messages. If no value is provided, the adapter daemon uses 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.

-timing
Allows a timing test to run. The default is not to allow timing tests.

-q qid
Specifies the name of a temporary administration queue that is used for stopping the adapter daemon. This parameter requires a value for the `[epicreceivetimout]` parameter in the kernel's configuration file, and an existing administration queue. If `[-noretry]` is specified, `-q` is ignored. For further details, refer to "Stopping the kernel".

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 75.

Stopping the kernel

To stop the kernel, stop each of the adapter daemons and trace servers. If the adapter daemon has been started with the `-q` option, you can use the **aqmstpad** command-line utility to stop an adapter daemon. To stop a trace server, you must use **Ctrl+C**. If you did not set the `-q` option when the adapter daemon was started, also use **Ctrl+C** to stop the adapter daemon.

The **aqmstpad** utility communicates with the administration queue that was created for the current application when the adapter daemon was started. It uses the unique identifier of the -q parameter to identify the corresponding administration queue.

To stop the the kernel by using the **aqmstpad** utility, perform the following steps:

Step 1. Open a command prompt. For each adapter daemon, enter the following command:

- On Windows systems:

```
aqmstpad.bat -a application_identifier -q qid
```

- On UNIX and OS/400:

  ```
  aqmstpad.sh -a application_identifier -q qid
  ```

  where

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

  -q qid
  : Specifies the unique name of a temporary administration queue that has been created when the adapter daemon was started and is now used to stop the adapter daemon.

Step 2. For each trace server, enter the following command:

- On Windows systems: Go to the command prompt from which the trace server was started and press **Ctrl+C**.
- On UNIX and OS/400: Go to the terminal from which the trace server was started and press **Ctrl+C**.

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

List of available messages for the start and stop command-line utilities

This section lists the messages that are displayed if an error occurs when using the start and stop utilities. A message consists of the following parts:

- The message number (for example, AQM4003, where AQM is the product code for the MQSeries Adapter Kernel)
- The message text, which gives you a short description of the error that has occurred.
- The message description, which gives you additional information about the message.

AQM4000:

The request to stop the AdapterDaemon for application ID <applID> completed successfully. The used administration queue is <qid>.

Description: The appropriate administration queue has been identified and the request to stop completed successfully.

AQM4001:

The request to stop the AdapterDaemon for application ID <applID> and administration queue <qid> failed.

Description: There are several possible reasons why the request failed:

1. An administration queue with the name <qid> does not exist.
2. The function cannot access the involved queue manager for the specified application.
3. The function cannot modify the queue.

AQM4002:

The AdapterDaemon could not process the administration queue parameter <qid>.

Description: There are several possible reasons why the request failed:

1. The administration model queue was not created correctly. See “Creating an administration model queue” on page 70.
2. An internal error occurred while processing the administration queue.

AQM4003:

Usage: com.ibm.epic.adapters.eak.util.Aqmutil::main(): Aqmutil <-a ApplicationID> <-q AdminQueueName>, where `a` is the daemon's application ID for configuraton information, and `q` is the administration queue name.

Description: The usage message describes how to enter the Aqmutil command that is used by the stop command-line utility **aqmstpad**.

AQM4004:

No <epicreceivetimeout> value was found for ApplicationID <applID>, bodyType <bt>, and bodyCategory <bc>. The AdapterDaemon did not start successfully.

Description: The <epicreceivetimeout> value was not found in the configuration for the specified parameter values. If you set the -q option, a timeout value must be set in the configuration file. The AdapterDaemon failed to start.

Note: The *Problem Determination Guide* contains a description of all available messages.

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 7.

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 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, potentially 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 36 for common problems that you can encounter during verification of installation and for potential responses.

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 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, potentially including a more recent edition of this book. The direct link to the MQSeries library page is 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 www.ibm.com/servers/eserver/iseries/library/. Also see the OS/400–specific books available from the MQSeries library Web site at 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 www.openapplications.org/
- The Extensible Markup Language (XML) 1.0 W3C Recommendation at www.w3.org/TR/1998/Rec-xml-19980210

These are not IBM Web sites.

Appendix A. Communications modes

This appendix provides information on the communications modes supported by MQSeries Adapter Kernel and on the Java classes that are used to support them. Some of the communications modes are provided as convenience modes with default formatters. See Table 10 on page 82 for the default formatters that are used with the convenience modes.

The following communications 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 Message Service (JMS). See "Using JMS object storage" on page 83 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 9 on page 82 lists the communications 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 communications mode; in this case, the class itself is used to support communication.

Table 9. Communications modes and supporting Java classes

Communications 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 10 lists the communications modes and their associated formatter interfaces. Table 11 cross-references formatter interfaces, formatter class names, 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 10. Communications modes and formatter interfaces

Communications mode	Formatter interface	Default formatter
MQPP	`MQFormatterInterface`	`MQNMXMLFormatter`
MQRFH1	`MQFormatterInterface`	`MQNMRFH1Formatter`
MQRFH2	`MQFormatterInterface`	`MQNMRFH2Formatter`
MQBD	`MQFormatterInterface`	`MQNMBDFormatter`
MQ	`MQFormatterInterface`	`MQNMXMLFormatter`
JMS	`JMSFormatterInterface`	`JMSNMRFH2Formatter`
FILE	`StringFormatterInterface`	`NMXMLFormatter`

Table 11. Formatter interfaces, formatter class names, and purposes

Formatter interface	Formatter class name	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
	`JMSNMBDFormatter`	Body data only
`StringFormatterInterface`	`NMXMLFormatter`	EpicMessage as XML

Table 12 on page 83 lists the supported LMS classes and their degree of transactional support. See “Transactional capabilities” on page 20 for information about using transactions with MQSeries Adapter Kernel.

Table 12. 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`.

```
#
# aqmjmscreatesample.scp 1.00 09Mar01
# Used for MQSeries Adapter Kernel
# Sample AQM JMS Configuration.
#
# Copyright (c) 2001 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
#
#
# This is a script to use with the JMS administration (JMSAdmin) tool
# which comes with MQSeries Support pac MA88.
# This tool requires the JMSAdmin.config to be set to use either
# FSCONTEXT (file) or LDAP. This script is setup to work with
# FSCONTEXT, but will work with LDAP with the following changes:
# - Change the "-" signs to "=".  Example: define ctx(o-ePIC)
#   becomes define ctx(o=ePIC)
# - In LDAP the contexts have to already be defined using the
#   LDAP administration tool.  For example you do not need
#   to "define ctx(o=ePIC) but only change into it with the
#   "change ctx(o-ePIC)" command.
# - There are some notes in the following script which highlight
#   differences when using LDAP.
#
#
```

```
# Example usage: MQSeries root\java\bin\jmsadmin.bat < aqmjmscreatesample.scp
#
# Some helpful commands:
# "display ctx" will display the context's of the context you are
# currently in.
# "=UP" means return to the parent context.  Example: change ctx(=UP)
# "=INIT" means return to root context. In this example one directory level
# above o-ePIC.  Example: change ctx(=INIT)
# "define xxx" is for creating either a context or object.
# "change xxx" is for changing/moving into the context.
#
# 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.
# Note: These two steps are not required for LDAP.
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.
# Note: These two steps are not required for LDAP.
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 MQSeries or JMS communications modes. For vlaid communications modes, see "Appendix A. Communications modes" on page 81.
- 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
  - With tracing enabled

  Note: See "Appendix C. Message headers" on page 87. 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 define the configuration.

MQSeries:

- Not using MQSeries clusters.

  Note: See "Appendix C. Message headers" on page 87. 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 87. 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 9 for which headers are used under which circumstances.

This appendix lists and describes the message header fields.

MQSeries Adapter Kernel message descriptor header

Header values used by MQSeries Adapter Kernel. These values are placed in message-holder objects. The **Propagated in replies?** column lists whether or not the value is propagated back to the source application in a reply message when the source application requests a reply. Some values are used only with WebSphere Business Integrator.

Table 13. MQSeries Adapter Kernel header

Header name	Propagated in replies?	Meaning or usage
UniqueID	No	Unique identifier for each message.
TransactionID	Yes	Transaction identifier shared by each message and its reply, if any. Equivalent to an Extricity PublicProcessID or a DataInterchange (DI) ApplicationID.
MessageType	No	Used for gateway and log/trace/exception messages.
SourceLogicalID	No	Logical identifier of the source application. Equivalent to reserved names in DI, Partner Agreement Manager (PAM), and Business Flow Manager (BFM).
DestinationLogicalID	No	Logical identifier of the target application. For DI and PAM, the default value is `none`, but can be overridden.
RespondToLogicalID	Yes	Used for responses.
CorrelationID	No	Reserved use.
GroupStatus	No	Reserved use.
ProcessingCategory	No	Reserved use.
QosPolicy	No	Reserved use.
DeliveryCategory	No	Reserved use.
AckRequested	No	Determines whether or not the source application requests a reply message.
PublicationTopic	No	Reserved use.

Table 13. MQSeries Adapter Kernel header (continued)

Header name	Propagated in replies?	Meaning or usage
SessionID	No	Reserved use.
EncryptionStatus	No	Determines type of body encryption and signature.
TimeStampCreated	No	Time and date when the message was created.
TimeStampExpired	No	Time and date after which the message is no longer meaningful. A value of -1 means no expiration.
Size	No	Reserved use.
BodyType	No	Represents the specific purpose of the message.
BodyCategory	No	Represents the message's application type.
BodySecondaryType	No	Reserved use.
UserArea	No	General area for user data.
RelatedSubjectID	No	Used for interprocess correlation.
ExternalID	No	Identifier of the current owner (for example, a user or trading partner) outside the application environment.
InternalID	No	Identifier of the current owner (for example, a user or trading partner) inside the application environment.
BodySignature	No	Reserved use.
TransportCorrelationID	Yes	Reserved use.

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 12 for details.

Table 14. 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.

Table 14. MQSeries header (continued)

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.
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 15. 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.
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.

Table 15. MQSeries Integrator version 1 header — RFH1 (continued)

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. <msd> folder — message service domain
4. <usr> folder — application (user) defined properties
5. Data section, which contains the kernel header and message body data

Table 16. MQSeries Integrator version 2 header — RFH2

Section or field	**Meaning or usage**
Fixed portion	Used as specified in MQSeries Integrator version 2.
<mcd>	Contains elements that describe the structure of the message data. Follow the MQSeries Integrator version 2 rules.
<msd>	Identifies how to handle a message. Possible values: XML or mrm.
<fmt>	Identifies the message format. If <msd> is XML, the format is XML. If <msd> is mrm, the format is xwf. “XML elements used in the configuration file” on page 51 describes how to enable the <epicrfh2messageset> tag.
<set>	Identifies the message set. For both, XML and mrm, the identifier is bodyCategory. Follow the MQSeries Integrator version 2 rules.
<type>	Identifies the message type. For both, XML and mrm, the identifier is bodyType. Follow the MQSeries Integrator version 2 rules.
<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.

Table 16. MQSeries Integrator version 2 header — RFH2 (continued)

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 97 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 potentially out of date.

See “The configuration file” on page 49 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 1.01 09Mar01                                        -->
<!-- Used for MQSeries Adapter Kernel -->
<!-- Sample AQM Configuration.           -->
<!--  -->
<!-- Copyright (c) 2001 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 WebSphere Business Integrator 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>
<!-- Represents the type of command the "epiccommandclassname" -->
<!-- represents.  MQAKEAB is the EAB style interface.  MQAKEJB -->
<!-- is an EJB Service Session Bean.  No entry defaults to MQAKEAB -->
               <epiccommandtype>MQAKEAB</epiccommandtype>
<!-- If the "epiccommandtype" is "MQAKEJB" this entry is the -->
<!-- method name to invoke.  No entry defaults to "execute". -->
               <epiccommandejbmethod>execute</epiccommandejbmethod>
<!-- If the "epiccommandtype" is "MQAKEJB" this entry is the -->
<!-- parameter type for the method specified by "epiccommandejbmethod". -->
<!-- This will be the same datatype returned by the -->
<!-- TerminalDataContainerMapper.  No entry defaults -->
<!-- to "com.ibm.mqao.mqak.ejbclient.TDCGenericMapper". -->
               <epiccommandejbmethodparmtype>com.ibm.mqao.mqak.ejbclient.TDCGenericMapper
               </epiccommandejbmethodparmtype>
<!-- If the "epiccommandtype" is "MQAKEJB" this entry is the -->
<!-- URL where the EJB specified in "epiccommandclassname" -->
<!-- has been deployed in the form "IIOP://hostname:900/". -->
<!-- No entry defaults to "IIOP:///". -->
               <epiccommandejburl>IIOP:///</epiccommandejburl>
<!-- If the "epiccommandtype" is "MQAKEJB" this entry is the -->
<!-- name of the Initial Context Factory used to to lookup the -->
<!-- home name for the EJB specified in "epiccommandclassname". -->
<!-- No entry defaults to "com.ibm.ejs.ns.jndi.CNInitialContextFactory". -->
               <epiccommandejbinitialcontext>com.ibm.ejs.ns.jndi.CNInitialContextFactory
               </epiccommandejbinitialcontext>
<!-- If the "epiccommandtype" is "MQAKEJB" this entry is the -->
<!-- name of the Mapper the Worker uses for creating the  -->
<!-- "epiccommandejbmethodparmtype" object passed in to the  -->
<!-- "epiccommandejbmethod" in to the "epiccommandclassname". -->
<!-- No entry defaults to "com.ibm.mqao.mqak.ejbclient.TDCGenericMapper". -->
               <epiccommandejbmapper>com.ibm.mqao.mqak.ejbclient.TDCGenericMapper
               </epiccommandejbmapper>
<!-- Default destinations to send messages to. -->
        <!-- Single destination. -->
               <epicdestids>TEST2</epicdestids>
        <!-- Multiple destinations. -->
               <!--
               <epicdestids>
                    <Value>TEST2</Value>
                    <Value>TEST3</Value>
               </epicdestids>
               -->
                    <!-- Receive transport communications 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>
-->
<!-- Receive 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 receive mode, refers to the JMS queue object name for -->
<!-- this application to receive messages from. -->
               <!-- Requires the attribute describing the object plus the attribute's value. -->
               <!-- For JMS the attribute is 'cn'. -->
               <epicjmsreceivequeuename>cn=TEST1AIQ</epicjmsreceivequeuename>
<!-- JMS receive 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 attribute's value. -->
               <!-- For JMS the attribute is 'cn'. -->
               <epicjmserrorqueuename>cn=TEST1AEQ</epicjmserrorqueuename>
<!-- JMS receive 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 attribute's 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 the AdapterDaemon will start. -->
  <!-- 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>
```

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 64 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>
   <!-- Receive 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 48 for additional information about this file. The `aqmsetup` file is located in the `samples` directory of the kernel’s root installation directory.

```
#
# aqmsetup 1.01 01/03/27
# Sample AQM Adapter runtime parameter configuration file entries.
#
# Copyright (c) 2001 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 Integrator(WSI) product 5724-A78 LDAP
# Directory Services or configuration file. No entry defaults to
# true (use configuration file). To use the WSI directory service
# set the value to false.  Refer to the WSI documentation for
# specifics on using the directory service.
#AdapterDirectoryUseFileFlag=true
# When using Websphere Business Integrator(WSI) product 5724-A78 LDAP
# Directory Services this additional entry is required. Refer to the
# WSI documentation for specifics on using the directory service.
#DirectoryServices=ChangeToDestDir/samples/DirectoryServices.properties
# Location of configuration file aqmconfig.xml when not using
# the Websphere Business Integrator(WSI) product 5724-A78 LDAP
# Directory Services.
# No entry defaults to current directory.
#AQMConfig=ChangeToDestDir/samples
#
####################################################################
####################################################################
# 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=ChangeToDestDir/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 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
```

Appendix F. Using a J2EE connector in an MQAK microflow

This appendix illustrates how to use a J2EE connector in an MQSeries Adapter Kernel microflow in a non-managed scenario. In the following, the mySAP.com J2EE connector is used as an example.

1. "Preparing the software environment"
2. "Creating a Java proxy bean to use the SAP connector"
3. "Creating an MQAO adapter to use the Bean" on page 108
4. "Using the adapter in MQAK" on page 113

Preparing the software environment

To prepare the software environment, perform the following:

1. Install Visual Age for Java V4.0
2. Install Connector for SAP R/3
3. Install EAB 4.0 Beta for J2EE Connectors

   Note: This includes *J2EE Connector for SAP R/3* and *EAB 4.0 for J2EE*.
4. In Visual Age for Java, add the following features:
   a. Access Builder for SAP R/3 Libraries
   b. Connector for SAP R/3
5. Install MQSeries Adapter Builder
6. Install MQSeries Adapter Kernel
7. Install mySAP.com connector
8. Read the MQSeries Adapter Kernel readme's advice about the `xml4j.jar` file.
9. Install Java™ 1.3

Creating a Java proxy bean to use the SAP connector

To create a bean that uses SAP connector via the Common Client Interface (CCI), perform the following steps:

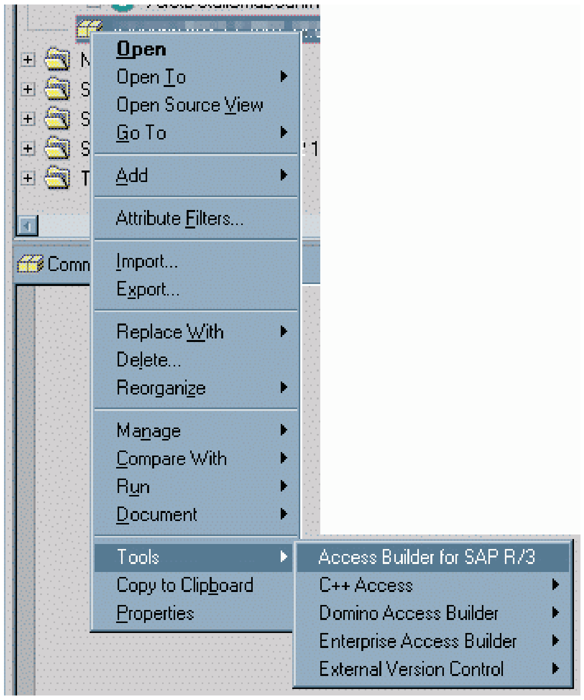
1. "Build the Java proxy beans"
2. "Create the command bean" on page 104
3. "Promote properties that must be visible in MQAB" on page 107
4. "Export the package" on page 108

Build the Java proxy beans

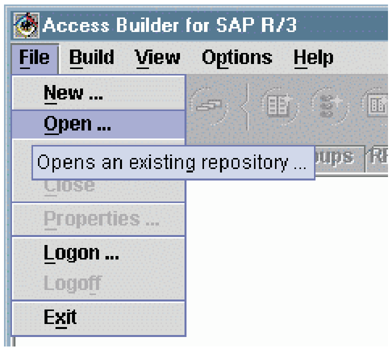
In Visual Age for Java (VAJ):

1. Create an empty package.

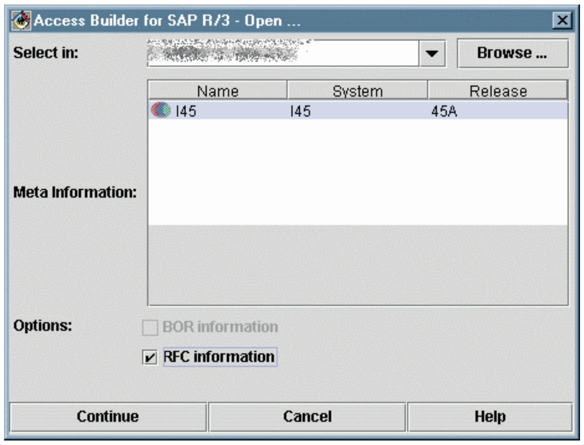
2. Select the Package, right-click, select Tools / "Access Builder for SAP R/3".



3. In the Access Builder select File / Open...



4. Select "I45" with the option "RFC information" checked.

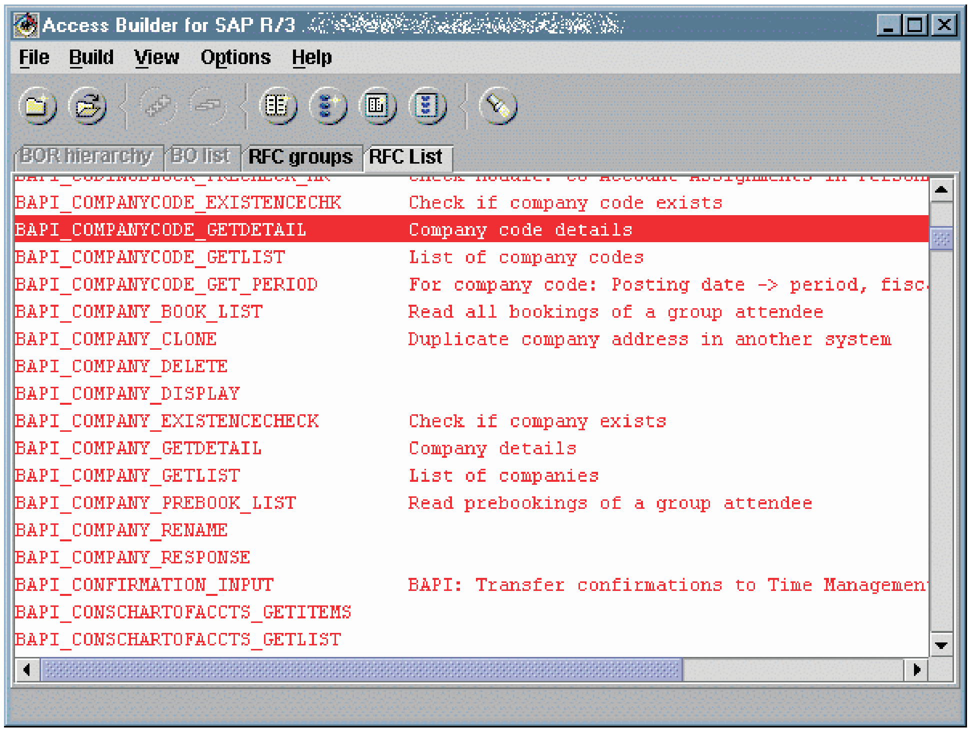


Name	System	Release
I45	I45	45A

BOR information
[✓ RFC information]

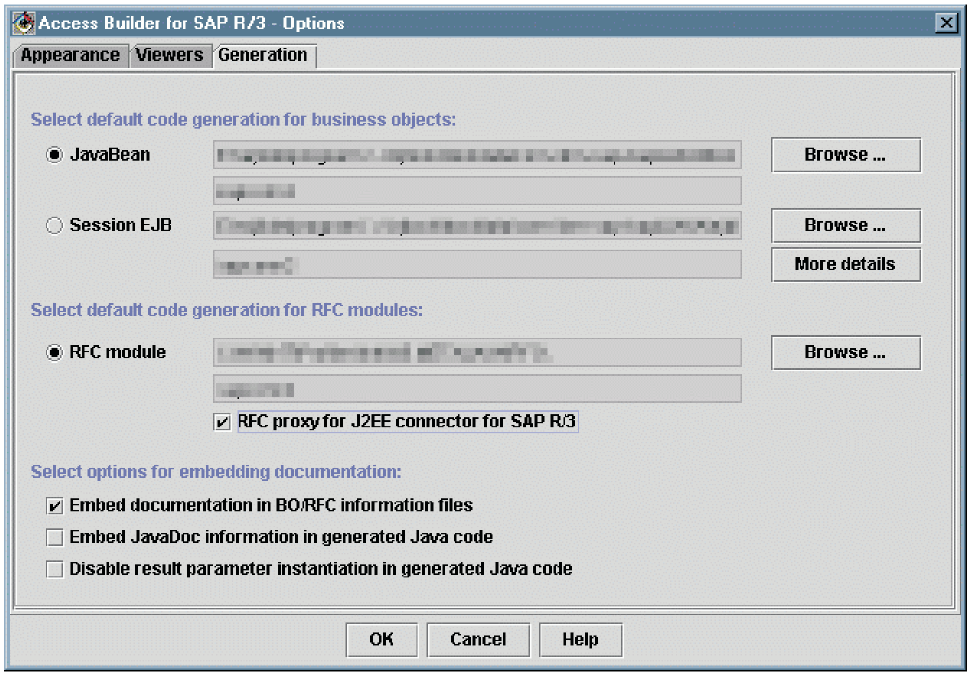
5. Click Continue to load the RFC information.

6. In the "RFC List" select the RFCs you need.


Access Builder for SAP R/3
File Build View Options Help
BOR hierarchy BO list RFC groups RFC List

BAPI_COMPANYCODE_EXISTENCECHK	Check if company code exists
BAPI_COMPANYCODE_GETDETAIL	Company code details
BAPI_COMPANYCODE_GETLIST	List of company codes
BAPI_COMPANYCODE_GET_PERIOD	For company code: Posting date -> period, fisc
BAPI_COMPANY_BOOK_LIST	Read all bookings of a group attendee
BAPI_COMPANY_CLONE	Duplicate company address in another system
BAPI_COMPANY_DELETE	
BAPI_COMPANY_DISPLAY	
BAPI_COMPANY_EXISTENCECHECK	Check if company exists
BAPI_COMPANY_GETDETAIL	Company details
BAPI_COMPANY_GETLIST	List of companies
BAPI_COMPANY_PREBOOK_LIST	Read prebookings of a group attendee
BAPI_COMPANY_RENAME	
BAPI_COMPANY_RESPONSE	
BAPI_CONFIRMATION_INPUT	BAPI: Transfer confirmations to Time Managemen
BAPI_CONSCHARTOFACCTS_GETITEMS	
BAPI_CONSCHARTOFACCTS_GETLIST	

7. Select Options / Generation to display the generation options screen and select the option "RFC proxy for J2EE connector for SAP R/3".


Access Builder for SAP R/3 - Options
Appearance Viewers Generation
Browse ...
Browse ...
More details
Browse ...
OK Cancel Help

Select default code generation for business objects:

[✓ JavaBean] Session EJB

Select default code generation for RFC modules:

[✓ RFC module]

[✓] RFC proxy for J2EE connector for SAP R/3

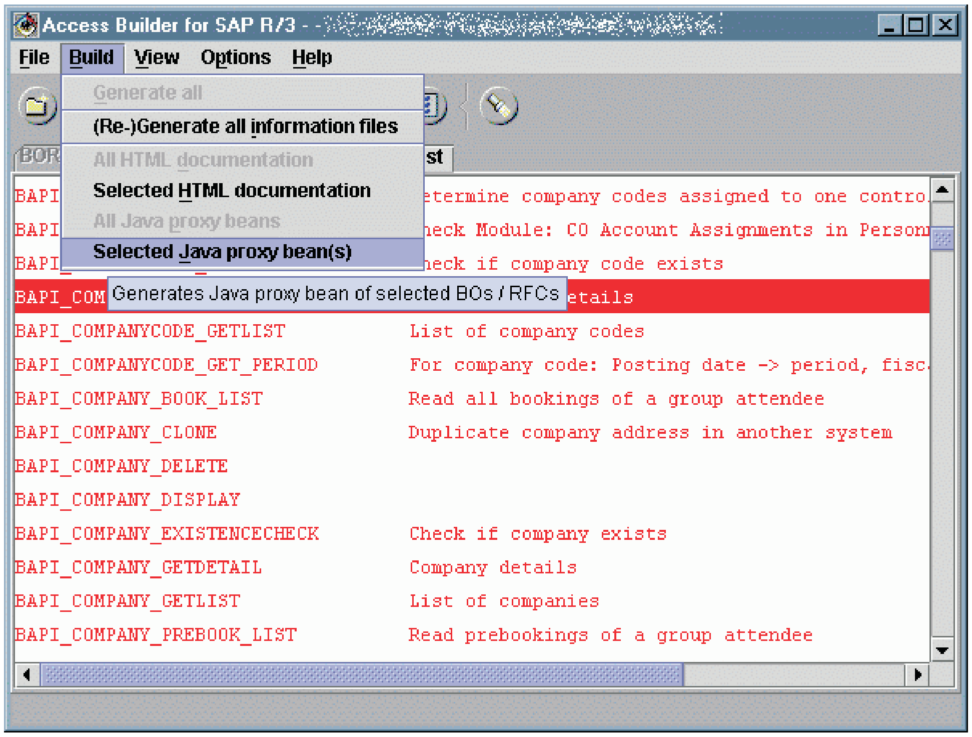
Select options for embedding documentation:

[✓] Embed documentation in BO/RFC information files

[ ] Embed JavaDoc information in generated Java code

[ ] Disable result parameter instantiation in generated Java code

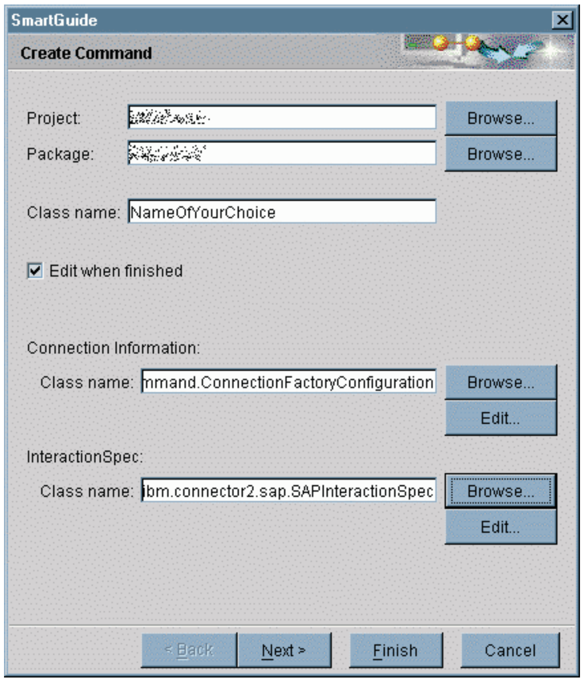
8. Build the proxy beans for the selected RFCs.


Access Builder for SAP R/3 -
File
Build
View
Options
Help
Generate all
(Re-)Generate all information files
All HTML documentation
Selected HTML documentation
All Java proxy beans
Selected Java proxy bean(s)
Generates Java proxy bean of selected BOs / RFCs

```
BAPI                               etermine company codes assigned to one contro
BAPI                               heck Module: CO Account Assignments in Person
BAPI                               heck if company code exists
BAPI_COM                           etails
BAPI_COMPANYCODE_GETLIST           List of company codes
BAPI_COMPANYCODE_GET_PERIOD        For company code: Posting date -> period, fisc
BAPI_COMPANY_BOOK_LIST             Read all bookings of a group attendee
BAPI_COMPANY_CLONE                 Duplicate company address in another system
BAPI_COMPANY_DELETE
BAPI_COMPANY_DISPLAY
BAPI_COMPANY_EXISTENCECHECK        Check if company exists
BAPI_COMPANY_GETDETAIL             Company details
BAPI_COMPANY_GETLIST               List of companies
BAPI_COMPANY_PREBOOK_LIST          Read prebookings of a group attendee
```

Create the command bean

1. In VAJ, select Package, right-click, select Tools / Enterprise Access Builder / Create Command to start the appropriate Smart Guide.
2. The Project and Package names should be OK. Give the class a name of your choice. Check the ″Edit when finished″ box and press the ″Browse″ button to set the Connection Information Class Name.



Project:

Package:

Class name: NameOfYourChoice

☑ Edit when finished

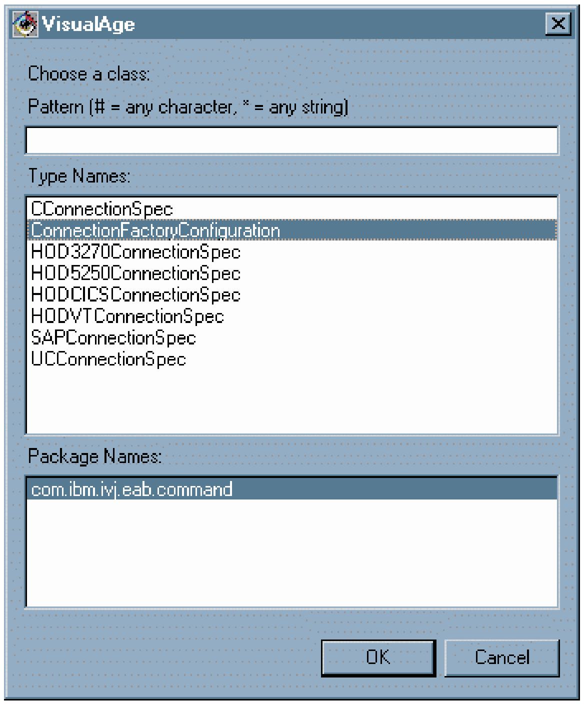
Connection Information:

Class name: mmand.ConnectionFactoryConfiguration

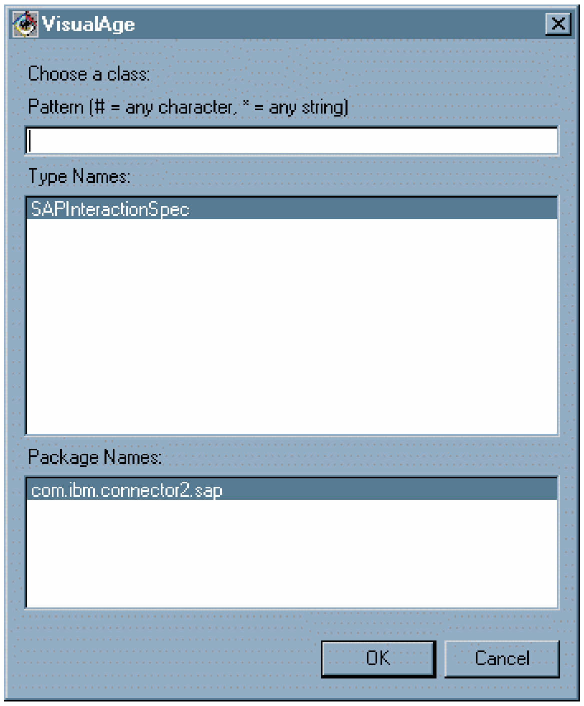
InteractionSpec:

Class name: ibm.connector2.sap.SAPInteractionSpec

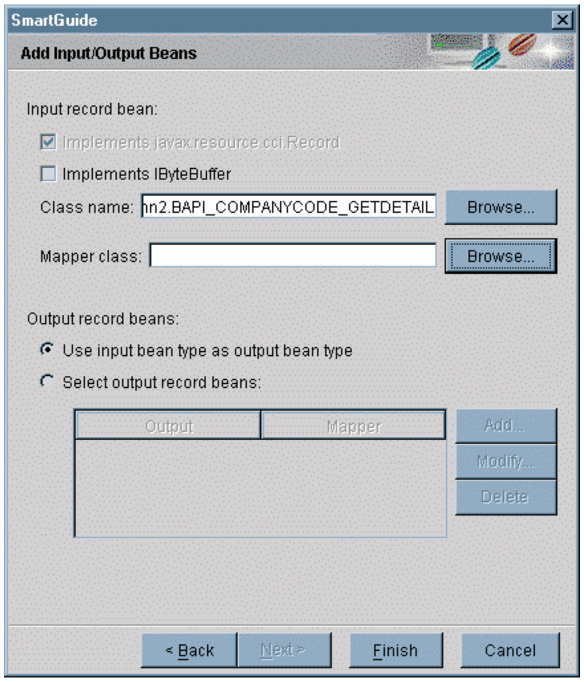
3. Select "ConnectionFactoryConfiguration" and press the "OK" button.



4. Using "Browse", for the InteractionSpec Class Name select the `SAPInteractionSpec`, package `com.ibm.connector2.sap`, then "OK".



5. Click "Next" to get to the "Add Input/Output Beans" panel.
6. Uncheck the "Implements IByteBuffer" checkbox and click the "Browse" button. In the pop-up window, select `BAPI_COMPANYCODE_GETDETAIL` and your package name. Click "OK" to add them to the Class name field. Select "Use input bean type as output bean type" and click the "Finish" button to start the command editor.


SmartGuide

Add Input/Output Beans

Input record bean:

- [✓] Implements javax.resource.cci.Record
- [ ] Implements IByteBuffer

Class name: nn2.BAPI_COMPANYCODE_GETDETAIL

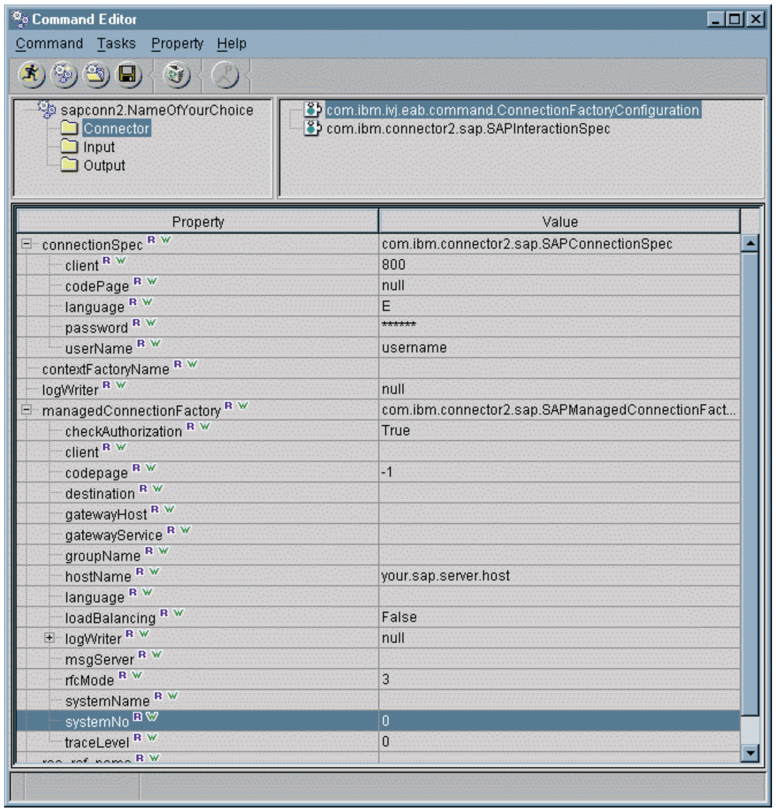
Mapper class:

Output record beans:

- (●) Use input bean type as output bean type
- ( ) Select output record beans:

Output	Mapper

7. In the command editor:
   a. Select ″Connector″ in the top left pane, ″...ConnectionFactoryConfiguration″ in the top right pane ″ConnectionSpec″ in the Property column of the bottom pane and ″com.ibm.connector2.sap.SAPConnectionSpec″ in the Value column of the bottom pane.



sapconn2.NameOfYourChoice
- Connector
- Input
- Output

com.ibm.ivj.eab.command.ConnectionFactoryConfiguration
com.ibm.connector2.sap.SAPInteractionSpec

Property	Value
connectionSpec	com.ibm.connector2.sap.SAPConnectionSpec
client	800
codePage	null
language	E
password	******
userName	username
contextFactoryName	
logWriter	null
managedConnectionFactory	com.ibm.connector2.sap.SAPManagedConnectionFact...
checkAuthorization	True
client	
codepage	-1
destination	
gatewayHost	
gatewayService	
groupName	
hostName	your.sap.server.host
language	
loadBalancing	False
logWriter	null
msgServer	
rfcMode	3
systemName	
systemNo	0
traceLevel	0

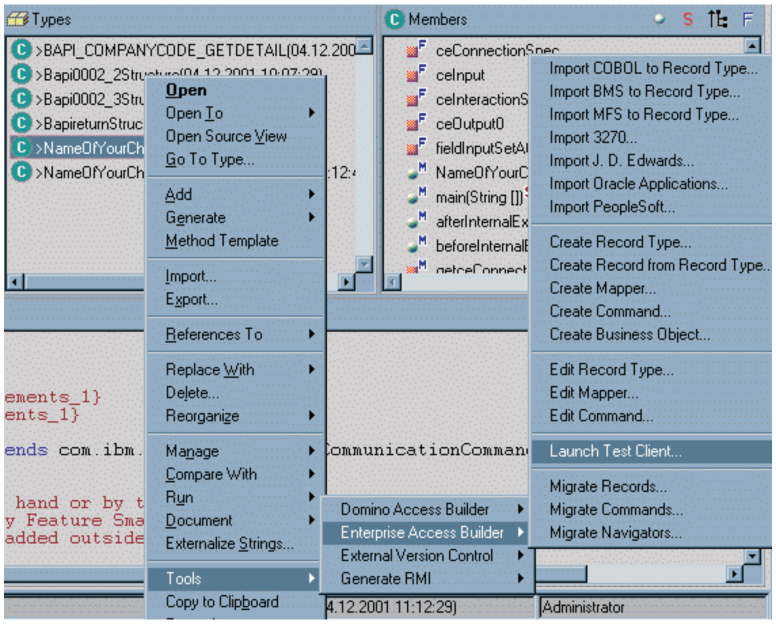
b. Expand the ConnectionSpec entry and set the appropriate values to connect to your SAP R/3 system.

c. For the ″managedConnectionFactory″ parameter select ″com.ibm.connector2.sap.SAPManagedConnectionFactory″.

d. Expand the tree and enter hostName and systemNo appropriately.

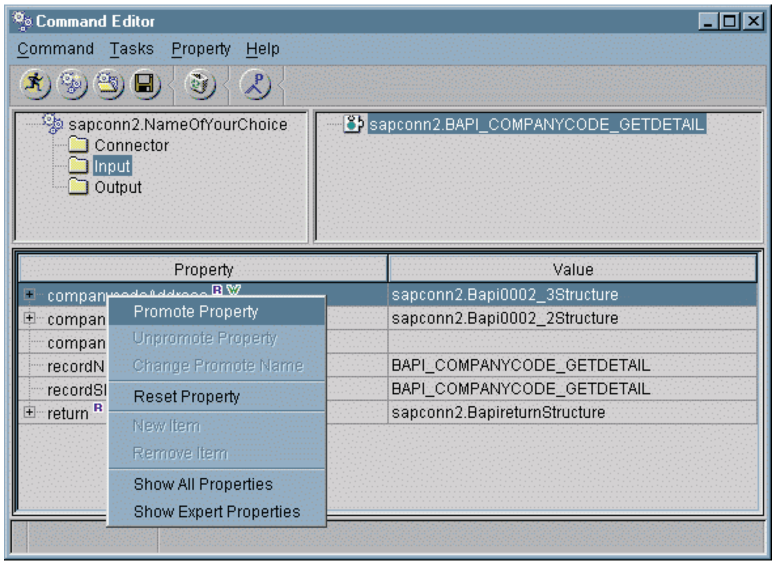
e. Select ″Command″ / ″Save″ to end the command editor.

8. To use the test client to verify what you have done so far, select the created command, right click and select Tools/Enterprise Access Builder/Launch Test Client.



Promote properties that must be visible in MQAB

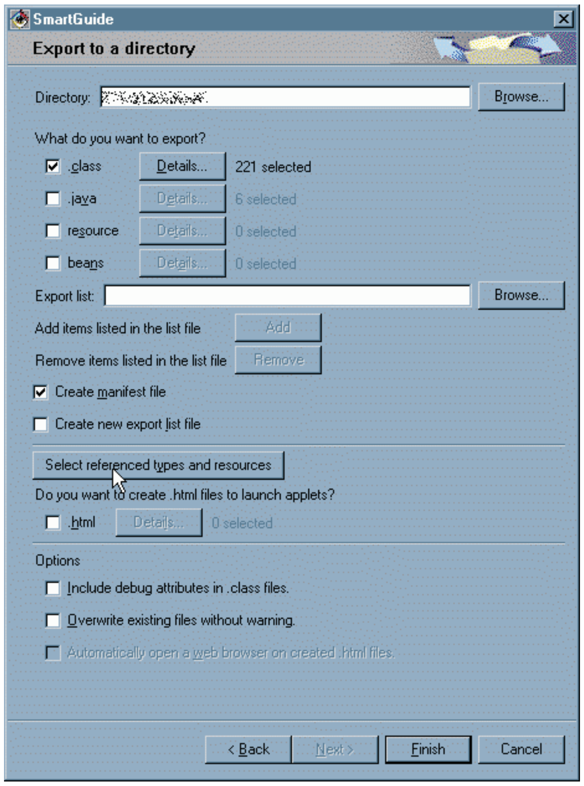
Using the Command Editor, promote the properties that must be visible in MQAB. In this sample, companycodeAddress, companycodeDetail, and companycodeid should be made visible for Input and Output.



Property	Value
compan	sapconn2.Bapi0002_3Structure
compan	sapconn2.Bapi0002_2Structure
compan	
recordN	BAPI_COMPANYCODE_GETDETAIL
recordSl	BAPI_COMPANYCODE_GETDETAIL
return	sapconn2.BapireturnStructure

Export the package

1. Select Package, right-click, select Export.
2. Select ″Directory″ as export destination.
3. Select a directory; only the class files need to be exported.



SmartGuide

Export to a directory

Directory: [redacted]

What do you want to export?

- [✓] .class — Details... — 221 selected
- [ ] .java — Details... — 6 selected
- [ ] resource — Details... — 0 selected
- [ ] beans — Details... — 0 selected

Export list:

Add items listed in the list file — Add

Remove items listed in the list file — Remove

- [✓] Create manifest file
- [ ] Create new export list file

Select referenced types and resources

Do you want to create .html files to launch applets?

- [ ] .html — Details... — 0 selected

Options

- [ ] Include debug attributes in .class files.
- [ ] Overwrite existing files without warning.
- [ ] Automatically open a web browser on created .html files.

4. Click the ″Select referenced types and resources″ button.
5. Click ″Finish″.

Creating an MQAO adapter to use the Bean

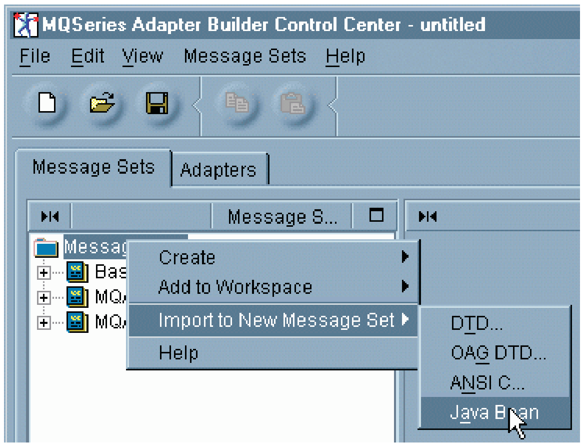
To import the command bean into the adapter builder to get the interfaces, define a microflow, and create an MQAO adapter, perform the following steps:

1. “Import the command bean”
2. “Create a new class type” on page 109
3. “Define a new microflow type” on page 110
4. “Create Java service adapter” on page 111
5. “Generate Java service adapter” on page 112
6. “Compile the adapter code” on page 113

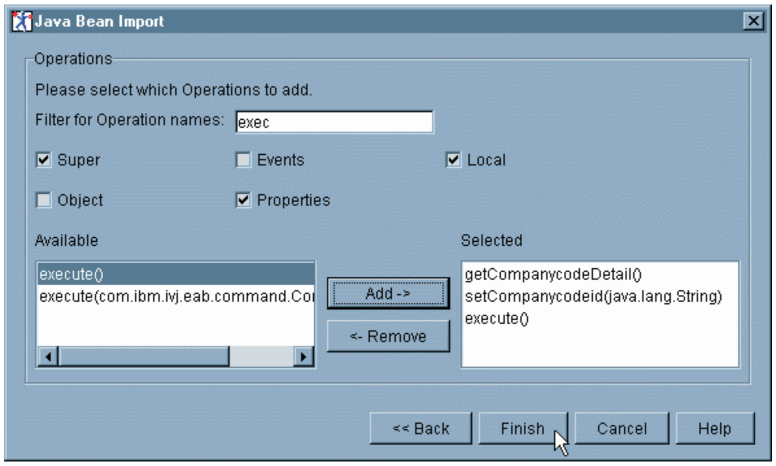
Import the command bean

To import the command bean message set, perform the following in the MQSeries Adapter Builder Control Center:

1. On the ″Message Sets″ tab, select ″Message Sets″, right click, select ″Import to New Message Set″, then ″Java Bean″.


MQSeries Adapter Builder Control Center - untitled
File Edit View Message Sets Help
Message Sets
Adapters
Message S...
Messag
Bas
MQ
MQ
Create
Add to Workspace
Import to New Message Set
Help
DTD...
OAG DTD...
ANSI C...
Java Bean

2. From the directory, just created during Export from VAJ, select the command bean class file.
3. On the "Search Directories" panel, add the "`eab/runtime35`" subdirectory of your VAJ installation to the search path.
4. In the "Java Bean Import" window, check the "Properties" box, select the methods to be used using the "Add ->" button, and click the "Finish" button.


Java Bean Import

Operations

Please select which Operations to add.

Filter for Operation names: exec

[✓] Super   [ ] Events   [✓] Local

[ ] Object   [✓] Properties

Available:
execute()
execute(com.ibm.ivj.eab.command.Co

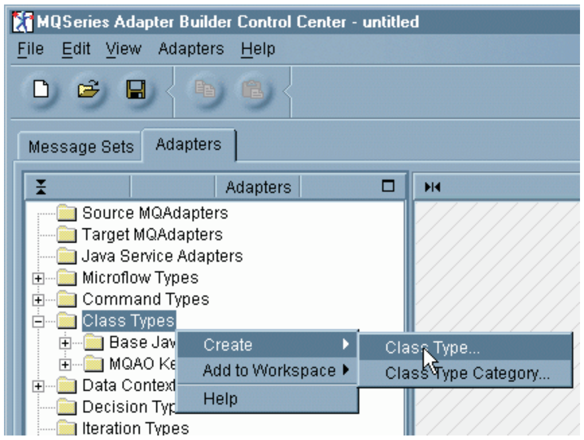
Selected:
getCompanycodeDetail()
setCompanycodeid(java.lang.String)
execute()

5. If you exported the package using "Select referenced types...", only "`vaj\eab\runtime35`" has to be added to the `classpath`.

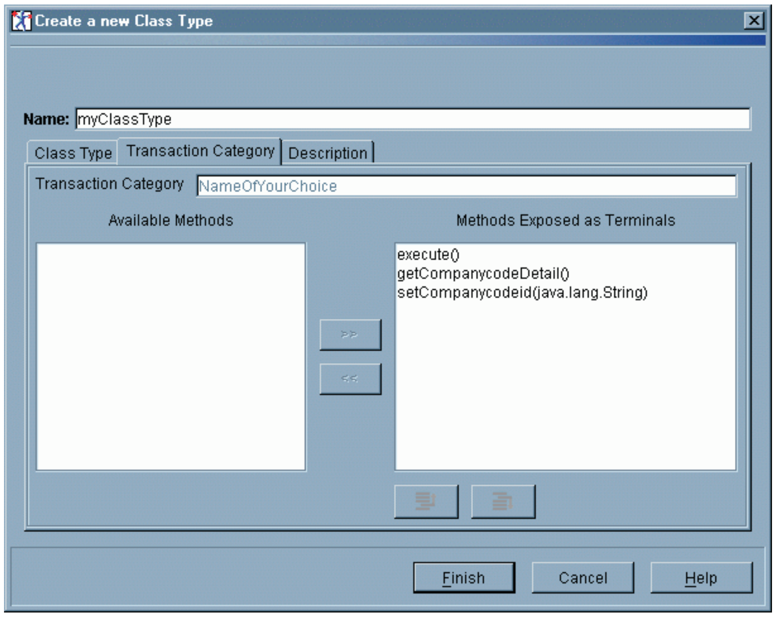
Create a new class type

To create a new class type, using the command bean's message set, perform the following in the MQSeries Adapter Builder Control Center:

1. On the "Adapters" tab, select "Class Types", then "Create", then "Class Type...".



2. Select the message set that you just created and select from the available transaction types.
3. On the ″Transaction Category″ tab, select which of the available methods will be exposed as Terminals.



Name: myClassType

Transaction Category: NameOfYourChoice

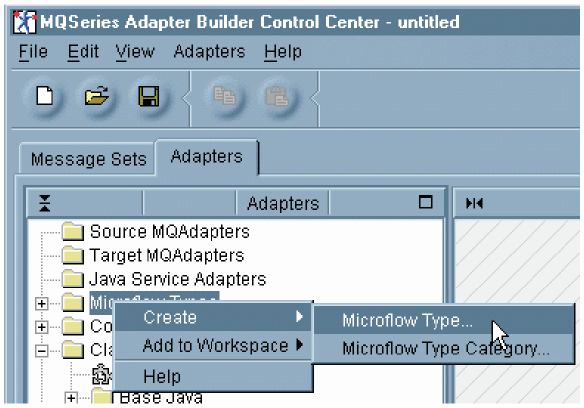
```
execute()
getCompanycodeDetail()
setCompanycodeid(java.lang.String)
```

4. Name the class type and click the ″Finish″ button.

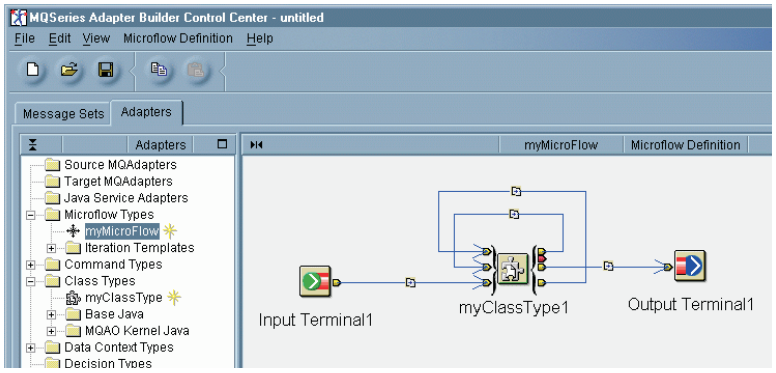
Define a new microflow type

To create and define a new microflow type, perform the following in the MQSeries Adapter Builder Control Center:

1. On the ″Adapters″ tab, select ″Microflow Types″, then ″Create″, then ″Microflow Type...″.



2. Enter a name and description for your microflow type.
3. Drag Input and Output terminals, and the previously created class type from the tree view on the left to the workspace window.

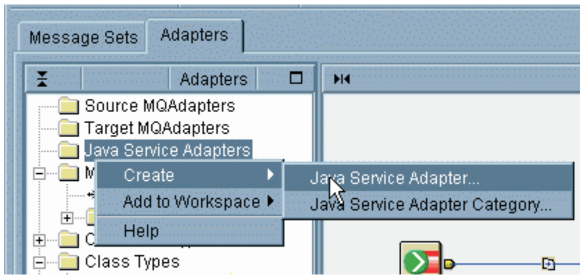


4. Connect ″Input Terminal1″ to the ″setCompanyCodeid()_Request″ input Terminal of ″myClassType1″, add a map to this connection and set the `arg_0` property to an existing ″CompanyCodeid″, for example `1000`.
5. Connect the ″setCompanycodeid()_Resp″ output terminal to the ″execute()_Request″ input terminal.
6. Connect the ″execute()_Resp″ output terminal to the ″getCompanycodeDetail()_Request″ input terminal.
7. Connect the ″getCompanycodeDetail()_Resp″ output terminal to ″Output Terminal1″ and map the return structures.

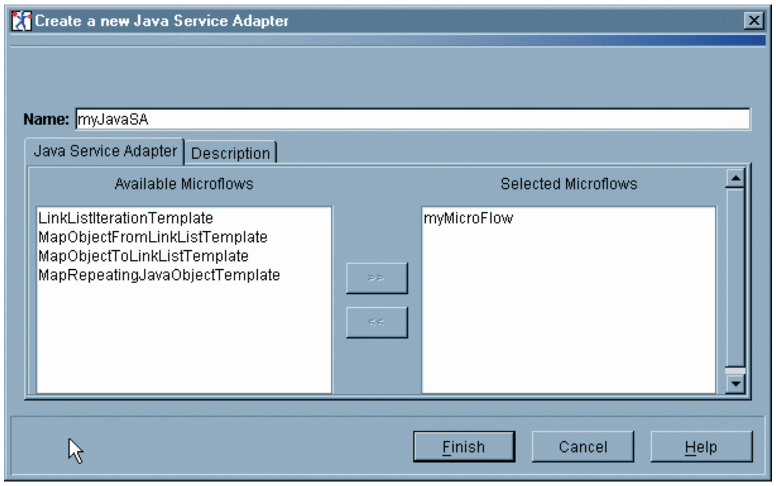
Create Java service adapter

To create the Java service adapter for your new microflow type, perform the following in the MQSeries Adapter Builder Control Center:

1. On the ″Adapters″ tab, select ″Java Service Adapters″, right click, select ″Create″ – ″Java Service Adapters...″.



2. Enter a name for the Service Adapter.



Name: myJavaSA

Available Microflows:
LinkListIterationTemplate
MapObjectFromLinkListTemplate
MapObjectToLinkListTemplate
MapRepeatingJavaObjectTemplate

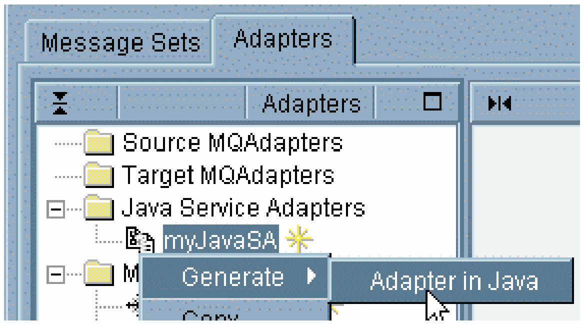
Selected Microflows:
myMicroFlow

3. Select your microflow from the list of "Available Microflows", and add it to the "Selected Microflows" list.
4. Click the "Finish" button.

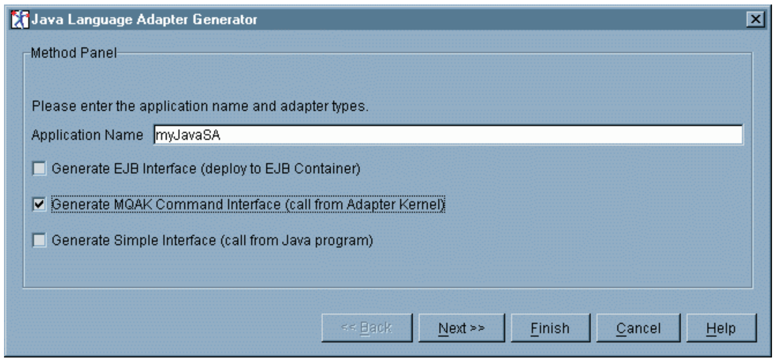
Generate Java service adapter

To generate the Java service adapter, perform the following in the MQSeries Adapter Builder Control Center:

1. Expand the "Java Service Adapters" folder.
2. Select the Service Adapter that you named in step 2, right-clisk on it.
3. Select "Generate", then "Adapter in Java".



4. Enter a name for the adapter, and select the option "Generate MQAK Command Interface ...".


Java Language Adapter Generator

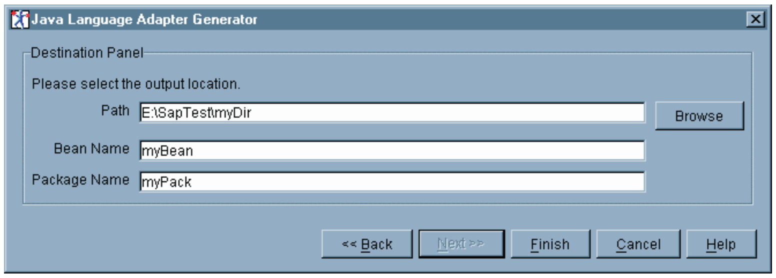
Method Panel

Please enter the application name and adapter types.

Application Name: myJavaSA

- [ ] Generate EJB Interface (deploy to EJB Container)
- [✓] Generate MQAK Command Interface (call from Adapter Kernel)
- [ ] Generate Simple Interface (call from Java program)

5. Click ″Next″ and skip the ″Import Panel″.
6. Specify the destination path, bean name, and package name.


Java Language Adapter Generator

Destination Panel

Please select the output location.

Path: E:\SapTest\myDir

Bean Name: myBean

Package Name: myPack

7. Click ″Finish″ to generate the adapter code.

Compile the adapter code

To compile the adapter code that you have just generated, perform the following from a command prompt:

1. Make sure that the following are in the `CLASSPATH`:
   - `MQAOJFramework.jar`
   - The directory where the Java Service adapter was created in step 6.
   - The directory where you exported the package in step 3 on page 108.
2. Change to the ″message″ subdirectory of the directory where you generated the Java Service adapter in step 6, then enter the command:
   ```
   javac *.java
   ```
3. Change to the ″mymicroflow″ subdirectory, then enter the command again:
   ```
   javac *.java
   ```

Using the adapter in MQAK

To deploy the adapter bean, perform the steps described in:

1. “Preparing the adapter’s environment”
2. “Starting the service adapter” on page 114

Preparing the adapter’s environment

To configure the environment for the command shell or prompt where the target adapter will be run:

1. Verify that the following are defined in the `classpath`:
   - The ″`bin`″ subdirectory of MQAK.
   - The MQAB file `MQAOJFramework.jar`.

- The directory where you exported the package in step 3 on page 108.
- For non-managed execution, add the Connector for SAP R/3 files: `ccf2.jar`, `conn4sap.jar`, and `ivjsap35.jar`.

2. Add the following to the `path` variable:
   - The VAJ ″eab\bin\″ subdirectory.
   - The MQAK ″`bin`″ subdirectory.
3. Set the environment variable ″`aqmsetupfile`″ to point to a suitable setup file.
4. If necessary, modify the adapter configuration file ″`aqmconfig.xml`″.

Starting the service adapter

To run the service adapter bean, using the command prompt you prepared in "Preparing the adapter's environment" on page 113, enter the command:

```
aqmstrad -a AdapterName -noretry
```

Where *AdapterName* is the name that you gave the adapter in 4 on page 112.

Your adapter should now work as designed.

Notices

Trademarks

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 potentially not included in the glossary. It can, however, potentially be found in the "Index" on page 123.

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 or Java 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 control flow; data flow; sequential navigation; conditional branching, including decision and iteration; data typing; storing data context; transformation of data elements; transactional control; logical operations; and custom code. You can reuse adapters that you have created.

See "message type" on page 120, "source application" on page 121, and "target application" on page 121.

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".

aqmsetup file. See "setup file" on page 121.

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 121 and "target logical identifier" on page 121.

application-neutral format. See "integration message" on page 120.

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 120 and "XML" on page 122.

BODs can be used by MQSeries Adapter Offering to define message bodies in its integration 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 120.

Body category also helps specify the message type. See "message type" on page 120.

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 120.

Body type also helps specify the message type. See "message type" on page 120.

configuration file. The `aqmconfig.xml` file, which contains most of the kernel's configuration values. See "The configuration file" on page 49 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.

communications 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 120.

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 122 and "integration 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 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.

Java service adapter. A type of Java-language adapter that, in a JMS Listener environment, provides the functions of an adapter daemon, worker, and target adapter.

JMS Listener. A component provided by the WebSphere Business Integrator product that enables tight integration between MQSeries Adapter Kernel and WebSphere Application Server Advanced Edition.

kernel. Synonymous with MQSeries Adapter Kernel.

logical message service. A component used by the native adapter to convert messages for transportation by the communications transport.

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 119 and "body type" on page 119.

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 119. 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 119.

push model of delivery. See “delivery models” on page 119.

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 119.

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 120, “application logical identifier” on page 119, 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 message according to how the adapter had been modeled.
- Transforms the integration 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 message, then a target adapter is potentially 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 119.

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 119.

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.

WebSphere Application Server Advanced Edition. A software product from IBM that enables the use of the Sun Microsystems Enterprise JavaBeans (EJB) specification. WebSphere Application Server Advanced Edition includes an EJB server, in which enterprise beans can run. Enterprise beans encapsulate the business logic and data used and shared by EJB clients. There are two types of enterprise beans: session beans, which encapsulate short-lived, client-specific tasks and objects; and entity beans, which encapsulate persistent data. A type of session bean called a worker message bean can be used on the target side of MQSeries Adapter Kernel.

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.

worker message bean. An enterprise bean that performs the function of a worker when WebSphere Application Server is used on the target side of the kernel.

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

Index

Readers' Comments — We'd Like to Hear from You

MQSeries® Adapter Kernel for Multiplatforms
Quick Beginnings
Version 1 Release 1

Publication No. GC34-5855-06

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
GC34-5855-06

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Str. 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

GC34-5855-06

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC34-5855-06