WebSphere MQ Everyplace

Application Programming Guide

Version 2.0.0.5

SC34-6278-01

WebSphere MQ Everyplace

Application Programming Guide

Version 2.0.0.5

SC34-6278-01

Take Note!

Before using this information and the product it supports, read the general information under Appendix D, “Notices”, on page 157

Second Edition (April 2003)

This edition applies to WebSphere® MQ Everyplace™ Version 2.0.0.5 (Program number: 5724-C77) and to all subsequent releases and modifications until otherwise indicated in new editions.

This document is continually being updated with new and improved information. For the latest edition, please see the WebSphere MQ family library Web page at `http://www.ibm.com/software/ts/mqseries/library/`.

Contents

Chapter 8. Security 79

Chapter 9. Java Message Service . . . 103

Chapter 10. Error and error handling 127

Chapter 11. Deployment of applications from Webshpere Studio Device Developer. 131

Chapter 12. Open Services Gateway Initiative 139

Appendix A. WebSphere MQ Everyplace Java programming examples 143

Appendix B. Applying maintenance to WebSphere MQ Everyplace 153

About this book

This book is a programming guide for the WebSphere MQ Everyplace product. It contains information on how to use the WebSphere MQ Everyplace Java™ class and C class libraries, that are described in *WebSphere MQ Everyplace Java Programming Reference*, on the product CD, and the client platform C APIs that are described in the WebSphere MQ Everyplace C Programming Reference.

It provides guidance to help you to decide which classes or APIs to use for common messaging tasks, and in many cases example code is supplied.

For more information on writing Java and C-based programs for WebSphere MQ Everyplace, refer to the *WebSphere MQ Everyplace Java Programming Reference* and the *WebSphere MQ Everyplace C Programming Reference*, on the product CD. The C APIs discussed in this manual are intended to provide client platform functionality. For information on writing C-based programs that provide server platform functionality, refer to the *WebSphere MQ Everyplace C Bindings Programming Guide*.

Chapter 1, "Introduction", on page 1 provides a brief introduction for those who are unfamiliar with the concepts and components of WebSphere MQ Everyplace. Chapter 3, "Running Applications", on page 11 provides help for setting up your environment, and shows you how to use examples to create applications. The rest of the book contains more detailed information about various aspects of programming with WebSphere MQ Everyplace.

If you choose to program in Java, you should use this book in conjunction with the *WebSphere MQ Everyplace Java Programming Reference* and existing books or manuals on Java and C programming.

To program in C, uset this book in conjunction with the WebSphere MQ Everyplace C Programming Reference.

This document is continually being updated with new and improved information. For the latest edition, please see the WebSphere MQ family library Web page at http://www.ibm.com/software/mqseries/library/.

License warning

WebSphere MQ Everyplace Version 2.0.0.5 is a toolkit that enables users to write WebSphere MQ Everyplace applications and to create an environment in which to run them.

Before deploying this product, or applications that use it, in a production environment, please make sure that you have the necessary licenses.

To use WebSphere MQ Everyplace on specified server platforms, other than for purposes of code development and test, you must obtain capacity-unit Use Authorizations so that you are licensed to use the program on each machine and machine upgrade. These licences are recorded on Proof of Entitlement documents and authorize the use of WebSphere MQ Everyplace according to published capacity unit and pricing group tables.

You require device platform use authorizations to use the product (other than for purposes of code development and test) on specified client platforms. These licences are recorded on Proof of Entitlement documents and authorize the use of WebSphere MQ Everyplace. However, they do not entitle you to use the WebSphere MQ Everyplace Bridge, or to run on the server platforms specified in the WebSphere MQ Everyplace pricing group lists published by IBM® and also available on the Web via the following URL:

Please refer to *http://www.ibm.com/software/mqseries* for details of these restrictions.

Who should read this book

This book is intended for anyone who wants to write Java and C based WebSphere MQ Everyplace programs to exchange secure messages within WebSphere MQ Everyplace systems, and between WebSphere MQ Everyplace systems and other members of the WebSphere MQ family of messaging and queueing products.

For information on the availability of development kits for environments other than Java and C, see the WebSphere MQ Web site at http://www.ibm.com/software/mqseries/

Prerequisite knowledge

This book assumes that the reader has a working knowledge of either Java or C programming.

An initial understanding of the concepts of secure messaging is an advantage. If you do not have this understanding, you may find it useful to read the following WebSphere MQ books:

- *WebSphere MQ An Introduction to Messaging and Queuing*
- *WebSphere MQ for Windows NT® V5R1 Quick Beginnings*, or the WebSphere MQ Quick Beginnings book that is relevant to the operating system that you are using.

These books are available in softcopy form from the Book section of the online WebSphere MQ library. The library can be reached from the WebSphere MQ Web site, URL address *http://www.ibm.com/software/mqseries/library/*

Terms used in this book

The following terms are used throughout this book:

WebSphere MQ family
refers to the following WebSphere MQ products:

- **WebSphere MQ Workflow** simplifies integration across the whole enterprise by automating business processes involving people and applications.
- **WebSphere MQ Integrator** is powerful message-brokering software that provides real-time, intelligent rules-based message routing, and content transformation and formatting.
- **WebSphere MQ Messaging** provides any-to-any connectivity from desktop to mainframe, through business quality messaging, supporting over 35 platforms.

WebSphere MQ Messaging
refers to the following messaging product groups:

- **Distributed messaging:** WebSphere MQ for Windows NT, AIX®, AS/400®, HP-UX, Sun Solaris, and other platforms
- **Host messaging:** WebSphere MQ for OS/390®
- **Workstation messaging:** WebSphere MQ for Windows
- **Pervasive messaging:** WebSphere MQ Everyplace

WebSphere MQ
refers to the following three WebSphere MQ Messaging product groups:

- Distributed messaging
- Host messaging
- Workstation messaging

WebSphere MQ Everyplace
Refers to the fourth WebSphere MQ Messaging product group, pervasive messaging.

Device platform
A small computer that is capable of running WebSphere MQ Everyplace only as a client

Server platform
A computer of any size that is capable of running WebSphere MQ Everyplace as a server or client

Gateway
A computer of any size running WebSphere MQ Everyplace programs that include the WebSphere MQ bridge function

Migration notes

This section contains information that you may need to consider when migrating from one version or release of WebSphere MQ Everyplace to a higher version or release.

General migration issues

The following lists the differences and similarities between the CBindings and the Native C implementations of WebSphere MQ Everyplace. Note that this is a complete Native C implementation of WebSphere MQ Everyplace rather than wrapping Java APIs. As a result of this, some changes have been required. However, it is still WebSphere MQ Everyplace, so concepts of assured message delivery, and so on, still apply.

What is the same?

- It is still a handle based API, with the same basic types, for example MQEINT32.
- APIs still have the same form.
- Errors are still handled in the same mannger.

What is different?

- Numeric Error codes have changed, some operations return new return or reason codes for situations.
- It is pirmilary a device QueueManager, therefore server style functionality does not apply.
- A limited set of adapters, cryptors, and compressors is available.

Not supported

The native C implementation is targeted as a device queuemanager.

WebSphere MQ bridge
MQeMQMessages can be constucted. Refer to the MQ_message example in the WebSphere MQ Everyplace C Programming Reference.

RunList
Not supported.

Adapters
Only supports the TcpIpHttpAdapter.

Cryptors
Only supports the RC4 cryptor.

Compressors
Only supports the RLE compressor.

Channel listener and manager
Not supported.

Migration warnings

- When compiliing against the bindings, a set of messages is dislayed. These are next to APIs, and so on, that have been changed. Details are given in the following section on Cross API warnings.
- LocalSecure localsecure:1 read attribute hdnl - Attribute Hanld change localsercure:2 write attribute hdnl

Cross API changes

The Bindings use MQeAttributeHndl as a base type for many differnet types of attributes. These types have been more tightly specified, for example getMessage calls take MQeFieldsAttrHndl.

The Bindings used a wrapper around the Java Enumeration classes. This has been replaced with a native MQeVector. The major change here is in terms of memory management. This is detailed in the Programming Reference/Examples/HTML reference guides.

Note: Specific sections of the API are referenced in the following sections. Note that the main headerfile MQe_API.h should always be included rather than specific API header files.

mqeQueueManager APIs

The following warnings are applicable to :

QM_01
QueueManager new parameters change. NAtive C requires the parameters, queue manager and registry, to be supplied in the new function rather than activate.

QM_02
QueueManager activate **- >** mqeQueueManager_start(). no longer requires additional params, see QM_01.

QM_03
No message listener currnetly in native C.

QM_04
Browse Messages attribute handle type is now MQeFieldsAttrHndl. Messages are now returned in an MQeVector.

QM_05
As QM_04 but for browseAndLock.

QM_06
mqeQueueManager_close is replaced with mqeQueueManager_stop.

QM_07
mqeQueueManager_getMessage attribute hndl.

QM_08
mqeQueueManager_getReference API has changed to mqeQueueManager_getCurrentQueueManager.

QM_09
mqeQueueManager_putMessage attribute hndl.

QM_10
Message Listener not currently supprted in native C.

QM_11
mqeQueueManager_waitForMessage not currently supported in native C.

QM_12
mqeQueueManager_setGlobalHashtable not supported in native C

Fields API

Fields_01
FieldEnumeraion deprecated, replace with MQeVector.

Fields_02
mqeFields_getAttribute.

Fields_03
Get and Put array length not supported in native C.

Fields_04
mqeFields_setAttribute attribute handle.

Fields_05
mqeFields_putDoubles() is deprecated. Use mqeFields_putArrayOfDouble() instead.

Fields_06
mqeFields_putFloats is deprecated. Use mqeFields_putArrayOfFloat() instead.

Fields_07
mqeFields_putInt32s is deprecated. Use mqeFields_putArrayOfInt32() instead.

Fields_08
mqeFields_putInt64s is deprecated. Use mqeFields_putArrayOfInt64() instead.

Fields_09
mqeFields_putInt16s is deprecated. Use mqeFields_putArrayOfInt16() instead.

Fields_10
mqeFields_putBytes is deprecated. Use mqeFields_putArrayOfByte() instead.

Fields_11
mqeFields_getDoubles() is deprecated. Use mqeFields_getArrayOfDouble() instead.

Fields_12
mqeFields_getFloats() is deprecated. Use mqeFields_getArrayOfFloat() instead.

Fields_13
mqeFields_getDoubles() is deprecated. Use mqeFields_getArrayOfInt32() instead.

Fields_14
mqeFields_getInt64s() is deprecated. Use mqeFields_getArrayOfInt64() instead.

Fields_15
mqeFields_getInt16s() is deprecated. Use mqeFields_getArrayOfInt16() instead.

Fields_16
mqeFields_getBytes() is deprecated. Use mqeFields_getArrayOfByte() instead.

Constants

WebSphere MQ Everyplace still supplies the full range of constants. New constants are thelist of constants to construct MQeMqMsgObjects from first principles. Specifiy MQeM.

- MQe_Adapter_Constants.h
- MQe_Admin_Constants.h
- MQe_Attribute_Constants.h

Administration messages

The native C implementation provides a fields implemention. The various subtypes though of fields, such as Administration messages are not supplied. A number of helper methods are provided to help construct the usual WebSphere MQ Everyplace Messages.

Various other subclasses of message can be constructed from a basic fields object, the required constants for the field labels, and the mqeFIelds_setClassName method. This method sets a field indicated what class this must be when reinstantiated in a Java queue manager.

Configuration and administration

`MQe_QueueManagerConfigure.h` is not applicable to the nativeC codebase. A new API defined by `MQe_Administrator.h` is provided. This allows configuration of the queue manager.

Deprecated classes

A number of classes have been removed from the product. We reccomend that you update any applications written to make use of the classes listed below to use the equivalent function provided in WebSphere MQ Everyplace version 2.0. To enable existing applications to be run during this migration, WebSphere MQ Everyplace provides the `MQeDeprecated.jar` jar file.

The `MQeDeprecated.jar file` contains the following classes:

- MQeMQBridge.class
- MQeChannelListener.class
- MQeChannelListenerTimer.class
- MQeChannelManager.class
- MQeTraceInterface.class

For more details on replacements for the above classes, refer to the the listing for each class in the WebSphere MQ Everyplace Java Programming Reference.

Migrating from version 1.2.7 to version 2.0

Changes to MQeFields

In order to comply with Java 2 Platform Micro Edition's (J2ME) Connected Limited Device Configuration(CLDC)/Mobile Information Device Protocol (MIDP) specification several methods have been modified or removed from MQeFields:

- The explicit use of the floating point types, float and double, have had to be removed, for example you putFloat(″Val1″, -1.234). Under java platforms that enable the use of float/double this functionality can be mimicked by explicity converting the data into the equivalent int or long using the base types Java Object convert method i.e. the above method is replaced with putFloatAsInt(″Val1″,Float.floatToIntBits(-1.234)).

  Note: Version 1 applications can retrieve these values as normal.
- Methods dumpToFile/restoreFromFile have been removed. Applications that used these functions now have to dump the MQeFields object and write the byte array to the specified file.
- Xor′ing of dumped data has also been removed due to changes made in the ′C′ code base.

Peer channels

Peer channels have been removed from the WebSphere MQ Everyplace version 2.0 codebase, because they caused problems with message routing and assured delivery.

Chapter 1. Introduction

This book describes how to customize the basic functions of WebSphere MQ Everyplace. It describes WebSphere MQ Everyplace messages, queues, and queue managers, detailing their functions, how to create links with other messaging software, and how to implement WebSphere MQ Everyplace security. It also provides C and Java programming examples for most sections.

As described in the WebSphere MQ Everyplace Introduction, WebSphere MQ Everyplace code can run on a large range of platforms including pervasive and mobile devices.

Chapter 2. ″HelloWorld″ messaging

This section describes how to create a basic application using the WebSphere MQ Everyplace Java and C APIs. It contains information on designing, developing, deploying, and running the application under the following main headings:

- Java codebase
- C codebase

Java codebase

This section describes how to develop and run a basic ″HelloWorld″ application in the Java codebase.

- Developing the Java ″HelloWorld″ application
- Running the Java ″HelloWorld″ application

Developing the Java ″HelloWorld″ application

The following code is in the `examples.helloworld.Run` class in its complete state. Solutions using WebSphere MQ Everyplace classes are often separated into several separate tasks:

- Installation of the solution
- Configuration of the queue manager, leaving the configuration information on the local hard disk
- Use of the queue manager
- Removal of the queue manager
- Un-install of the solution

Before reading the information in this chapter, you need to configure a queue manager. The `examples.helloworld.Configure` program demonstrates the configuration of the queue manager. The `examples.helloworld.Unconfigure program` demonstrates the removal of the queue manager. This section of the documentation describes how to use the queue manager.

Overview of the examples.helloworld.run program

The main method controls the flow of the hello world application. From this code, you can see that the queue manager is started, a message is put to a queue, a message is got from a queue, and the queue manager is stopped.

Trace information can be redirected to the standard output stream if the `MQE_TRACE_ON` symbolic constant has its’ value changed to ’`true`’.

```
public static void main(String[] args) {
        try {
            Run me = new Run();

            if (MQE_TRACE_ON) {
               me.traceOn();
            }
            me.start();
            me.put();
            me.get();
            me.stop();
             if (MQE_TRACE_ON) {
                me.traceOff();
            }
        } catch (Exception error) {
            System.err.println("Error: " + error.toString());
            error.printStackTrace();
        }
    }
```

Stage 1: Start the queue manager

The `examples.helloworld.Configure` program creates an image of the `HelloWorldQM` queue manager on disk.

Before a queue manager can be used, it must be instantiated in memory, and started. The start method in the example program does this.

```
 public void start() throws Exception {

      System.out.println("Starting the queue manager.");

         String queueManagerName = "HelloWorldQM";
         String baseDirectoryName =
       "./QueueManagers/" + queueManagerName;

         // Create all the configuration
    information needed to construct the
         // queue manager in memory.
         MQeFields config = new MQeFields();

       // Construct the queue manager section parameters.
         MQeFields queueManagerSection = new MQeFields();

         queueManagerSection.putAscii(MQeQueueManager.Name,
             queueManagerName);
         config.putFields(MQeQueueManager.QueueManager,
         queueManagerSection);

        // Construct the registry section parameters.
         // In this examples, we use a public registry.
         MQeFields registrySection = new MQeFields();

       registrySection.putAscii(MQeRegistry.Adapter,
      "com.ibm.mqe.adapters.MQeDiskFieldsAdapter");
         registrySection.putAscii(MQeRegistry.DirName,
          baseDirectoryName + "/Registry");

         config.putFields("Registry", registrySection);

         System.out.println("Starting the queue manager");
         myQueueManager = new MQeQueueManager();
         myQueueManager.activate(config);
         System.out.println("Queue manager started.");
     }
```

To start the queue manager, at a minimum you must know its name, location, and the adapter which should be used to read the queue manager’s configuration information from its registry.

Activating the queue manager causes the configuration data from the disk to be read using the disk fields adapter, and the queue manager is then started and running, available for use.

Stage 2: Create a message and put to a local queue

The following code constructs a message, adds a unicode field with a value of ″Hello World!″ and the message is then put to the `SYSTEM.DEFAULT.LOCAL.QUEUE` on the local `HelloWorldQM` queue manager.

```
 public void put() throws Exception {
        System.out.println("Putting the test message");
        MQeMsgObject msg = new MQeMsgObject();

   // Add my hello world text to the message.
        msg.putUnicode("myFieldName" , "Hello World!");

        myQueueManager.putMessage(queueManagerName,
    MQe.System_Default_Queue_Name, msg, null, 0L);
        System.out.println("Put the test message");
    }
```

Stage 3: Get message from a local queue

The following code gets the ″top″ message from the local queue, `SYSTEM.DEFAULT.LOCAL.QUEUE`, checks that a message with the field `myFieldName` was obtained, and displays the text held in the unicode field.

Stage 4: Shutdown

This section describes how to stop a queue manager and delete the definition of the queue manager.

Stopping the queue manager: You can stop the queue manager using a controlled shutdown.

```
  public void stop() throws Exception {
        System.out.println("Stopping the queue manager.");
        myQueueManager.closeQuiesce(QUIESCE_TIME);
        myQueueManager = null;
        System.out.println("Queue manager stopped.");
    }
```

Deleting the definition of the queue manager from the disk: You can use the `examples.helloworld.Unconfigure` program to remove the queue manager from disk.

Running the Java ″HelloWorld″ application

From a command prompt, set up your classpath to refer to the WebSphere MQ Everyplace class files. These are available in the Java directory, in which you installed the Websphere MQ Everyplace product.

Ensure that your shell has the ability to create and modify the `./QueueManagers directory` on your system. If it does not have this ability, you should change the source of the `examples.helloworld programs`, such that they refer to an accessible directory, and re-compile the java code.

Invoke the Configure program to create the queue manager. The syntax will depend on the Java Virtual Machine (JVM) you use. The IBM JVM is invoked using the ″java″ command, for example `java examples.helloworld.Configure`. This creates the queue manager on disk.

Run the `java examples.helloworld.Run` hello world program. This puts a message to a local queue, gets the message back and displays part of it.

You can now destroy the queue manager on the disk using `java examples.helloworld.Unconfigure`.

Refer to Chapter 3, “Running Applications”, on page 11, to understand how to set up an environment, which runs WebSphere MQ Everyplace applications.

C codebase

This section describes how to design, develop, deploy and run a ″HelloWorld″ application in the C codebase, under the following headings:

- Designing the C ″HelloWorld″ application
- Developing the C ″HelloWorld″ application
- Deploying the C ″HelloWorld″ application
- Running the C ″HelloWorld″ application

Designing the C ″HelloWorld″ application

This application aims to create and use a single queue manager with a local queue. It involves putting a message to the local queue and then removing it.

You can create queue managers for use by one program. Once this program has completed, you can run a second program that reinstates the previous queue manager configuration.

Typically, configuring new entities is a separate process from their actual use. Once configured, administering these entities also requires a different process than using them. This section concentrates on usage rather than administration. The WebSphere MQ Everyplace Configuration Guide contains information on configuring and administering resources.

Assuming that the queue manager entity has already been configured, the HelloWorld application has the following flow for both the C and Java codebases:

1. **Start the queue manager**
   This starts the queue manager based on information already created
2. **Create a message**
   Creates a structure that you can use to send a message from one queue manager to another
3. **Put to a local queue**
   Puts the message on the local queue
4. **Get from a local queue**
   Retrieves the message from the local queue and checks that the message is valid
5. **Shutdown**
   Clears and stops the queue manager

Note: The C codebase does not have an equivalent of the Java Garbage Collection function. Therefore, clearing the queue manager features more strongly in C.

Developing the C ″HelloWorld″ application

This section covers the high level coding required for the ″HelloWorld″ application.

C Development

The following code is in the example HelloWorld_Runtime.c in its complete state. The example contains code to handle the specifics of running a program on a PocketPC, which mainly involves writing to a file to cope with the lack of command line options. Use the display function to write to a file, as shown in the examples contained in the following sections.

Preparation: You need to include just one header file to access the APIs.

Note: You must include the NATIVE definition to indicate that this is not the CBindings. You must also define the MQE_PLATFORM upon which you intend to run the application.

```
#define  NATIVE
#define MQE_PLATFORM = PLATFORM_WINCE
#include<published/MQe_API.h>
```

All of the code, including variable declarations, is inside the main method. You require structures for error checking. The **MQeExceptBlock** structure is passed into all functions to get the error information back. In addition, all functions return a code indicating success or failure, which is cached in a local variable:

```
/* ... Local return flag */
 MQERETURN           rc;
 MQeExceptBlock      exceptBlock;
```

You must create a number of strings, for example for the queue manager name:

```
MQeStringHndl     hLocalQMName;

    ...

 if ( MQERETURN_OK == rc ) {
  rc = mqeString_newUtf8(&exceptBlock,
          &hLocalQMName,
          "LocalQM");
}
```

The first API call made is session initialize:

```
/* ... Initalize the session */
rc = mqeSession_initialize(&exceptBlock);
```

Stage 1: Start the queue manager: This process involves two steps:

1. Create the queue manager item.
2. Start the queue manager.

Creating the queue manager requires two sets of parameters, one set for the queue manager and one for the registry. Both sets of parameters are initialized. The *queue store* and the registry require directories.

Note: All calls require a pointer to ExceptBlock and a pointer to the queue manager handle.

```
 if (MQERETURN_OK == rc) {

MQeQueueManagerParms qmParams  = QMGR_INIT_VAL;
MQeRegistryParms     regParams = REGISTRY_INIT_VAL;
qmParams.hQueueStore           = hQueueStore;
qmParams.opFlags               = QMGR_Q_STORE_OP;

/* ... create the registry parameters -
  minimum that are required */
regParams.hBaseLocationName     =  hRegistryDir;
display("Loading Queue Manager from registry \n");
rc = mqeQueueManager_new( &exceptBlock,
                       &hQueueManager,
                          hLocalQMName,
                         &qmParams,
                         &regParams);
}
```

You can now start the queue manager and carry out messaging operations:

```
 /* Start the queue manager  */

if ( MQERETURN_OK == rc ) {
  display("Starting the Queue Manager\n");
  rc = mqeQueueManager_start(hQueueManager,
          &exceptBlock);
}
```

Stage 2: Create a message: To create a message, firstly create a new fields object. The following example adds a single field. Note that the field label strings are passed in:

```
MQeFieldsHndl hMsg;

display("Creating a new message\n");
rc = mqeFields_new(&exceptBlock,&hMsg);
if ( MQERETURN_OK == rc ) {
 rc = mqeFields_putInt32(hMsg,&exceptBlk,
          hFieldLabel,42);
}
```

Stage 3: Put to a local queue: Once you have created the message, you can put it to a local queue using the *putMessage* function. Note that the queue and queue manager names are passed in. NULL and 0 are passed in for the security and assured delivery parameters, as they are not required in this example. Once the message has been put, you can free the MQeFields object:

```
if ( MQERETURN_OK == rc ) {
      display("Putting a message \n");
      rc = mqeQueueManager_putMessage(hQueueManager,
                                     &exceptBlock,
                                      hLocalQMName,
                                      hLocalQueueName,
                                     hMsg,
                                      NULL,
                                      0);

     (void) mqeFields_free(hMsg,NULL);
   }
```

Stage 4: Get from a local queue: Once the message has been put to a queue, you can retrieve and check it. Similar options are passed to the getMessage function. The difference is that a pointer to a fields handle is passed in. A new Fields object is created, removing the message from the queue:

```
MQeFieldsHndl hReturnedMessage;
display("Getting the message back \n");

rc = mqeQueueManager_getMessage(hQueueManager,
                                &exceptBlock,
             &hReturnedMessage,
            hLocalQMName,
            hLocalQueueName,
                                 NULL,
                                 NULL,
                                  0);
 }
```

Once the message has been obtained, you can check it for the value that was entered. Obtain this by using the getInt32 function. If the result is valid, you can print it out:

```
if (MQERETURN_OK == rc) {
 MQEINT32 answer;
  rc = mqeFields_getInt32(hReturnedMessage,
                          &exceptBlock,
          &answer,
                            hFieldLabel);

if (MQERETURN_OK == rc) {
 display("Answer is %d\n",answer);
}
else {
  display(  "\n\n       %s  (0x%X)     %s (0x%X)\n",
    mapReturnCodeName(EC(&exceptBlock))  ,
    EC(&exceptBlock),
    mapReasonCodeName(ERC(&exceptBlock)),
    ERC(&exceptBlock)  );
 }

}
```

Stage 5: Shutdown: Following the removal of the message from the queue, you can stop and free the queue manager. You can also free the strings that were created. Finally, terminate the session:

```
(void)mqeQueueManager_stop(hQueueManager,&exceptBlock);
(void)mqeQueueManager_free(hQueueManager,&exceptBlock);

/* Lets do some clean up */
(void)mqeString_free(hFieldLabel,&exceptBlock);
(void)mqeString_free(hLocalQMName,&exceptBlock);
(void)mqeString_free(hLocalQueueName,&exceptBlock);
(void)mqeString_free(hQueueStore,&exceptBlock);
(void)mqeString_free(hRegistryDir,&exceptBlock);


(void)mqeSession_terminate(&exceptBlock);
```

C Compilation

To simplify the process of compiling, the examples directory includes a makefile. This is the makefile exported from eMbedded Visual C (EVC). A batchfile runs this makefile. This batch file will setup the paths to the EVC directories, along with the paths to the WebSphere MQ Everyplace installation. You may need to edit the batch file, depending on how you want to install WebSphere MQ Everyplace.

Running the batch file will compile the example. By default, the batch file compiles for Debug PocketPC 2000 (either Emulator or ARM processor).

Deploying the C ″HelloWorld″ application

In order to deploy the ″HelloWorld″ application, you need to create a queue manager. There are various ways to do this, which are covered in the WebSphere MQ Everyplace Configuration Guide. In this case, the HelloWorld_Admin program is used. Run this as described below.

C deployment

The next chapter, Running applications, covers C deployment in detail. The basics, applicable to both the emulator and an actual device, are as follows:

1. Copy across all the DLLs to the root of the device. Take these from either the arm or x86 emulator directories.
2. Build the example code using the supplied makefile.

   Note: You need to compile the HelloWorld_Admin.c and HelloWorld_Runtime.c files.
3. Copy across these binaries to the device or emulator that is running PocketPC or Emulator.

Running the C ″HelloWorld″ application

This section describes how to run the ″HelloWorld″ application in Java and C.

PocketPC or emulator

This example involves two steps:

1. Create the queue manager. To do this, run the HelloWorld_Admin program.Running this creates the persistent disk representation of the QueueManager.
2. Run the HelloWorld_Runtime program. This starts a QueueManager based upon the established registry. To check the program has worked correctly, look at the log file that has been generated. By default, this is in the root of the device.

Chapter 3. Running Applications

This section introduces Version 2.0.0.5 of the WebSphere MQ Everyplace Development Kit. The development kit is a development environment for writing messaging and queuing applications based on Java 1.1 and C.

Note: For information on the availability of development kits for environments, other than Java and C, and on other supported platforms, see the WebSphere MQ Web site at `http://www.ibm.com/software/ts/mqseries/`

The code portion of the Java development kit comes in two sections:

Base WebSphere MQ Everyplace classes
A set of Java classes that provide all the necessary function to build messaging and queuing applications.

Examples
Java source code and classes that demonstrate how to use many features of WebSphere MQ Everyplace. Some examples are supplied in Appendix A, "WebSphere MQ Everyplace Java programming examples", on page 143 of this book.

The code portion of the C development kit also comes in two sections:

Base WebSphere MQ Everyplace functions
C code that provides all the necessary function to build messaging and queuing applications.

Examples
C source code that demonstrates how to use many features of WebSphere MQ Everyplace.

Development environment

This section describes the tools you will need to develop programs using the WebSphere MQ Everyplace Development Kit. It covers the following environments:

- Java development
- C development
- J2ME environment

Java development

To develop programs in Java using the WebSphere MQ Everyplace development kit, you must set up the Java environment as follows:

- Set the *CLASSPATH* so that the Java Development Kit (JDK) can locate the WebSphere MQ Everyplace classes.

  Windows

  In a Windows® environment, using a standard JDK, you can use the following:

  ```
  Set CLASSPATH=<MQeInstallDir>\Java;%CLASSPATH%
  ```

  UNIX®

  In a UNIX environment you can use the following:

```
CLASSPATH=<MQeInstallDir>/Java:$CLASSPATH
export CLASSPATH
```

- If you are developing code that uses or extends the WebSphere MQ–bridge, the WebSphere MQ Classes for Java must be installed and made available to the JDK.

You can use many different Java development environments and Java runtime environments with WebSphere MQ Everyplace. The system configuration for both development and runtime is dependent on the environment used. WebSphere MQ Everyplace includes a file that shows how to set up a development environment for different Java development kits. On Windows systems this is a batch file called JavaEnv.bat, for UNIX systems it is a shell script called JavaEnv. To use this file, copy the file and modify the copy to match the environment of the machine that you want to use it on.

A set of batch files and shell scripts that run some of the WebSphere MQ Everyplace examples use the environment file described above, and, if you wish to use the example batch files, you must modify the environment file as follows:

- Set the *JDK* environment variable to the base directory of the JDK.
- Set the *JavaCmd* environment variable to the command used to run Java applications.
- If WebSphere MQ Classes for Java is installed, set the *MQDIR* environment variable to the base directory of the WebSphere MQ Classes for Java.

Note: Customized versions of JavaEnv.bat or JavaEnv may be overwritten if you reinstall WebSphere MQ Everyplace.

When you invoke JavaEnv.bat on Windows you must pass a parameter that determines the type of Java development kit to use.

Possible values are:

Note: These parameters are case sensitive and must be entered exactly as shown.

Sun	- Sun
JB	- Borland JBuilder
MS	- Microsoft®
IBM	- IBM

If you do not pass a parameter, the default is IBM.

The JavaEnv shell script on UNIX does not use a corresponding parameter.

On Windows, by default, you must run JavaEnv.bat from the <MQeInstallDir>\java\demo\Windows directory. On UNIX, by default, you must run JavaEnv from the <MQeInstallDir>/Java/demo/UNIX directory. Both files can be modified to allow then to be run from other directories or to use other Java development kits.

C development

To develop programs in C, using the WebSphere MQ Everyplace Development Kit, you need the following tools:

Microsoft eMbedded Visual C++ (EVC) Version 3.0.
This is included in Microsoft eMbedded Visual Tools 3.0, which is available as a free download from the Microsoft web page:
http://msdn.microsoft.com/mobile/
You must use version 3.0 as version 4.0 does not support PocketPC.

An SDK for your chosen platform
Microsoft eMbedded Visual Tools 3.0 includes an SDK for PocketPC 2000. You can also download an SDK for PocketPC 2002 from Microsoft:
http://msdn.microsoft.com/mobile/

Compilation information

The two main subdivisions of the native C codebase are code for PocketPC 2000 and code for PocketPC 2002. For both, there are binary files for the actual device and also for the emulator. Binary files are compiled for the ARM processors.

Binary files

The root of the binarie, as well as documentation and examples, is the C directory, found in the main installation directory. This contains directories for the examples, documentation, and separate directories for PocketPC 2000 and 2002.

PocketPC 2002

- DLL files in C\PocketPC 2002\arm\bin
- LIB files in C\PocketPC 2002\arm\lib

PocketPC 2000

- DLL files in C\PocketPC 2000\arm\bin
- LIB files in C\PocketPC 2000\arm\lib

Using eMbedded Visual C++

You can compile applications using the EVC Integrated Development Environment (IDE), or optionally, from the command line. However, you must consider the following:

- Set the appropriate "Active WCE Configuration", using the WCE Configuration toolbar. To do this, under **Target Operating System** select either PocketPC or PocketPC 2002. Also, under **Target Processor** , select one of the following:
  - Win32 (WCE x86em) Debug
  - Win32 (WCE x86em) Release
  - Win32 (WCE ARM) Debug
  - Win32 (WCE ARM) Release

  Note: Some of the **Target Processor** or **Target Operating System** options may not be available, depending on which SDKs you have installed.
- Include the header files for the native C codebase. These are shared between the two versions of PocketPC and by the C Bindings. The header file location is in the installation directory under `include`. If you include the root header file, MQe_API.h, you include all the functions that you may require. As header files are shared, you need to define which version of the codebase you are using, as shown in the following example:

```
#define NATIVE
#define MQE_PLATFORM  PLATFORM_WINCE

/*Alternatively, we recommened that you add this to the Preprocessor Definintions
in the Project Settings Dialog.  Add the following to the start
```

```
of the list*/
NATIVE,MQE_PLATFORM=PLATFORM_WINCE

#include <published\MQe_API.h>
```

- Include an entry for the top level WebSphere MQ Everyplace include directory in "Additional include directories". This varies according to where you install the product.
- Insert the following .lib file names in the "Project Settings" dialog, under **Link —> Input** :
  - HMQ_nativeAPI.lib
  - HMQ_nativeCnst.lib

Note: There are variations of these files for each supported release, for example one for PocketPC 2000 ARM, one for PocketPC 2000 x86em, and so on. To ensure that you use the correct verion, qualify the filename fully for each target build.

We recommend that you develop applications using the PocketPC or PocketPC2002 emulator as this typically provides a faster compilation and debug environment. However, current emulators are API emulators, meaning that they do not emulate ARM hardware. They emulate PocketPC API calls, but the code is still x86, that is running in an x86 virtual machine in the PocketPC 2002 emulator case. Therefore, we recommend that you regularly test the application on the real target device, as many problems such as byte-alignment only becomes apparent on the real device.

Note: WebSphere MQ Everyplace emulator binaries are provided only for development purposes and are not suitable for deployment into a production environment.

Threading

The native codebase is designed to be re-entrant. The actual codebase does not use threads, but this does not preclude the use of multiple threads in the application. For example, you can create an application thread to repeatedly call `mqeQueueManager_triggerTransmission()`. If you want to use multiple threads, you do not need to call any specific APIs.

Although it is not a requirement. we recommend that you have an exception block per thread. If you use one exception block shared across threads, an exception block for a thread that fails can be overwritten by the exception block for a thread that succeeds.

Note: You must call `mqeSession_initialize` or `mqeSessuion_terminate`once only, before any threads use a WebSphere MQ Everyplace API call. To ensure this, call it in the main thread before any application threads are created. For example, **do not** use the following:

```
mqeSession_initialize();
mqeSession_initialize();
mqeSession_terminate();
mqeSession_terminate();
```

Calling convention

The calling convention for all of the APIs has been explicitly set at `_cdecl`. However, you can use

a different default calling convention in your application.

Handles and items

An application needs a mechanism for accessing WebSphere MQ Everyplace items such as the queue manager, fields, strings, and so on. Handles use WebSphere MQ Everyplace items. The handle points to an area of memory used to store the specific information for that instance of the item. Type information is held for each item. Therefore, you must take care to initialize the handle correctly.

To use a handle, you must initialize it. You can do this by calling the new function of the associated item to be used. For example, to create an MQeString, you must first call the `mqeString_new()` function and pass a pointer to `MQeStringHndl` to that function. The `mqeString_new()` function allocates memory for the internal structure and sets the required default values by MQeString. Once completed successfully, the function returns the handle, which can now be used in subsequent calls to MQeString functions.

Once an item has been finished with, it is important to call the `free()` function of the item with which the handle is associated. The `free()` functions release all the systems resources used by that item. Seting the handle to `NULL` introduces a memory leak to the application and the system may run out of resources. To avoid this, set the handle to `NULL` after it has been freed.

Note: We recomment that you do not attempt to free a handle more than once, as this can cause unprecedented results.

You must use handles only with their associated items. You must also initialize and free them in the correct manner. The only instances where the application is not responsible for initializing the handle is when a pointer to a handle is passed as an input parameter to a WebSphere MQ Everyplace API. In such instances, a fully initialized handle is returned to the application without the user having to invoke the relevant new() function. An example of this is `mqeQueueManager_BrowseMessages()`, which has a pointer to an `MQeVectorHndl` as an input parameter. However, in instances like this, the application is still responsible for freeing the handle.

WebSphere MQ Everyplace memory functions

WebSphere MQ Everyplace provides the following functions for memory management:

- `mqeMemory_allocate`
- `mqeMemory_free`
- `mqeMemory_reallocate`

These functions use the same memory management routines that are used within the WebSphere MQ Everyplace codebase. These are available for use by application programs. An application can generally use its own choice of memory management. However, some API calls, for example `mqeAdministrator_QueueManager_inquire`, need to return blocks of memory containing information. In this case, the memory must be freed using the `mqeMemory_free function`.

An additional advantage of using the mqeMemory functions is that their use gets traced along with mqe processing. However, never mix the memory allocation calls. For example, do not free memory allocation with `mqeMemory_allocate` with the C runtime `free()` call, as the application can become unstable.

MQeString

The MQeString class contains user defined and system strings. It is an abstraction of character strings used throughout the C API where a string is required. MQeString allows you to create a string in a number of formats, such as arrays containing Unicode code points, with each code point stored in a 1, 2, or 4 byte memory space, and UTF-8. The current implementation of MQeString supports external formats only.

Note: Although they are passed using an MQeString, some API calls require the actual string to lie within the valid ASCII range.

Constant Strings

A number of constant strings are provided. These are defined in the following header files:

- MQe_Admin_Constants.h
- MQe_Adapter_Constants.h
- MQe_Attribute_Constants.h
- MQe_Connection_Constants.h
- MQe_MQe_Constants.h
- MQe_MQeMessage_Constants.h
- MQe_Queue_Constants.h
- MQe_Registry_Constants.h

Constructor

```
MQERETURN osaMQeString_new(MQeExceptBlock* pExceptBlock,
                              MQEVOID*        pInputBuffer,
                              MQETYPEOFSTRING type,
                              MQeStringHndl * phNewString
                             );
```

This function creates a new MQeString object from a buffer containing character data. The data can be in a number of supported formats including, null terminated single byte character arrays (i.e. normal C char* strings), null terminated double-byte Unicode character arrays, null terminated quad-byte Unicode character arrays, and null terminated UTF-8 arrays. The type parameter tells the function what format the input buffer is in.

Destructor

```
MQERETURN osaMQeString_delete(MQeExceptBlock* pExceptBlock,
                               MQeString_*     pString
                              );
```

This function destroys an MQeString object that was created using **osaMQeString_new**, or **MQeString_duplicate**, or **MQeString_getMQeSubstring**

Getter

```
 MQERETURN osaMQeString_get(MQeExceptBlock*      pExceptBlock,
                       MQEVOID*            pOutputBuffer,
                       MQEINT32*           pBufferLength,
                       MQETYPEOFSTRING     requiredType,
                       MQECONST MQeStringHndl hString
                       );
```

This function populates a character buffer with the contents of an MQeString performing conversion wherever necessary. Only simple conversions are carried out. No codepage conversion is attempted. For example, if an SBCS string has been put into the string, then trying to get the data out as DBCS (Unicode) data works correctly. If the data was put in as DBCS however, and you try to get the data out as SBCS, this only works if the data does not have any values that cannot be represented with a single byte. When **get()** is used for SBCS, DBCS, or QBCS, each character is represented by its Unicode code point value.

```
MQERETURN osaMQeString_getSubstring(MQeExceptBlock* pExceptBlock,
                  MQEVOID*       pOutputBuffer,
                  MQEINT32*      pBufferLength,
                  MQETYPEOFSTRING requiredType,
                  MQECONST MQeStringHndl hString,
                  MQEINT32 from,
                  MQEINT32 to
                  );
```

This function is very similar to osaMQeString_get except that it only gets a substring (from **from** to **to** inclusive).

```
MQERETURN osaMQeString_getMQeSubstring(MQeExceptBlock* pExceptBlock,
                  MQeStringHndl *      phOutput,
                  MQECONST MQeStringHndl   hString,
                  MQEINT32 from,
                  MQEINT32 to
                  );
```

This function is very similar to osaMQeString_getSubstring except it returns its result as an MQeString.

```
MQERETURN osaMQeString_duplicate(MQeExceptBlock  * pExceptBlock,
                  MQeStringHndl * phNewString,
                  MQECONST MQeStringHndl   hString
                  );
```

This function duplicates an MQeString.

```
MQERETURN osaMQeString_codePointSize(MQeExceptBlock* pExceptBlock,
                  MQEINT32 * pSize,
                  MQECONST MQeStringHndl   hString
                  );
```

This function finds the memory size (in bytes) required for the largest character in the string.

```
MQERETURN osaMQeString_getCharLocation( MQeExceptBlock* pExceptBlock,
                  MQEINT32*      pOutIndex,
                  MQECONST MQeStringHndl    hString,
                  MQECHAR32      charToFind,
                  MQEINT32       startFrom,
                  MQEBOOL        searchForward
                  );
```

This function returns the location index (starting from 0) of the first appearance of a specified character, specified as its Unicode code point value. You can specify the starting point of your search and the direction of the search.

Tester

```
MQERETURN osaMQeString_isAsciiOnly(MQeExceptBlock*    pExceptBlock,
                       MQEBOOL*          pIsAsciiOnly,
                       MQECONST MQeString_* pString
                       );
```

This function determines whether the string contains any non-invariant ASCII characters.

```
MQERETURN osaMQeString_equalTo(MQeExceptBlock*   pExceptBlock,
                         MQEBOOL*          pIsEqual,
                         MQECONST MQeString_* pString,
                         MQECONST MQeString_* pEqualToString
                         );
```

This function determines whether two strings are equivalent.

```
MQERETURN osaMQeString_isNull(MQeExceptBlock  * pExceptBlock,
                         MQEBOOL * pIsNull,
                         MQECONST MQeStringHndl  hString
                         );
```

This function determines if a string is a null string. A a NULL handle is considered as a null string as well.

The Single Byte Character Set (SBCS) is the standard mode of operating with C on an ASCII code page. Java works in Unicode only and there may be platforms to support, that do not load an SBCS code page, for example in some countries languages are represented in DBCS. As it does not include the character pointer, the string item allows you to create strings on an ASCII machine without considering Unicode requirements. WebSphere MQ Everyplace carries out any necessary conversions. Use the UTF-8 representation of the string as this can cope with any character representation and does the conversion for you. Once created, an MQeString cannot be altered. However, a number of functions facilitate the use of the MQeString type. You can also create constant MQeStrings in a similar manner to using `#define NAME "mystring"`. Using MQeString ensures portability of the application.

J2ME environment

There are two distinct J2ME environments:

Connected Device Configuration (CDC) and Profile

An example is Foundation + Applications in the CDC environment, which can effectively be developed like a normal Java 2 Platform Standard Edition (J2SE) application. The only change required is modifying the bootclasspath option to point to the relevent CDC jar or zip class file.

Note: The 'bootclasspath' option may not be available on all JVM's

Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP)

Applications developed for MIDP can also be compiled using a normal J2SE JVM (again using the bootclasspath to point to the required Midp class library), but they normally have to be run within a Midp Emulator. Therefore, we recommend developing the application using one of the MIDP Toolkits available on the Web. WebSphere MQ Everyplace provides a MIDP jar that should be used within this environment. The `MQeMidpBase.jar` is in the <MQeInstallDir>\Java\Jars directory.

Windows security configuration

Java development

WebSphere MQ Everyplace provides a sample Windows NT authenticator, but the default WebSphere MQ Everyplace installation does not make all the changes necessary for this authenticator to execute. If you wish to use the authenticator you should complete the following configuration.

Note: The Windows NT authenticator is used by the MQe_Explorer that is shipped in SupportPac ES02.

1. The file JavaNT.dll, which interfaces between WebSphere MQ Everyplace and Windows security, must be placed in the search path or in the current directory. In a standard installation, this file is located in C:\Program Files\MQe\Java\demo\Windows\i86\NT. Put a copy of this file in the directory that contains your Windows .dll files (normally C:\WINNT\system32).

   Note: This makes the sample authenticator available to all WebSphere MQ Everyplace applications. If you only wish to make the authenticator available to the MQe_Explorer, put the copy of JavaNT.dll in the same directory as MQe_Explorer.exe.
2. Security permissions must be set correctly for the JavaNT.dll to be granted permission to access the Windows user/password database.

   On Windows 2000:

   a. From the Start button click on Programs, then Administrative Tools, then Local Security Policy
   b. In the Local Security Settings panel click on Local Policies in the left hand pane, then User Rights Assignment. In the right hand pane check that your current `user ID` is assigned all of the following privileges:
      - Act as part of the operating system
      - Log on as a service
      - Log on locally

   If all these privileges are not assigned to your ID, double click the relevant privilege and add your `user ID`.

   On Windows NT:

   a. From the Start button click on Programs, then Administrative Tools, then User Manager.
   b. In the Policies menu click on User Rights
   c. In the User Rights Policy dialogue, check the box Show Advanced User Rights. Check the following rights in turn:
      - Act as part of the operating system
      - Log on as a service
      - Log on locally

      Each right should be granted to the logged on `user ID`. If your ID, or a group to which your ID belongs, is not listed for any of these rights, click the Add button to add your ID to the Grant to list.

   When all the privileges have been set you must then logoff Windows and logon again to get these privilege enabled for the current session (it is not necessary to reboot the machine).

C development

WebSphere MQ Everyplace provides a sample Windows CE authenticator. If you want to use the authenticator, copy the WinCEAuthenticator.dll to your decive. Refer to Chapter 10, “Error and error handling”, on page 127for information on how to do this.

Deploying applications

This section dsescribes how to deploy WebSphere MQ Everyplace applications for Java and C.

Java deployment

When deploying WebSphere MQ Everyplace applications, you are recommended to pack the minimum set of classes required by the application into compressed jar files. This ensures that the application requires the minimum system resources. WebSphere MQ Everyplace provides the following examples of how the WebSphere MQ Everyplace classes can be packaged into .jar files. These examples are in the<MQeInstallDir>\Java\Jars directory of a standard WebSphere MQ Everyplace installation.

WebSphere MQ Everyplace ships the following .jar files

MQeBase.jar
This file contains basic information only. It does not contain any security information, compressors, cryptors, or authenticators. It contains all communication information, queue types, administration for all of the included resources. It also contains a client and server. With the MQeBase.jar, you can send WebSphere MQ Everyplace and WebSphere MQ Everyplace MQ messages. It supports the diskfields, memory, and reduceddiskfields adapters.

MQeBindings.jar
This file contains all C bindings specific information. You need this .jar file to use the C library bindings to control a Java WebSphere MQ Everyplace queue manager. It includes com.ibm.mqe.bindings, which you need to use the C library bindings.

MQeCore.jar
This contains mandatory classes.

MQeDeprecated.jar
This contains all of the deprecated class files that are no longer needed by a WebSphere MQ Everyplace application. These deprecated class files help you run applications written using a previous version of WebSphere MQ Everyplace, without making any changes.

MQeDiagnostics.jar
This file helps to diagnose problems with WebSphere MQ Everyplace classes. It contains tooling to search the class path to find out the level of each class found.

MQeExamples.jar
A packaging of all the WebSphere MQ Everyplace examples into one jar file. This includes all of the examples supplied with WebSphere MQ Everyplace, but excludes the deprecated classes.

MQeGateway.jar
This contains the classes that can be used on a server platform. It includes the bridge class, but excludes the deprecated classes.

MQeJMS.jar
This contains the classes that provide a subset of the JMS interface, suitable for use on smaller devices.

MQeMidp.jar
This is equivalent to MQeBasic.jar, but it is for use with J2ME.

MQeMigration.jar
This contains classes in the com.ibm.mqe.validation package.

MQeRetail.jar
This contains extra message stores with short filenames.

MQeSecurity.jar
A set of classes that can be used to extend both the MQeBasic.jar to allow both queue and message based security.

A new jar file, the MQeMidpBase.jar

To run WebSphere MQ Everyplace applications, you must set up the Java runtime environment to include the required WebSphere MQ Everyplace and application classes. Using a standard Java runtime environment (JRE), you must set the CLASSPATH to include any required `jar` files.

Example statements are:

Windows

```
Set CLASSPATH=<MQeInstallDir>\Jars\MQeDevice.jar;%CLASSPATH%
```

UNIX

```
CLASSPATH=<MQeInstallDir>/Java/Jars/MQeDevice.jar:$CLASSPATH
export CLASSPATH
```

C deployment

To deploy applications on the PocketPC 2000 or PocketPC 2002 devices, you need to copy the WebSphere MQ Everyplace DLLs to the device. Copy the DLLs to the Windows directory, the root directory, or the same directory that holds the application. The following tables show which DLLs you need for different WebSphere MQ Everyplace entities. You need the following DLLs for the local queuing base:

- HMQ_Core.dll
- HMQ_DiskAdapter.dll
- HMQ_HAL.dll
- HMQ_nativeAPI.dll
- HMQ_nativeOSA.dll
- HMQ_RegistryFileSession.dll
- HMQ_LocalQueue.dll

Along with the base DLLs you require the following DLLs depending on how you wish to configure your application:

Remote queuing
Add the HMQ_HttpAdapter.dll to the local queuing base DLLs.

Note: You can remove HMQ_LocalQueue.dll, if you do not want to support administration queues or local queueing.

Synchronous remote queue support
Add HMQ_SyncRemoteQueue.dll to the local queuing base DLLs.

Asynchronous remote queue support
Add HMQ_AsyncRemoteQueue.dll to the local queuing base DLLs.

Home server queue support
Add HMQ_HomeServerQueue.dll to the local queuing base DLLs.

Administration queue support
Add HMQ_AdminQueue.dll and HMQ_LocalQueue.dll to the local queuing base DLLs.

RLE compressor support
Add HMQ_RleCompressor.dll to the local queuing base DLLs.

RC4 crytpor support
Add HMQ_RC4Cryptor.dll to the local queuing base DLLs.

Support for included examples
Add BrokerDLL.dll to the local queuing base DLLs.

Post install test

This section describes how to run a set of examples that verify the successful installation of a WebSphere MQ Everyplace development kit.

Java

Once you have installed WebSphere MQ Everyplace you can use the following procedures to run a set of examples that determine whether the installation of the development kit was successful.

- Ensure that the Java environment is set up as described in “Development environment” on page 11. When running any of the Windows batch files described in this section, the first parameter of each is the name of the Java development kit to use. If you do not specify a name, the default is IBM.

  Note: The UNIX shell scripts do not have a corresponding parameter.
- Move to the correct directory:

  Windows
  Change to the <MQeInstallDir>\Java\demo\Windows directory.

  UNIX Change to the <MQeInstallDir>/Java/demo/UNIX directory.
- Create a queue manager as follows:

  Windows
  Run the batch file

  ```
  CreateExampleQM.bat <JDK>
  ```

  UNIX Run the shell script

  ```
  CreateExampleQM
  ```

  to create an example queue manager called `ExampleQM`.

Part of the creation process sets up directories to hold queue manager configuration information and queues. The example uses a directory called ExampleQM that is relative to the current directory. Within this directory are two other directories:

  - Registry - holds files that contain queue manager configuration data.
  - Queues - for each queue there is a subdirectory to hold the queue's messages. (The directory is not created until the queue is activated.)

- Run a simple application as follows:

  Once you have created a queue manager you can start it and use it in applications. You can use the batch file ExamplesMQeClientTest.bat or the shell script ExamplesMQeClientTest to run some of the simple application examples.

  The batch file runs examples.application.Example1 by default. This example puts a test message to queue manager `ExampleQM` and then gets the message from the same queue manager. If the two messages match, the application ran successfully.

  There are a set of applications in the examples.application package that demonstrate different features of WebSphere MQ Everyplace. You can run these examples as follows:

  Windows
  Pass parameters to the batch files:

  ```
  ExamplesMQeClientTest <JDK> <ExampleNo>
  ```

  UNIX Pass parameters to the shell scripts:

  ```
  ExamplesMQeClientTest <ExampleNo>
  ```

  where *`ExampleNo`* is the suffix of the example. This can range from 1 to 6.

- Delete a Queue manager.

  When a queue manager is no longer required you can delete it. To delete the example queue manager `ExampleQM`:

  Windows
  Run the batch file

  ```
  DeleteExampleQM.bat <JDK>
  ```

  UNIX Run the shell script

  ```
  DeleteExampleQM
  ```

  .

  Once you have deleted a queue manager you cannot start it.

  Note: The examples use relative directories for ease of set up. You are strongly recommended to use absolute directories for anything other than base development and demonstration. If the current directory is changed, and you are using relative directories, the queue manager can no longer locate its configuration information and queues.

C

Once you have installed WebSphere MQ Everyplace you can run the examples, from the WebSphere MQ Everyplace C Programming Reference, to determines whether the installation of the development kit was successful.

Chapter 4. Messaging

The WebSphere MQ Everyplace programming model uses several entities, for example messages, queues, and queue managers, that work together as a flexible toolkit. Each entity has a specific purpose and works together with other entities to provide solutions for message topologies.

Assuming you have read the WebSphere MQ Everyplace Introduction, this chapter introduces the concept of messaging under the following headings:
- MQeFields
- What are WebSphere MQ Everyplace messages?
- Message filters
- Message expiry
- Queue aliases

MQeFields

MQeFields is a container data structure widely used in WebSphere MQ Everyplace. You can put various types of data into the container. It is particularly useful for representing data that needs to be transported, such as messages. The following code creates an MQeFields structure:

Java code

```
/* create an MQeFields  object   */
MQeFields fields = new MQeFields( );
```

C code

```
MQeFieldsHndl hFields;
 rc = mqeFields_new(&exceptBlock, &hFields);
```

MQeFields contains a collection of orderless fields. Each field consists of a triplet of entry name, entry value, and entry value type. MQeFields forms the basis of all WebSphere MQ Everyplace messages.

Use the entity name to retrieve and update values. It is good practice to keep names short, because the names are included with the data when the MQeFields item is transmitted.

The name must:
- Be at least 1 character long
- Conform to the ASCII character set (characters with values 20 < value < 128)
- Exclude any of the characters { } [ ] # ( ) : ; , ′ ″ =
- Be unique within MQeFields

The following example shows how to store values in an MQeFields item:

Java code

```
/* Store integer values into a fields object  */
  fields.putInt( "Int1", 1234 );
  fields.putInt( "Int2", 5678 );
  fields.putInt( "Int3",    0 );
```

C code

```
MQeStringHndl hFieldName;
 rc = mqeString_newChar8(&errStruct, &hFieldName, "A Field Name");
 rc = mqeFields_putInt32(hNewFields,&errStruct,hFieldName,1234);
```

The following example shows how to retrieve values from an MQeFields item:

Java code

```
/* Retrieve an integer value from a fields object  */
  int Int2 = fields.getInt( "Int2" );
```

C code

```
MQEINT32 value;
 rc = mqeFields_getInt32(hNewFields, &errStruct, &value, hFieldName);
```

WebSphere MQ Everyplace provides methods for storing and retrieving the following data types:

- A fixed length array is handled using the **putArrayOf***type* and **getArrayOf***type* methods. *type* can be Byte, Short, Int, Long, Float, or Double.
- The ability to store variable length arrays is possible, but has been deprecated in this release. You can access these arrays using the Java **put***type***Array** and **get***type***Array** calls or the C **put***types* calls. Refer to the WebSphere MQ Everyplace Java Programming Reference and WebSphere MQ Everyplace C Programming Reference for more information.
- The Java codebase has a slightly special form of operations for Float and Double types. This provides compatability with the MicroEdition. Floats are put using an Int representation and Doubles are put using a Long representation. Use the `Float.floatToIntBits()` and `Double.doubleToLongBits()` to perform the conversion. However, this is not required on the C API.

An MQeFields item may be embedded within another MQeFields item by using the **putFields** and **getFields** methods.

The contents of an MQeFields item can be dumped in one of the following forms:

binary Binary form is normally used to send an MQeFields or MQeMsgObject object through the network. The **dump** method converts the data to binary. This method returns a binary byte array containing an encoded form of the contents of the item.

Note: This is not Java serialization.

When a fixed length array is dumped and the array does not contain any elements (its length is zero), its value is restored as null.

encoded string (Java only)

The string form uses the **dumpToString** method of the MQeFields item. It requires two parameters, a template and a title. The template is a pattern string showing how the MQeFields item data should be translated, as shown in the following example:

```
"(#0)#1=#2\r\n"
```

where

#0 is the data type (ascii or short, for example)

#1 is the field name

#2 is the string representation of the value

Any other characters are copied unchanged to the output string. The method successfully dumps embedded MQeFields objects to a string, but due to restrictions, the embedded MQeFields data may not be restored using the **restoreFromString** method.

What are WebSphere MQ Everyplace messages?

Messages are simply collections of data sent by one application and intended for another application. WebSphere MQ Everyplace messages contain application-defined content. When stored, they are held in a queue and such messages may be moved across a WebSphere MQ Everyplace network.

WebSphere MQ Everyplace messages are a special type of MQeFields items, as described in “MQeFields” on page 25. Therefore, you can use methods that are applicable to MQeFields with messages.

Therefore, messages are Fields objects with the addition of some special fields. Java provides a subclass of MQeFields, MQeMsgObject which provides methods to manage these fields. The C codebase does not provide such a subclass. Instead, there are a number of mqeFieldsHelper_operation functions. The following fields form the *`Unique ID`* of a WebSphere MQ Everyplace message:

- In Java, the timestamp, generated when the message is first created or, in C, when the message is first put to a queue
- The name of the queue manager, to which the message is first put

The `Unique ID` identifies a message within a WebSphere MQ Everyplace network provided all queue managers within the WebSphere MQ Everyplace network are named uniquely. However, WebSphere MQ Everyplace does not check or enforce the uniqueness of queue manager names.

In Java, the message is created when an instance of MQeMsgObject is created. In C, the Message is "created", that is UniqueID fields are added, when the message is put to a queue.

The **getMsgUIDFields()**method or **mqeFieldsHelpers_getMsgUidFields()** function accesses the `UniqueID` of a message, for example:

Java code

```
MQeFields msgUID = msgObj.getMsgUIDFields();
```

C code

```
rc = mqeFieldsHelpers_getMsgUidFields(hMgsObj,
       &exceptBlock,&hUIDFields);
```

WebSphere MQ Everyplace adds property related information to a message (and subsequently removes it) in order to implement messaging and queuing operations. When sending a message between queue managers, you can add resend information to indicate that data is being retransmitted.

Typical application-based messages have additional properties in accordance with their purpose. Some of these additional properties are generic and common to many applications, such as the name of the reply-to queue manager. Therefore, WebSphere MQ Everyplace supports the following message properties:

Table 1. Message properties

Property name	Java type	C type	Description
Action	int	MQEINT32	Used by administration to indicate actions such as inquire, create, and delete
Correlation ID	byte[]	MQEBYTE[]	Byte string typically used to correlate a reply with the original message
Errors	MQeFields	MQeFieldsHndl	Used by administration to return error information
Expire time	int or long	MQEINT32 or MQEINT64	Time after which the message can be deleted (even if it is not delivered)
Lock ID	long	MQEINT64	The key necessary to unlock a message
Message ID	byte[]	MQEBYTE[]	A unique identifier for a message
Originating queue manager	string	MQeStringHndl	The name of the queue manager that sent the message
Parameters	MQeFields	MQeFieldsHndl	Used by administration to pass administration details
Priority	byte	MQEBYTE	Relative order of priority for message transmission
Reason	string	MQeStringHndl	Used by administration to return error information
Reply-to queue	string	MQeStringHndl	Name of the queue to which a message reply should be addressed
Reply-to queue manager	string	MQeStringHndl	Name of the queue manager to which a message reply should be addressed
Resend	boolean	MQEBOOL	Indicates that the message is a resend of a previous message
Return code	byte	MQEBYTE	Used by administration to return the status of an administration operation
Style	byte	MQEBYTE	Distinguishes commands from request/reply for example
Wrap message	byte[]	MQEBYTE[]	Message wrapped to ensure data protection

The following table lists the symbolic names corresponding to the message properties given in the previous table.

Table 2. Symbolic names that correspond to message property names

Property name	Java constant	C constant
Action	MQeAdminMsg.Admin_Action	MQE_ADMIN_ACTION
Correlation ID	MQe.Msg_CorrelID	MQE_MSG_CORRELID
Errors	MQeAdminMsg.Admin_Errors	MQE_ADMIN_ERRORS
Expire time	MQe.Msg_ExpireTime	MQE_MSG_EXPIRETIME
Lock ID	MQe.Msg_LockID	MQE_MSG_LOCKID
Message ID	MQe.Msg_MsgID	MQE_MSG_MSGID
Originating queue manager	MQe.Msg_OriginQMgr	MQE_MSG_ORIGIN_QMGR
Parameters	MQeAdminMsg.Admin_Params	MQE_ADMIN_PARAMS

Table 2. Symbolic names that correspond to message property names (continued)

Property name	Java constant	C constant
Priority	MQe.Priority	MQE_MSG_PRIORITY
Reason	MQeAdminMsg.Admin_Reason	MQE_ADMIN_REASON
Reply-to-queue	MQe.Msg_ReplyToQ	MQE_MSG_REPLYTO_Q
Reply-to queue manager	MQe.Msg_ReplyToQMgr	MQE_MSG_REPLYTO_QMGR
Resend	MQe.Msg_Resend	MQE_MSG_RESEND
Return code	MQeAdminMsg.Admin_RC	MQE_ADMIN_RC
Style	MQe.Msg_Style	MQE_MSG_STYLE
Wrap message	MQe.Msg_WrapMsg	MQE_MSG_WRAPMSG

In all cases, a defined constant allows the property name to be carried in a single byte. For example, priority (if present) affects the order in which messages are transmitted, correlation ID triggers indexing of a queue for fast retrieval of information, expire time triggers the expiry of the message, and so on. Also, the default message dump command minimizes the size of the generated byte string for more efficient message storage and transmission.

The WebSphere MQ Everyplace `Message ID` and `Correlation ID` allow the application to provide an identity for a message. These are also used in interactions with the rest of the WebSphere MQ family:

Java

```
MQeMsgObject msgObj = new MQeMsgObject;
msgObj.putArrayOfByte( MQe.Msg_ID, MQe.asciiToByte( "1234" ));
```

C

```
rc = mqeFields_putArrayOfByte(hMsg,&exceptBlock,
     MQE_MSG_MSGID,pByteArray,sizeByteArray);
```

`Priority` contains message priority values. Message priority is defined as in other members of the WebSphere MQ family. It ranges from 9 (highest) to 0 (lowest):

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putByte( MQe.Msg_Priority, (byte)8 );
```

C

```
rc = mqeFields_putByte(hsg,&exceptBlock, MQE_MSG_PRIORITY, (MQEBYTE)8);
```

Applications can create fields for their own data within messages:

Java

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "PartNo", "Z301" );
msgObj.putAscii( "Colour", "Blue" );
msgObj.putInt( "Size", 350 );
```

C

```
MQeFieldsHndl hPartMsg;
MQeStringHndl hSize_FieldLabel;
rc = mqeFields_new(&exceptBlock,&hPartMsg);
rc = mqeString_newUtf8(&exceptBlock,
        &hSize_FieldLabel,"Size");
```

```
rc = mqeFields_putInt32(hPartMsg,
        &exceptBlock,hSize_FieldLabel,350);
```

The priority of the message is used, in part, to control the order in which messages are removed from the queue. If the message does not specify any, then the queue default priority is used . This, unless changed, is 4. However, the application must interpret the different levels of priority.

In Java, you can extend the MQeMsgObject to include some methods that assist in creating messages, as shown in the following example:

```
package messages.order;
import com.ibm.mqe.*;

/*** This class defines the Order Request format */
public class OrderRequestMsg extends MQeMsgObject
{

  public OrderRequestMsg() throws Exception
  {
  }

 /*** This method sets the client number */
  public void setClientNo(long aClientNo) throws Exception
  {
    putLong("ClientNo", aClientNo);
  }

 /*** This method returns the client number */
  public long getClientNo() throws Exception
  {
    return getLong("ClientNo");
  }
```

To find out the length of a message, you can enumerate on the message as each data type has methods for getting its length.

Message Filters

Filters allow WebSphere MQ Everyplace to perform powerful message searches. Most of the major queue manager operations support the use of filters. You can create filters using MQeFields.

Using a filter, for example in a **getMessage()** call, causes an application to return the first available message that contains the same fields and values as the filter. The following examples create a filter that obtains the first message with a message id of "1234":

Java

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte( MQe.Msg_MsgID,
    MQe.AsciiToByte( "1234" ) );
```

C

rc = mqeFields_putArrayOfByte(hMsg,,
&exceptBlock, MQE_MSG_MSGID,
pByteArray, sizeByteArray);

You can use this filter as an input parameter to various API calls, for example `getMessage`.

Message Expiry

Queues can be defined with an expiry interval. If a message has remained on a queue for a period of time longer than this interval then the message is automatically deleted. When a message is deleted, a queue rule is called. Refer to Chapter 3, Rules, of the WebSphere MQ Everyplace System Programming Guide for information on queue rules. This rule cannot affect the deletion of the message, but it does provide an opportunity to create a copy of the message.

Messages can also have an expiry interval that overrides the queue expiry interval. You can define this by adding a C `MQE_MSG_EXPIRETIME` or Java `MQe.Msg_ExpireTime` field to the message. The expiry time is either relative (expire 2 days after the message was created), or absolute (expire on November 25th 2000, at 08:00 hours). Relative expiry times are fields of type Int or MQEINT32, and absolute expiry times are fields of type Long or MQEINT64.

In the example below, the message expires 60 seconds after it is created (60000 milliseconds = 60 seconds).

```
/* create a new message     */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* expiry time of sixty seconds after message was created  */
msgObj.putInt( MQe.Msg_ExpireTime, 60000 );
```

In the example below, the message expires on 15th May 2001, at 15:25 hours.

```
/* create a new message     */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* create a Date object for 15th May 2001, 15:25 hours   */
Calendar calendar = Calendar.getInstance();
calendar.set( 2001, 04, 15, 15, 25 );
Date expiryTime = calendar.getTime();
/* add expiry time to message    */
msgObj.putLong( MQe.Msg_ExpireTime, expiryTime.getTime() );
/* put message onto queue    */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

Checking for expired messages

A message is checked for expiry when:

It is added to a queue
Expiry can occur when a message is added from the local API, pulled down via a Home Server Queue, or pushed to a queue.

It is removed from a queue
Expiry can occur when a message can be removed from the local API, or when a message is pulled remotely.

A queue is activated
When a queue is activated, a reference to the queue is created in memory. Any message that has expired is removed. The state of the message is irrelevant to this operation.

A queue is deleted
If an admin message arrives to delete a queue, the queue must be empty first. Therefore, before this check is done, any expired messages are removed from the queue. The state of the message is irrelevant to this operation.

A queue is checked for size

If an admin message arrives to inquire on the size of a queue, the queue is first purged of admin messages.

You can add a queue rule to notify you when messages expire. However, in a certain situation between two queue managers, a message may seem to expire twice. This is not because the message has been duplicated, but is outlined in the following paragraph.

Assume that an asynchronous queue has a message on it due to expire at 10:00 1st Jan 2005. All messages on such queues are transmitted using a 2 stage process. This process is equivalent to a putMessage and confirmPutMessage pair of operations. Suppose that the first transmission stage occurs at 09:55. A reference to the message appears on the remote queue manager. However, it is not yet available to an application on that queue manager. Then, if the network fails until 10:05, the expiry time of the message is missed. Therefore, the message expires on the remote queue and the queue expiry rule gets fired. Also, in due course, the queue expiry rule gets fired on the destination queue manager.

Accuracy of expiry

The expiry time can be calculated to the millisecond. For correct operation the clocks of the machines running the queue managers must be accurately aligned. Failure to do this within accuracy determined by your choice of expiry times causes messages to appear active on one queue manager, while they have expired on others. Ensure that you use the correct field type for the expiry value. An int (32 bit) field is used for relative expiry times, and a long (64 bit) field is used for absolute times. The field name is the same in both cases.

Chapter 5. Queues

This chapter provides information on different types of WebSphere MQ Everyplace queues under the following headings:

- What are WebSphere MQ Everyplace queues?
- Queue names
- Queue types
- Queue persistent storage
- Queue ordering

What are WebSphere MQ Everyplace queues?

WebSphere MQ Everyplace queues store messages. The queues are not directly visible to an application and all interactions with the queues take place through queue managers. For queue proxies, in the case of Java queue rules, refer to Chapter 3, Rules, of the WebSphere MQ Everyplace System Programming Guide. Each queue manager can have queues that it manages and owns. These queues are known as *local* queues. WebSphere MQ Everyplace also allows applications to access messages on queues that belong to another queue manager. These queues are known as *remote* queues. Similar sets of operations are available on both local and remote queues, with the exception of defining message listeners. Refer to "Message listeners" on page 60 for more information. The Queue types section provides more information on the different types of queue you can have.

Messages are held in the queue's persistent store. A queue accesses its persistent store through a *queue store adapter*. These adapters are interfaces between WebSphere MQ Everyplace and hardware devices, such as disks or networks, or software stores such as a database. Adapters are designed to be pluggable components, allowing the protocols available to talk to the device to be easily changed.

Queues may have characteristics, such as authentication, compression and encryption. These characteristics are used when a message object is stored on a queue. Chapter 9, *Security* provides more information on this.

Queue Names

WebSphere MQ Everyplace queue names can contain the following characters:

- Numerics 0 to 9
- Lower case a to z
- Upper case A to Z
- Underscore _
- Period .
- Percent %

There are no inherent name length limitations in WebSphere MQ Everyplace.

Queues are configured using administration messages. Refer to the WebSphere MQ Everyplace Configuration Guide for more information on configuring WebSphere MQ Everyplace using administration messages.

Queue properties are shown in the following table. Not all the properties shown apply to all the queue types:

Table 3. Queue properties

Property	Explanation	Java type	C type
Admin_Class	Queue class	String	admtype
Admin_Name	ASCII queue name	String	admname
Queue_Active	Queue in active/inactive state	boolean	qact
Queue_AttRule	Rule class controlling security operations	String	qar
Queue_Authenticator	Authenticator class	String	qau
Queue_BridgeName	Owning WebSphere MQ bridge name	String	q-mq-bridge
Queue_ClientConnection	Client connection name	String	q-mq-client-con
Queue_CloseIdle	Close the connection to the remote queue manager once all messages have been transmitted	boolean	qcwi
Queue_CreationDate	Date that the queue was created	long	qcd
Queue_Compressor	Compressor class		qco
Queue_Cryptor	Cryptor class		qcr
Queue_CurrentSize	Number of messages on the queue		qcs
Queue_Description	Unicode description		qd
Queue_Expiry	Expiry time for messages		qe
Queue_ FileDesc	Location and adapter for the queue		qfd
Queue_MaxMsgSize	Maximum length of messages allowed on the queue		qms
Queue_MaxQSize	Maximum number of messages allowed		qmqs
Queue_Mode	Synchronous or asynchronous		qm
Queue_MQQMgr	WebSphere MQ queue manager proxy		
Queue_Priority	Priority to be used for messages (unless overridden by a message value)		qp
Queue_QAliasNameList	Alternative names for the queue	String[]	qanl
Queue_QMgrName	Queue manager owning the real queue		qqmn
Queue_QMgrNameList	Queue manager targets		?
Queue_RemoteQName	Remote WebSphere MQ field name		?
Queue_Rule	Rule class for queue operations		qr
Queue_QTimerInterval	Delay before processing pending messages		qti
Queue_TargetRegistry	Target registry type		qtr
Queue_Transporter	Transporter class		qtc

Table 3. Queue properties (continued)

Property	Explanation	Java type	C type
Queue_TransporterXOR	Transporter to use XOR compression		qtxor
Queue_Transformer	Transformer class		q-mq-transformer

For the symbolic names corresponding to the queue properties in the previous table, refer to the WebSphere MQ Everyplace Java Programming Reference and the WebSphere MQ Everyplace C Programming Reference.

Queue types

There are several different types of *queues* that you can use in a WebSphere MQ Everyplace environment. The following are the types of queue that are available in the WebSphere MQ Everyplace development package:

Local queue

The simplest type of queue is a local queue. This type of queue is local to, and owned by, a specific queue manager. It is the final destination for all messages. Applications on the owning queue manager can interact directly with the queue to store messages in a safe and secure way, excluding hardware failures or loss of the device.

You can use local queues either online or offline, either connected or not connected to a network. Queues can also have security attributes set, in a very similar manner to protecting messages with attributes. Chapter 8, “Security”, on page 79, discusses queue security in more detail.

Access to messages on local queues is always synchronous, which means that the application waits until WebSphere MQ Everyplace returns after completing the operation, for example a put, get, or browse operation.

The queue owns access and security and may allow a remote queue manager to use these characteristics, when connected to a network. This allows others to send or receive messages to the queue.

Remote queue

A remote queue is a local queue belonging to another queue manager. This remote queue definition exchanges messages with the remote local queue.

WebSphere MQ Everyplace can establish remote queues automatically. If you attempt to access a queue on another queue manager, for example to send a message to that queue, WebSphere MQ Everyplace looks for a remote queue definition. If one exists it is used. If not, *queue discovery* occurs.

Note: The concept of queue discovery does not apply to the C codebase. WebSphere MQ Everyplace discovers the authentication, cryptography, and compression characteristics of the real queue and creates a remote queue definition. Such queue discovery depends upon the target being accessible. If the target is not accessible, a remote definition must be supplied in some other way. When queue discovery occurs, WebSphere MQ Everyplace sets the access mode to synchronous, because the queue is now known to be synchronously available.

Synchronous remote queues are queues that can be accessed only when connected to a network that communicates with the owning queue manager. If the network is not established, the operations return an error. The owning queue controls the access permissions and security requirements needed to access the queue. It is the application's responsibility to handle any errors or retries when sending or receiving messages, because, in this case, WebSphere MQ Everyplace is no longer responsible for once and once-only assured delivery.

Asynchronous remote queues are queues used to send messages to remote queues and can store messages pending transmission. They cannot remotely retrieve messages. If the network connection is established, messages are sent to the owning queue manager and queue. However, if the network is not connected, messages are stored locally until there is a network connection and then the messages are transmitted. This allows applications to operate on the queue when the device is offline. As a result, these queues temporarily store messages at the sending queue manager while awaiting transmission.

Store-and-forward queue

Note: Store-and-forward queues are not implemented in the C codebase.

A store-and-forward queue stores messages on behalf of one or more remote queue managers until they are ready to receive them. This can be configured to perform either of the following:

- Push messages either to the target queue manager or to another queue manager between the sending and the target queue managers.
- Wait for the target queue manager to pull messages destined for it.

A store-and-forward queue stores messages associated with one or more target queue manager destinations. Messages addressed to a specific or target queue manager are placed on the relevant store-and-forward queue. The store-and-forward queue can optionally have a forwarding queue manager name set. If this name is set, the queue attempts to send all its messages to that named queue manager. If the name is not set, the queue just holds the messages.

Note: A store-and-forward queue and a *home server queue* should not have the same target queue manager. A store-and-forward queue with a queue QueueManagerName that is not the same as its host QueueManagerName, attempts to push messages to the remote queue manager. If that remote queue manager has a home server queue, it may attempt to pull the same message simultaneously, causing the message to lock.

Store-and-forward queues can hold messages for many target queue managers, or there may be one store-and-forward queue for each target queue manager.

This type of queue is normally, but not necessarily, defined on a server or gateway in Java only. Multiple store-and-forward queues can exist on a single queue manager, but the target names must not be duplicated. The contents of a store-and-forward queue are not available to application programs. Likewise a message sending application is quite unaware of the presence or role of store-and-forward queues in message transmission.

Dead-letter queue

WebSphere MQ Everyplace has a similar dead-letter queue concept to WebSphere MQ. Such queues store messages that cannot be delivered. However, there are important differences in the manner in which they are used.

- In WebSphere MQ, if a message is being moved from queue manager `A` to queue manager `B`, then if the target queue on queue manager `B` cannot be found, the message can be placed on the *receiving queue manager's* (`B`'s) dead-letter queue.
- In WebSphere MQ Everyplace, if home-server queue on a client pulls a message from a server and is not able to deliver the message to a local queue and the client has a dead letter queue, the message will be placed on the client's dead letter queue.

  Note: In C, the Dead letter queue is just a local queue with a specific name.

  The use of dead-letter queues with an WebSphere MQ bridge needs special consideration. Refer to the chapter on the WebSphere MQ bridge in the WebSphere MQ Everyplace Configuration Guide for more details.

Administration queue

The administration queue is a specialized queue that processes administration messages.

Messages put to the administration queue are processed internally. Because of this applications cannot get messages directly from the administration queue. Only one message is processed at a time, other messages that arrive while a message is being processed are queued up and processed in the sequence in which they arrive.

Home-server queue

This type of queue usually resides on a client and points to a store-and-forward queue on a server known as the *home-server*. The home-server queue pulls messages from the home-server store-and-forward queue when the client connects on the network.

In Java, home-server queues normally have a polling interval that causes them to check for any pending messages on the server while the network is connected.

When this queue pulls a message from the server, it uses assured message delivery to put the message to the local queue manager. The message is then stored on the target queue.

Home-server queues have an important role in enabling clients to receive messages over client-server connections.

WebSphere MQ bridge queue

Note: The C codebase does not support WebSphere MQ bridge queues.

This type of queue is always defined on a WebSphere MQ Everyplace gateway queue manager and provides a path from the WebSphere MQ Everyplace environment to the WebSphere MQ environment. The WebSphere MQ bridge queue is a remote queue definition that refers to a queue residing on a WebSphere MQ queue manager.

Applications can use **put**, **get**, and **browse** operations on this type of queue, as if it were a local WebSphere MQ Everyplace queue.

Queue persistent storage

Local queues and asynchronous remote queues store messages and therefore have properties to determine how and where the messages are stored.

The message store determines how the messages are mapped to the storage medium. The C and Java versions of WebSphere MQ Everyplace support a default message store, allowing long file names. The Java version of WebSphere MQ Everyplace has two additional message stores, `MQeShortFilenameMessageStore` that ensures the file name does not exceed eight characters, and the `MQe4690ShortFilenameMessageStore` that supports the default file system on a 4690. A storage adapter provides the message store access to the storage medium, the Java and C versions of WebSphere MQ Everyplace provide disk adapters with the Java version also providing a case insensitive adapter and a memory adapter.

The backing store used by a queue can be changed using a WebSphere MQ Everyplace administration message. Changing the backing store is not allowed while the queue is active or contains messages. If the backing store used by the queue allows the messages to be recovered in the event of a system failure, then this allows WebSphere MQ Everyplace to assure the delivery of messages.

WebSphere MQ Everyplace connection definitions

WebSphere MQ Everyplace supports a method of establishing logical connections between queue managers, in order to send or receive data.

WebSphere MQ Everyplace clients and servers communicate over connections called *client/server channels*.

Client/server channels have the following attributes:

- They are *dynamic*, that is created on demand. This differentiates them from WebSphere MQ connections which have to be explicitly created.
- You can only establish the connection from the client-side.
- A client can connect to many servers, with each connection using a separate channel.
- The server-side queue manager can accept many connections simultaneously, from a multitude of different clients, using a listener for each protocol.
- They work through a Firewall, if the server-side of the connection is behind the Firewall. However, this depends on the configuration of the Firewall.
- They are *unidirectional* and support the full range of functions provided by WebSphere MQ Everyplace, including both synchronous and asynchronous messaging.

  Note: Unidirectional means that the client can send data to, or request data from the server, but the server-side cannot initiate requests of the client.

Standard connections, used for the client/server connection style, are unidirectional, but depend on a listener at the server, as servers cannot initiate data transfer. The client initiates the connection request and the server responds. A server can usually handle multiple incoming requests from clients. Over a standard connection, the client has access to resources on the server. If an application on the server needs synchronous access to resources on the client, a second connection is required where the roles are reversed. However, because standard connections are themselves bidirectional, messages destined for a client from its server's transmission queue, are delivered to it over the standard (client/server) connection that it initiated.

A client can be a client to multiple servers simultaneously. The client/server connection style is generally suited for use through Firewalls, because the target of the incoming connection is normally identified as being acceptable to the Firewall.

Note: Supposing there are two server queue managers, SQM1 and SQM2. SQM2 has listener address host 2: 8082. Also, suppose that SQM1 has a connection to SQM2 and a listener addresss, host 1:8081. If you create a connection definition on a client queue manager, named SQM2 with address host 1: 8081, this transports commands for SQM2 to SQM1, which then transports them to SQM2. Avoid this construct, as it is inefficient.

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same attribute rule, `examples.rules.AttributeRule` and `com.ibm.mqe.MQeAttributeRule` are treated as the same rule. If this is not a desirable behaviour, you can specify a null rule for the target queue. In this case, com.ibm.mqe.`MQeAttributeDefaultRule` takes effect.

Connections can have various attributes or characteristics, such as authentication, cryptography, compression, or the transmission protocol to use. Different connections can use different characteristics. Each connection can have its own value set for each of the following attributes:

Authenticator
This attribute causes authentication to be performed. This is a security function that challenges the putting application environment or user to prove their identity. It has a value of either `NULL` or an *authenticator* that can perform user or connection authentication.

Cryptor
This attribute causes encryption and decryption to be performed on messages passing through the channel. This is a security function that encodes the messages during transit so that you cannot read them without the decoding information. Either null or a *cryptor* that can perform encryption and decryption.

The simplest type of cryptor is MQeXorCryptor, which encrypts the data being sent by performing an exclusive-OR of the data. This encryption is not secure, but it modifies the data so that it cannot be viewed. In contrast, MQe3DESCryptor implements triple DES, a symmetric-key encryption method.

Channel
The class providing the transport services.

Compressor
This attribute causes compression and decompression to be performed on messages passing through the channel. This attempts to reduce the size of messages while they are being transmitted and stored. Either null or a *compressor* that can perform data compression and decompression. The simplest type of compressor is the MQeRleCompressor, which compresses the data by replacing repeated characters with a count.

Destination
The server and port number for the connection. The target for this connection, for example SERVER.XYZ.COM

Typically, authentication only occurs when setting up the connection. All flows normally use compressors and cryptors.

For more information about authenticators, compressors, and cryptors, see Chapter 8, “Security”, on page 79.

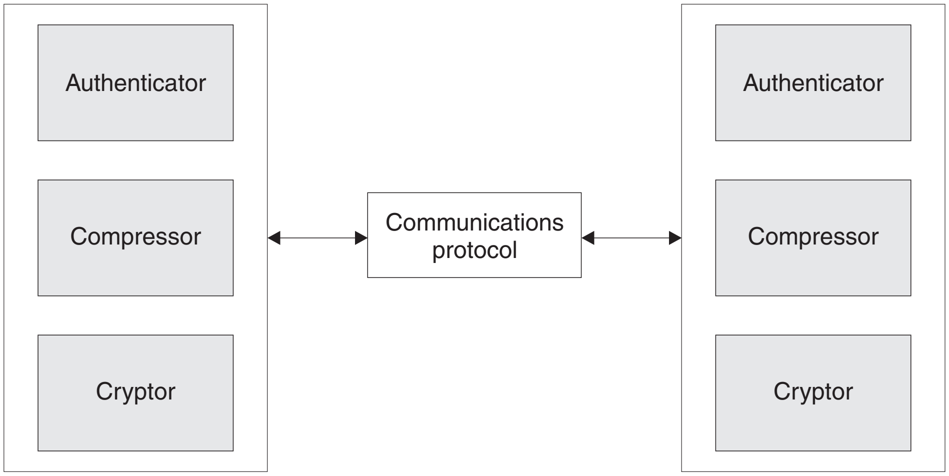


Figure 1. WebSphere MQ Everyplace connection

You can establish WebSphere MQ Everyplace connections using a variety of protocols allowing them to connect in a number of different ways, for example:

- Permanent connection, for example a LAN, or leased line
- Dial out connection, for example using a standard modem to connect to an Internet service provider (ISP)
- Dial out and answer connection, using a CellPhone, or ScreenPhone for example

WebSphere MQ Everyplace implements the communications protocols as a set of adapters, with one adapter for each of the supported protocols. This enables you to add new protocols.

Using queue aliases

Aliases can be assigned for WebSphere MQ Everyplace queues to provide a level of indirection between the application and the real queues. Hence the attributes of a queue that an alias relates to can be changed without the application needing to change. For instance, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

The following examples illustrate some of the ways that aliasing can be used with queues.

Examples of queue aliasing

Merging applications

Suppose you have the following configuration:

- A client application that puts data to queue `Q1`
- A server application that takes data from `Q1` for processing
- A client application that puts data to queue `Q2`
- A server application which takes data from `Q2` for processing

Some time later the two server applications are merged into one application supporting requests from both the client applications. It may now be appropriate for the two queues to be changed to one queue. For example, you may delete `Q2`, and add an alias of the `Q1` queue, calling it `Q2`. Messages from the client application that previously used `Q2` are automatically sent to `Q1`.

Upgrading applications

Suppose you have the following configuration:

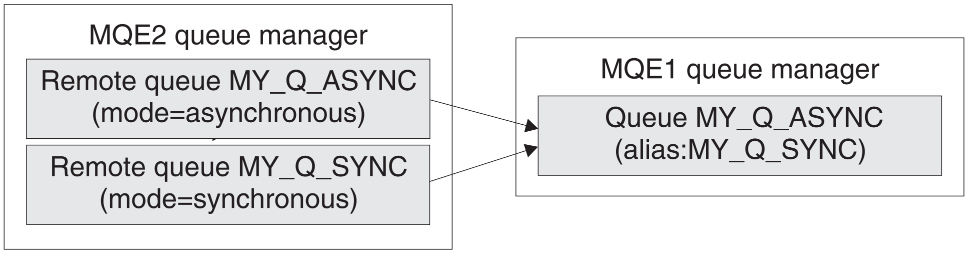
- A queue `Q1`
- An application that gets messages from `Q1`
- An application that puts messages to `Q1`

You then develop a new version of the application that gets the messages. You can make the new application work with a queue called `Q2`. You can define a queue called `Q2` and use it to exercise the new application. When you want it to go live, you let the old version clear all traffic off the `Q1` queue, and then create an alias of `Q2` called `Q1`. The application that puts to `Q1` will still work, but the messages will end up on `Q2`.

Using different transfer modes to a single queue

Suppose you have a queue `MY_Q_ASYNC` on queue manager `MQE1`. Messages are passed to `MY_Q_ASYNC` by a different queue manager `MQE2`, using a remote queue definition that is defined as an asynchronous queue. Now suppose your application periodically wants to get messages in a synchronous manner from the `MY_Q_ASYNC` queue.

The recommended way to achieve this is to add an alias to the `MY_Q_ASYNC` queue, perhaps called `MY_Q_SYNC`. Then define a remote queue definition on your `MQE2` queue manager, that references the `MY_Q_SYNC` queue. This provides you with two remote queue definitions. If you use the `MY_Q_ASYNC` definition, the messages are transported asynchronously. If you use the `MY_Q_SYNC` definition, synchronous message transfer is used.



Both remote queues reference the same queue,
using different attributes and different names

Figure 2. Two modes of transfer to a single queue

Chapter 6. Queue manager operations

This chapter explains in detail the messaging operations that you can perform on a queue manager. Chapter 1, “Introduction”, on page 1 provides a high-level description of the services provided by WebSphere MQ Everyplace queues and queue managers, while Chapter 3, ″Messaging″ and Chapter 4, ″Queues″ of this book explain messaging and detail the different types of queues that you can have.

This chapter describes the services, functions, and uses of queue managers under the following headings:

- What is a WebSphere MQ Everyplace queue manager?
- The queue manager lifecycle
- Starting queue managers
- Messaging lifecycle
- Deleting queue managers
- Servlet
- Security
- Aliases

What is a WebSphere MQ Everyplace queue manager?

The WebSphere MQ Everyplace queue manager is the focal point of the WebSphere MQ Everyplace system. It provides:

- A central point of access to a messaging and queueing network for WebSphere MQ Everyplace applications
- Optional client-side queuing
- Optional administration functions
- Once and once-only assured delivery of messages
- Recovery from failure conditions
- Extendable rules-based behavior

Unlike base WebSphere MQ, WebSphere MQ Everyplace has a single queue manager type. However, you can program WebSphere MQ Everyplace queue managers to act as traditional clients or servers. You can also customize queue manager behavior using rules. The WebSphere MQ Everyplace queue manager is embedded within user written programs and these programs can run on any WebSphere MQ Everyplace supported device or platform.

You can configure queue managers in a number of different ways, the main types being client, server, and gateway. Refer to “Starting queue managers” on page 44 for descriptions of these types. You can also update the queue store of a queue manager using administration messages. For more information on administration messages, refer to the WebSphere MQ Everyplace Configuration Guide.

A WebSphere MQ Everyplace queue manager can control the various types of queue that are described in “Queue Names” on page 33. Communication with other queue managers on the WebSphere MQ messaging network can be synchronous or asynchronous. If you want to use synchronous communications, the originator, and the target WebSphere MQ Everyplace queue managers must both be available on the network. Asynchronous communication allows a WebSphere MQ Everyplace application to send messages even when the remote queue manager is offline.

The queue manager lifecycle

Typically, an application creates a new queue manager, configures it with a number of queues, and then frees the queue manager. An application also opens an existing queue manager, starts it, carries out messaging operations, and then stops. A further administration program can reopen the queue manager, remove all of its queues, and then stop. The following diagram displays this information:

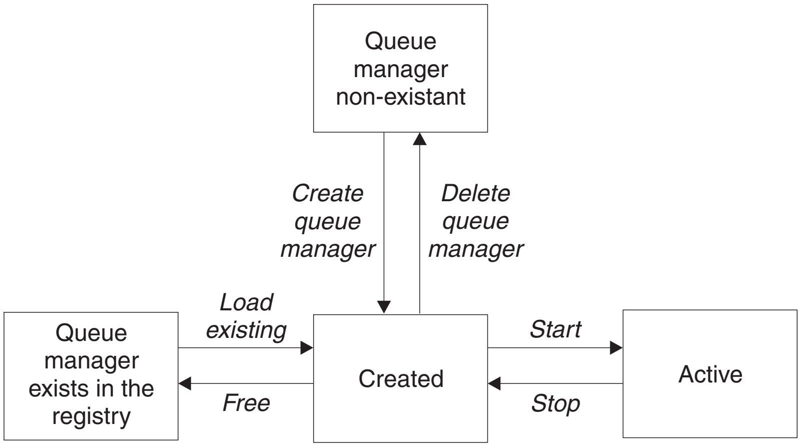


Figure 3. The queue manager lifecycle

Starting queue managers

Queue managers need to be created before use. The creation step uses the **QueueManagerConfigure** Java class or the C administration API to create persistent queue manager data in a registry. The queue manager then uses the registry each time its starts. The WebSphere MQ Everyplace Configuration Guide provides further information on configuring queue managers.

Starting queue managers in Java

Normally, creating and starting a queue manager can require a large set of parameters. Therefore, the required parameters are supplied as an instance of **MQeFields**, storing the values as fields of correct type and name.

The parameters fall into two categories, queue manager parameters and registry parameters. Each of these categories is represented by its own **MQeFields** instance, and both are also enclosed in an **MQeFields** instance. The following Java example explains this concept, passing the queue managers name, ″ExampleQM″ and the location of a registry, ″C:\ExampleQM″:

```
/*create fields for queue manager parameters and place the queue manager name
MQeFields queueManagerParameters = new MQeFields();
queueManagerParameters.putAscii(MQeQueueManager.Name, "ExampleQM");

/*create fields for registry parameters and place the registry location
MQeFields registryParameters = new MQeFields();
registryParameters.putAscii(MQeRegistry.DirName, "C:\\ExampleQM\\registry");

/*create fields for combined parameters and place the two sub fields
```

```
MQeFields parameters = new MQeFields();
parameters.putFields(MQeQueueManager.Registry, queueManagerParameters);
parameters.putFields(MQeQueueManager.Registry, registryParameters);
```

Wherever you see ″`initialize the parameters`″ in code snippets, prepare a set of parameters as shown in the example, including the appropriate options. Only one queue manager name and one registry location are mandatory.

Queue manager parameters

The following lists the parameter names that you can pass to the queue manager and the registry:

Queue manager Parameters

MQeQueueManager.Name(ascii)
This is the name of the queue manager being started.

Registry Parameters

MQeRegistry.LocalRegType(ascii)
This is the type of registry being opened. WebSphere MQ Everyplace currently supports:

file registry
Set this parameter to `com.ibm.mqe.registry.MQeFileSession`.

private registry
Set this parameter to `com.ibm.mqe.registry.MQePrivateSession`.

You also need a private registry for some security features. Chapter 9, *Security*, for more information on security.

MQeRegistry.DirName(ascii)
This is the name of the directory holding the registry files. You must pass this parameter for a file registry.

MQeRegistry.PIN(ascii)
You need this PIN for a private registry.

Note: For security reasons, WebSphere MQ Everyplace deletes the PIN and KeyRingPassword, if supplied, from the startup parameters as soon as the queue manager is activated.

MQeRegistry.CAIPAddrPort(ascii)
You need this address and port number of a mini-certificate server for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.CertReqPIN(ascii)
This is the certificate request number allocated by the mini-certificate administrator to allow the registry to obtain its credentials. You need this for auto-registration, so that the queue manager can obtain its credentials from the mini-certificate server.

MQeRegistry.Separator(ascii)
This is used to specify a non-default separator. A separator is the character used between the the components of an entry name, for example `<QueueManager><Separator><Queue>`. Although this parameter is specified as a string, it must contain a single character. If it contains more than one, only the first character is used. Use the same separator for each registry

opened and do not change it once a registry is in use. If you do not specify this parameter, the separator defaults to ″+″.

MQeRegistry.RegistryAdapter(ascii)
This is the class, or an alias that resolves to a class, of the adapter that the registry uses to store its data. You must include this class if you want the registry to use an adapter other than the default **MQeDiskFieldsAdapter**. You can use any valid storage adapter class.

You always need the first two parameters. The last two are for auto-registration of the registry if it wishes to obtain credentials from the mini-certificate server.

MQeRegistry.RegistryAdapter (ascii)
The class, (or an alias that resolves to a class), of the adapter that the registry uses to store its data. This value should be included if you want the registry to use an adapter other than the default MQeDiskFieldsAdapter. Any valid adapter class can be used.

A queue manager can run:

- As a client
- As server
- In a servlet

The following sections describe the example client, servers and servlet that are provided in the examples.queuemanager package. All queue managers are constructed from the same base WebSphere MQ Everyplace components, with some additions that give each its unique properties. WebSphere MQ Everyplace provides an example class, `MQeQueueManagerUtils`, that encapsulates many of the common functions.

All the examples require parameters at startup. These parameters are stored in standard ini files. The ini files are read and the data is converted into an MQeFields object. The `loadConfigFile()` method in the MQeQueueManagerUtils class performs this function.

Starting queue managers in C

The `mqeQueueManager_new` function loads a queue manager for an established registry. To do this, you need information supplied by a queue manager parameter structure and a registry parameter structure.

The following example shows how you can set these structures to their default values, supplying only the directories of the queue store and registry:

```
MQeQueueManagerHndl hQueueManager;
MQeRegistryParms regParms = REGISTRY_INIT_VAL;
MQeQueueManagerParms qmParms = QMGR_INIT_VAL;
regParms.hBaseLocationName = hRegistryDirectory;
qmParms.hQueueStore = hStore;
qmParms.opFlags = QMGR_Q_STORE_OP;
rc = mqeQueueManager_new(&exceptBlock,
         &hQueueManager, hQMName,
        &regParams, &qmParms);
```

This creates a queue manager and loads its persistant information from the registry and creates queues. However, you must start the queue manager to:

- Create messages
- Get and put messages

- Process administration messages, using the administration queue

Note: In C, the queues are activated on starting the queue manager.

To start the queue manager, use

```
rc = mqeQueueManager_start(&hQueueManager, &exceptBlock);
```

Once the queue manager is started, messaging operations can take place and any queues that have messages on them are loaded.

To stop the queue manager, use:

```
rc = mqeQueueManager_stop(&hQueueManager, &exceptBlock);
```

Once stopped, you can restart the queue manager as required.

At the end of the application, you must free the queue manager to release any resources it uses, for example memory. First, stop the queue manager and then use:

```
rc = mqeQueueManager_free(&hQueueManager, &exceptBlock);
```

Registry parameters for a queue manager

The registry is the primary store for queue manager-related information; one exists for each queue manager. Every queue manager uses the registry to hold its:

- Queue manager configuration data
- Communications listener resource definitions
- Queue definitions
- Remote queue definitions
- Remote queue manager definitions
- User data, including configuration-dependent security information
- Optional bridge resource defintitions

Registry type

`MQE_REGISTRY_LOCAL_REG_TYPE`

The type of registry being opened. *file registry* and *private registry* are currently supported. A private registry is required for some of the security features. See Chapter 8, “Security”, on page 79.

For a file registry this parameter should be set to:

`com.ibm.mqe.registry.MQeFileSession`

For a private registry it should be set to:

`com.ibm.mqe.registry.MQePrivateSession`

Aliases can be used to represent these values.

Client queue managers

A client typically runs on a device platform, and provides a queue manager that can be used by applications on the device. It can open many connections to other queue managers.

A server usually runs for long periods of time, but clients are started and stopped on demand by the application that use them. If multiple applications want to share a client , the applications must coordinate the starting and stopping of the client.

Starting a client queue manager

Starting a client queue manager involves:

1. Ensuring that there is no client already running. (Only one client is allowed per Java Virtual Machine.)
2. Adding any aliases to the system
3. Enabling trace if required
4. Starting the queue manager

The following code fragment starts a client queue manager:

```
/*-------------------------------------*/
/* Init - first stage setup          */
/*-------------------------------------*/
public void init( MQeFields parms ) throws Exception
{
  if ( queueManager != null )
/* One queue manager at a time   */
  {
    throw new Exception( "Client already running" );
  }
  sections = parms;
/* Remember startup parms        */
  MQeQueueManagerUtils.processAlias( sections );
/* set any alias names       */

// Uncomment the following line to start trace
  before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeClient Trace", null);
/* Turn trace on   */

  /* Display the startup parameters */
  System.out.println( sections.dumpToString("#1\t=\t#2\r\n"));

  /* Start the queue manage  */
  queueManager = MQeQueueManagerUtils.processQueueManager( sections, null);
}
```

Once you have started the client, you can obtain a reference to the queue manager object either from the static class variable `MQeClient.queueManager` or by using the static method `MQeQueueManager.getReference(queueManagerName)`.

The following code fragment loads aliases into the system:

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )
/* section present ?           */
  {
/* ... yes                     */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );
/* get all the keywords        */
    while ( keys.hasMoreElements() )
/* as long as there are keywords*/
    {
      String    key   = (String) keys.nextElement();
/* get the Keyword   */
      MQe.alias( key, section.getAscii( key ).trim( ) );
/* add              */
    }
  }
}
```

Use the `processAlias` method to add each alias to the system. WebSphere MQ Everyplace and applications can use the aliases once they have been loaded.

Starting a queue manager involves:

1. Instantiating a queue manager. The name of the queue manager class to load is specified in the alias `QueueManager`. Use the WebSphere MQ Everyplace class loader to load the class and call the null constructor.
2. Activate the queue manager. Use the `activate` method, passing the MQeFields object representation of the ini file. The queue manager only makes use of the `[QueueManager]` and `[Registry]` sections from the startup parameters.

The following code fragment starts a queue manager:

```
public static MQeQueueManager processQueueManager( MQeFields sections,
   Hashtable ght ) throws Exception
{
/*                             */
  MQeQueueManager queueManager = null;
/* work variable               */
  if ( sections.contains( Section_QueueManager) )
/* section present ?    */
  {
/* ... yes                     */
   queueManager = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
    if ( queueManager != null )
/* is there a Q manager ?      */
    {
      queueManager.setGlobalHashTable( ght );
      queueManager.activate( sections );
/* ... yes, activate           */
    }
  }
  return( queueManager );
/* return the alloated mgr     */
}
```

Example MQePrivateClient

MQePrivateClient is an extension of MQeClient with the addition that it configures the queue manager and registry to allow for secure queues. For a secure client, the `[Registry]` section of the startup parameters is extended as follows:

```
(ascii)LocalRegType=PrivateRegistry

   Location of the registry

(ascii)DirName=.\ExampleQM\PrivateRegistry
   Adapter on which registry sits
(ascii)Adapter=RegistryAdapter
Network address of certificate authority

(ascii)CAIPAddrPort=9.20.7.219:8082
```

Refer to Chapter 8, “Security”, on page 79 for more details on secure queues and MQePrivateClient.

For MQePrivateClient and MQePrivateServer to work, the startup parameters must *not* contain `CertReqPIN`, *`KeyRingPassword`* and `CAIPAddrPort`.

Server queue managers

A server usually runs on a server platform. A server can run server-side applications but can also run client-side applications. As with clients, a server can open connections to many other queue managers on both servers and clients. One of the main characteristics that differentiate a server from a client is that it can handle many concurrent incoming requests. A server often acts as an entry point for many clients into an WebSphere MQ Everyplace network . WebSphere MQ Everyplace provides the following server examples:

MQeServer
A console based server.

MQePrivateServer
A console based server with enhanced security.

AwtMQeServer
A graphical front end to MQeServer.

MQBridgeServer
In addition to the normal WebSphere MQ Everyplace server functions, this server can send and receive messages to and from other members of the WebSphere MQ family. This server is in package examples.mqbridge.queuemanager and is described in the WebSphere MQ Everyplace Configuration Guide.

Example MQeServer

MQeServer is the simplest server implementation.

When two queue managers communicate with each other, WebSphere MQ Everyplace opens a connection between the two queue managers. The connection is a logical entity that is used as a queue manager to queue manager pipe. Multiple connections may be open at any time.

Server queue managers, unlike client queue managers can have one or more listeners. A listener waits for communications from other queue managers, and processes incoming requests, usually by forwarding them to its owning queue manager. Each listener has a specified adapter that defines the protocol of incoming communications, and also specifies any extra data required.

You can create listeners on the local queue manager using either the **MQeAdministrator** class or administration messages, remotely and locally. However, a remote queue manager must have a listener in order to receiver a message.

This section describes how to create a listener using the **MQeAdministrator** class. As the listener can take a number of arguments, use **MQeProperties** to pass the parameters as name and value string pairs. The adapter defines the parameter names and their values. The following example defines a listener using a TcpipHttp adapter, listening on port 8080:

```
/*create a properties object to pass the
  adapter parameters to the listener
String listenerName = "MyListener";
String adapter = "com.ibm.mqe.adapters.MQeTcpipHttpAdapter";
String port = "8080";
MQeProperties properties = new MQeProperties();
properties.setProperty(MQeCommunicationsAdapter.COMMS_ADAPTER_CLASS,
        adapter);
properties.setProperty(MQeCommunicationsAdapter.COMMS_ADAPTER_PORT,
        port);
properties.setProperty(MQeCommunicationsAdapter.COMMS_ADAPTER_LISTEN,
        true);

/* create an administrator
MQeAdministrator admin = new MQeAdministrator(myQMgr);
```

```
/* created the administrator, now use it to create the listener
admin.listenCreateNew(listenerName, properties,
        timeout, maxChannels);
/* now use the administrator
 itself to start the listener
admin.listenerStart(listenerName);
```

When the listener is started, the server is ready to accept network requests.

When the server is deactivated:

1. The listener is stopped, preventing any new incoming requests
2. The queue manager is closed

Example MQePrivateServer

MQePrivateServer is an extension of MQeServer with the addition that it configures the queue manager and registry to allow for secure queues. See Chapter 8, “Security”, on page 79.

Environment relationship

The following section describes some requirements for running Java and C implementations of WebSphere MQ Everyplace and.

Java code

The java queue manager runs inside an instance of a JVM. You can have only one queue manager per JVM. However, you can invoke multiple instances of the JVM.

Each of these queue managers must have a unique name. Java applications run inside the same JVM as the queue manager they use.

C code

You can run only one queue manager within a native C process. You need multiple processes for multiple queue managers. Each of these queue managers must have a unique name.

Messaging lifecycle

When a message is put to a queue it progresses through a series of states. This section describes these states and related commands or events under the following headings:

- Message states
- Message events
- Message index fields

Although, this section gives brief details of algorithms required for assured message delivery, Chapter 7, *Message Delivery*, provides more information on assured message delivery.

Message states

Most queue types hold messages in a persistent store, for example a hard disk. While in the store, the state of the message varies as it is transferred into and out of the store. As shown in Figure 4 on page 52:

Messaging lifecycle

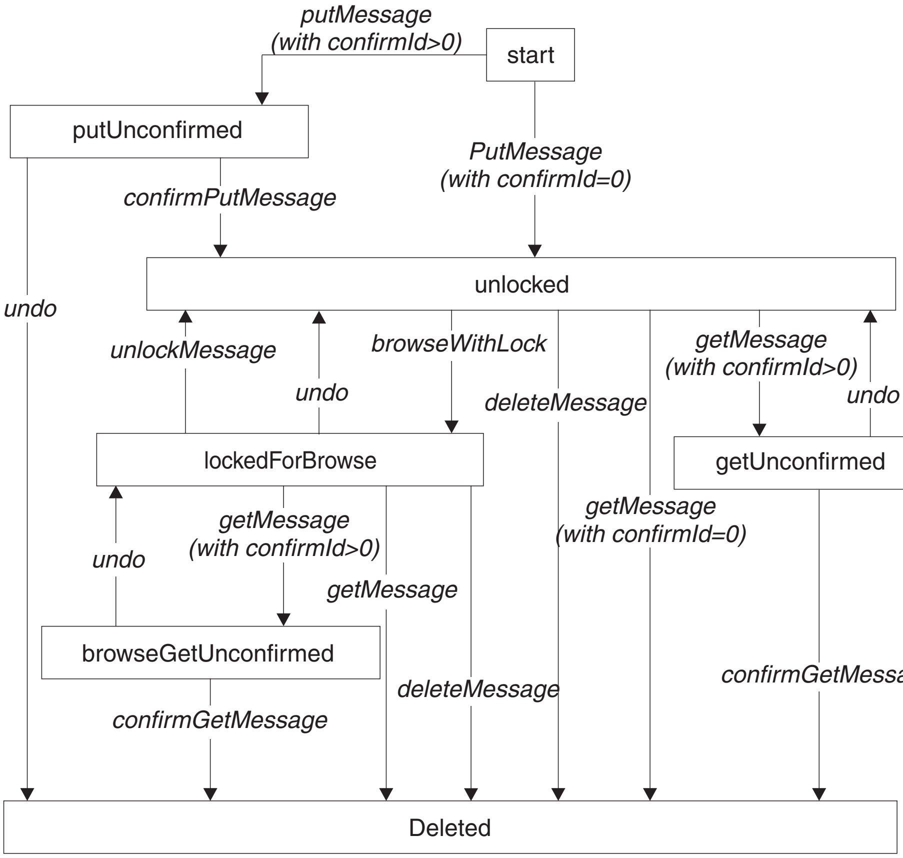


Figure 4. Stored message state flow

In Figure 2, "start" and "deleted" are not actual message states. They are the entry and exit points of the state model. The message states are:

Put unConfirmed
A message is put to the message store of a queue with a `confirmID`. The message is effectively hidden from all actions except `confirmPutMessage` or `undo`.

Unlocked
A message has been put to a queue and is available to all operations.

Locked for Browse
A browse with lock retrieves messages. Messages are hidden from all queries except `getMessage`, `unlock`, `delete`, `undo`, and `unlockMessage`. A `lockID` is returned from the browse operation. You must supply this `lockID` to all other operations.

Get Unconfirmed
A `getMessage` call has been made with a `confirmID`, but the get has not been confirmed. The message is invisible to all queries except `confirmGetMessage`, `confirm`, and `undo`. Each of these actions requires the inclusion of the matching `confirmID` to confirm the get.

Browse Get Unconfirmed
A message got while it is locked for browse. You can do this only by passing the correct `lockID` to the `getMessage` function.

On an asynchronous remote queue, other states exist where a message is being transmitted to another machine. These states are entered as "unlocked", that is only confirmed messages are transmitted.

Message events

Messages pass from one state to another as a result of an event. These events are typically generated by an API call. The possible message events, as shown in Figure 4 on page 52, are:

`putMessage`
Places a message on a queue. This does not require a `confirmID`.

`getMessage`
Retrieves a message from a queue. This does not require a `confirmID`.

`putMessage with confirmId>0`
Places a message on a queue. This requires a `confirmID`. However, messages do not arrive at the receiving end in the order of sending, but in the order of confirmation.

`confirmPutMessage`
A confirm for an earlier `putMessage` with a `confirmID`>0.

`getMessage with confirmId>0`
Retrieves message from a queue. This requires a `confirmID`.

`confirmGetMessage`
A confirm for an earlier `getMessage` with a `confirmID`>0.

`browseWithLock`
Browses messages and lock those that match. Prevents messages from changing while browse is in operation.

unlockMessage
Unlocks a message locked with a `browsewithLock` command.

undo Unlocks a message locked with a browse, undoes a `getMessage` with a `confirmID`>0, or undoes a **putMessage** with a `confirmID`>0.

deleteMessage
Removes a message from a queue.

Message index fields

Due to memory size constraints, complete messages are not held in memory, but, to enable faster message searching, WebSphere MQ Everyplace holds specific fields from each message in a *message index*. The fields that are held in the index are:

Java In Java, the following fields are held in the index:

`UniqueID`
`MQe.Msg_OriginQMgr + MQe.Msg_Time`

`MessageID`
`MQe.Msg_ID`

`CorrelationID`
`MQe.Msg_CorrelID`

`Priority`
`MQe.Msg_Priority`

C In C, the following fields are held in the index:

`UniqueID`
MQE_MSG_ORIGIN_QMGR + MQE_MSG_TIME

`MessageID`
MQE_MSG_MSGID

`CorrelationID`
MQE_MSG_CORRELID

`Priority`
MQE_MSG_PRIORITY

Providing these fields in a filter makes searching more efficient, since WebSphere MQ Everyplace may not have to load all the available messages into memory.

Messaging operations

The following table shows which types of messaging operations are valid on local queues, synchronous remote queues, and asynchronous remote queues. Note that the `Listen` and `Wait` operations are supported in Java only.

Table 4. Messaging operations on WebSphere MQ Everyplace queues

Operation	Local queue	Synchronous remote queue	Asynchronous remote queue
Put	Yes	Yes	Yes
Get	Yes	Yes	No
Browse	Yes	Yes	No
Delete	Yes	Yes	No
Listen	Yes	No	No
Wait	Yes	Yes	No

Notes:

1. The synchronous remote wait operation is implemented through a poll of the remote queue, so the actual wait time is a multiple of the poll time
2. The WebSphere MQ bridge supplied with WebSphere MQ Everyplace only supports an assured or unassured put, unassured get, and unassured browse (without lock).

The following list describes each message operation in detail:

`Put` This operation places specified messages on a specified queue. The queue can belong to a local or remote queue manager. Puts to remote queues can occur immediately, or at a later time, depending on how the remote queue is defined on the local queue manager.

If a remote queue is defined as synchronous, message transmission occurs immediately. If a remote queue is defined as asynchronous, the message is stored within the local queue manager. The message remains there until it is transmitted. The put message call may finish before the message is put. Refer to Chapter 7, “Message Delivery”, on page 69 for more information.

Note: In Java, if the local queue manager does not hold a definition of the remote queue then it attempts to contact the queue sychronously. This does not apply to the C codebase.

Assured delivery depends on the value of the `confirmID` parameter. Passing a non-zero value transmits the message as normal, but the message is locked on the target queue until a subsequent confirm is received. Passing a value of zero transmits the message without the need for a subsequent confirm. However, message delivery is not assured. Refer to Chapter 7, “Message Delivery”, on page 69, for more information on assured and non-assured message delivery.

You can protect a message using message-level security. Refer to Chapter 8, “Security”, on page 79 for detailed information on message-level security.

`Get`

This operation returns an available message from a specified queue and removes the message from the queue. The queue can belong to a local or remote WebSphere MQ Everyplace queue manager, but cannot be an asynchronous remote queue.

If you do not specify a filter, the first available message is returned. If you do specify a filter, the first available message that matches the filter is returned. Including a valid `lockID` in the message filter allows you to get messages that have been locked by a previous browse operation. If no message is available, the get operation returns an error.

Using assured message delivery depends on the value of the `confirmID` parameter. Passing a non-zero value returns the message as normal. However, the message is locked and is not removed from the target queue until it receives a subsequent confirm. You can issue a confirm using the `confirmGetMessage()` method. However, message delivery is not assured. Refer to Chapter 7, “Message Delivery”, on page 69, for more information on assured and non-assured message delivery.

Delete

This method deletes a message from a queue. It does not return the message to the application that called it. You must specify the UniqueID and you can delete only one message per operation. The queue can belong to a local or synchronous remote WebSphere MQ Everyplace queue manager. Including a valid `lockID` in the message filter allows you to delete messages that have been locked by a previous operation, for example browse. If a message is not available, the application returns an error.

```
/* Example for deleting a message */
MQeFieldsHndl hMsg,hFilter;

/* create the new message */
rc = mqeFields_new(&exceptBlock, &hMsg);
if (MQERETURN_OK == rc) {

    /* add application fields here */
    /* ... */


    /* put message to a queue */
    rc = mqeQueueManager_putMessage(hQueueManager,
            &exceptBlock,
            hQMName,
            hQueueName, hMsg,
            NULL,0);
    if (MQERETURN_OK == rc) {
        /* Delete requires a filter -
    this can most easily be*/
   /* found from the UID fields of the message*/
        rc = mqeFieldsHelper_getMsgUidFields(hMsg,
            &exceptBlock,
            &hFilter);
    }

}


/* some time later want to delete the message  -
  use the esatblished filter */
rc = mqeQueueManager_deleteMessage(hQueueManager,
                                  &exceptBlock,
                                   hQMName,
                                   hQueueName,
                                   hFilter);
```

Browse

You can browse queues for messages using a filter, for example `message ID` or `priority` . Browsing retrieves all the messages that match the filter, but leaves them on the queue. The queue can belong to a local or remote queue manager. However, the implementation of the browse command is codebase specific.

WebSphere MQ Everyplace also supports *Browsing under lock*. This allows you to lock the matching messages on the queue. You can lock messages individually, or in groups identified through a filter, and the locking operation returns a `lockID`. Use the `lockID` to get or delete messages. An option on browse allows you to return either the full messages, or only the UniqueIDs.

```
 MQeVectorHndl hListMsgs;

 rc = mqeQueueManager_browseMessages(hQueueManager,
                                     &exceptBlock,
                                     &hListMsgs,
                                     hQMName,
                                     hQueueName,
                                     hFilter,
                                     NULL,MQE_FALSE);
if (MQERETURN_OK == rc) {
    /* process list using mqeVector_* apis */

    /* free off the vector */
    rc = mqeVector_free(hListMsgs,&exceptBlock);
}
```

Returning an entire collection of messsages can be expensive in terms of system resources. Setting the `justUID` parameter to true and returns the `uniqueID` of each message that matches the filter only.

The messages returned in the collection are still visible to other WebSphere MQ Everyplace APIs. Therefore, when performing subsequent operations on the messages contained in the enumeration, the application must be aware that another application can process these messages once the collection is returned. To prevent other applications from processing messages, use the `browseMessagesAndLock` method to lock messages contained in the enumeration.

`confirmPut`

This method performs the confirmation of a previously successful `putMessage()` operation.

`confirmGet`
This method confirms the successful receipt of a message retrieved from a queue manager by a previous `getMessage()` operation. The message remains locked on the target queue until it receives a confirm flow.

`Listen` Applications can listen for WebSphere MQ Everyplace message events, again with an optional filter. However, in order to do this, you must add a listener to a queue manager. Listeners are notified when messages arrive on a queue.

`Wait`

This method implements message polling. It allows you to specify a time for messages to arrive on a queue. Java implements a helper function for this. The C codebase, as it is non-threaded, must implement a function in application layer code. The following example demonstrates the `Wait` method:

Java Message polling uses the `waitForMessage()` method. This command issues a `getMessage()` command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application:

```
qmgr.waitForMessage("RemoteQMgr",
          "RemoteQueue",
          filter,
          null,
          0,
          60000);
```

The `waitForMessage()` method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds. Therefore, in the example, polling lasts for 6 seconds. This blocks the thread on which the command is running for 6 seconds, unless a message is returned earlier. Message polling works on both local and remote queues.

Note: Using this technique sends multiple requests over the network.

Queue Ordering

The order of messages on a queue is primarily determined by their priority. Message priority ranges from 9 (highest) to 0 (lowest). Messages with the same priority value are ordered by the time at which they arrive on the queue, with messages that have been on the queue for the longest being at the head of the priority group.

Reading messages on a queue

If you issue a `getMessage` command when a queue is empty, the queue throws a Java codebase `Except_Q_NoMatchingMsg` exception or returns a C codebase `MQERETURN_QUEUE_ERROR`, `MQEREASON_NO_MATCHING_MSG`. This allows you to create an application that reads all the available messages on a queue.

Java

Encasing the `getMessage()` call inside a `try..catch` block allows you to test the code of the resulting exception. This is done using the `code()` method of the MQeException class. You can compare the result from the `code()` method with a list of exception constants published by the WebSphere MQ Everyplace class. If the exception is not of type `Except_Q_NoMatchingMsg`, throw the exception again.

The following code shows this technique:

```
try
{
  while(true)
   { /* keep getting messages until
   an exception is thrown  */
    MQeMsgObject msg = qmgr.getMessage( "myQMgr", "myQueue",
              null, null, 0 );
    processMessage(msg);
   }
}
catch (Exception e)
{
   if ( e.code() != MQe.Except_Q_NoMatchingMsg )
    throw e;
}
```

Therefore, you can read all messages from a queue by iteratively getting messages until `MQe.Except_Q_NoMatchingMsg` is returned.

C

You can read all messages from a queue by looping, until the return code is `MQERETURN_QUEUE_WARNING` and the reason code is `MQEREASON_NO_MATCHING_MSG`.

Browse and Lock

Performing `BrowseAndLock` on a group of messages allows an application to ensure that no other application is able to process messages when they are locked. The messages remain locked until that application unlocks them. No other application can unlock the messages. Any messages that arrive on the queue after the `BrowseAndLock` operation are not locked.

An application can perform either a `get` or a `delete` operation on the messages to remove them from the queue. To do this, the application must supply the `lockID` that is returned with the enumeration of messages.

Specifying the `lockID` allows applications to work with locked messages without having to unlock them first.

Instead of removing the messages from the queue, it is also possible just to unlock them. This makes them visible once again to all WebSphere MQ Everyplace applications. You can achieve this by using the `unlockMessage` method.

Note: See the WebSphere MQ Everyplace Configuration Guide for special considerations with WebSphere MQ bridge queues.

The following examples demonstrate the use of `BrowseAndLock`:

Java example

The MQeEnumeration object contains all the messages that match the filter supplied to the browse. MQeEnumeration can be used in the same manner as the standard Java Enumeration. You can enumerate all the browsed messages as follows:

Note: You must supply a confirmID, in case the action of locating messages fails. It must be possible to undo the location, and this action requires the confirmID.

```
long confirmID = MQe.uniqueValue();
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null,
          "MyQueue",
          null, null,
                            confirmID, false);

while( msgEnum.hasMoreElements() )
{
   MQeMsgObject msg = (MQeMsgObject)msgEnum.nextElement();
   System.out.println( "Message from  queue manager: " +
                       msg.getAscii( MQe.Msg_OriginQMgr ) );
}
```

The following code performs a `delete` on all the messages returned in the enumeration. The message's `UniqueID` and `lockID` are used as the filter on the `delete` operation:

```
while(msgEnum.hasMoreElements())
{
   MQeMsgObject msg = (MQeMsgObject)
        msgEnum.getNextMessage(null,0);

   processMessage(msg);

   MQeFields filter = msg.getMsgUIDFields();
   filter.putLong(MQe.Msg_LockID,
       msgEnum.getLockId());

   qmgr.deleteMessage(null, "MyQueue", filter);
}
```

C example

The C codebase example gets the actual message. Note the additional parameters, a `confirmID` in case the operation needs undoing, and the `lockID`.

```
MQeVectorHndl hMessages;
MQEINT64 lockID, confirmID=42;
rc = mqeQueueManager_browseAndLock(hQueueManager,
              &exceptBlock,
              &hmessages,
              &lockID,
              hQueueManagerName,
              hQueueName,
              hFilter,
              NULL,   /*No Attribute*/
              confirmID,
              MQE_TRUE);  /*Just UIDs*/
/*process vector*/
MQeFieldsHndl hGetFilter;
rc = mqeFields_new(&exceptBlock, &hGetFilter);
if (MQERETURN_OK == rc){
 rc = mqeFields_putInt64(&hGetFilter,
           &exceptBlock,
           MQE_MSG_LOCKID,
           lockID);
  if (MQERETURN_OK == rc){
   rc = mqeQueueManager_getMessage(&hQueueManager,
                 &exceptBlock,
                 hQueueManagerName,
                 hQueueName,
              hGetFilter,
              &hMsg);
}
```

Message listeners

Note: This section does not apply to the C codebase.

WebSphere MQ Everyplace allows an application to *listen* for events occurring on queues. The application is able to specify message filters to identify the messages in which it is interested, as shown in the following Java example:

```
/* Create a filter for "Order" messages of priority 7 */
MQeFields filter = new MQeFields();
filter.putAscii( "MsgType", "Order" );
filter.putByte( MQe.Msg_Priority, (byte)7 );
/* activate a listener on "MyQueue"    */
qmgr.addMessageListener( this, "MyQueue", filter );
```

Listeners do not start automatically when you create a queue manager. A call to MQeAdministrator is required. However, listeners are persistent in the registry. This means that, once created, listeners that exist at queue manager start-up are started automatically.

The following parameters are passed to the **addMessageListener()** method:

- The name of the queue on which to listen for message operations
- A `callback` object that implements **MQeMessageListenerInterface**
- An MQeFields object containing a message filter

When a message arrives on a queue with a listener attached, the queue manager calls the `callback` object that it was given when the message listener was created.

The following is an example of the way in which an application would normally handle message events in Java:

```
public void messageArrived(MQeMessageEvent msgEvent)
 {
  String queueName =msgEvent.getQueueName();
  if (queueName.equals("MyQueue"))
  {
         try
         {
   /*get message from queue */
   MQeMsgObject msg =qmgr.getMessage(null,queueName,
      msgEvent.getMsgFields(),null,0);

   processMessage(msg );
         }
         catch (MQeException  e)
        {
        ...
        }
  }
 }
```

messageArrived() is a method implemented in **MQeMessageListenerInterface**. The `msgEvent` parameter contains information about the message, including:

- The name of the queue on which the message arrived
- The `UID` of the message
- The `messageID`
- The `correlationID`
- Message `priority`

Message filters only work on local queues. A separate technique known as polling allows messages to be obtained as soon as they arrive on remote queues.

Message polling

Note: This section does not apply to the C codebase.

Message polling uses the **waitForMessage()** method. This command issues a **getMessage()** command to the remote queue at regular intervals. As soon as a message that matches the supplied filter becomes available, it is returned to the calling application.

A wait for message call typically looks like this:

```
qmgr.waitForMessage( "RemoteQMgr", "RemoteQueue",
        filter, null, 0, 60000 );
```

The **waitForMessage()** method polls the remote queue for the length of time specified in its final parameter. The time is specified in milliseconds, so in the example above, the polling lasts for 60 seconds. The thread on which the command is executing is blocked for this length of time, unless a message is returned earlier.

Message polling works on both local and remote queues.

Note: Use of this technique results in multiple requests being sent over the network.

Trigger transmission

This method attempts to transmit pending messages. Only unlocked messages are transmitted.

Asynchronous remote queues and home server queues respond to trigger transmission processing. Put messages with no `confirmID` or put messages and confirm them before calling this method. Only messages that are fully put can be transmitted.

Trigger transmission rules

There are a number of rules, which can control the trigger transmission processing, if processing occurs. Chapter 3, Rules, or the WebSphere MQ Everyplace System Programming Guide provides detailed information on trigger transmission rules. Chapter 3, ″Rules″, of the WebSphere MQ Everyplace System Programming Guide contains information on trigger transmission rules.

```
rc = mqeQueueManager_triggerTransmission(hQueueManager,&exceptBlock);
```

Deleting queue managers

This section details how to delete a queue manager in Java and C.

Java

The basic steps required to delete a queue manager are:

1. Use the administration interface to delete any definitions
2. Create and activate an instance of `MQeQueueManagerConfigure`
3. Delete the standard queue and queue manager definitions
4. Close the `MQeQueueManagerConfigure` instance

When these steps are complete, the queue manager is deleted and can no longer be run. The queue definitions are deleted, but the queues themselves are not deleted. Any messages remaining on the queues are inaccessible.

Note: If there are messages on the queues they are not automatically deleted. Your application programs should include code to check for, and handle, remaining messages before deleting the queue manager.

1. Delete any definitions

You can use MQeQueueManagerConfigure to delete the standard queues that you created with it. You should use the administration interface to delete any other queues before you call MQeQueueManagerConfigure.

2. Create and activate an instance of MQeQueueManagerConfigure

This process is the same as when creating a queue manager.

3. Delete the standard queue and queue manager definitions

Delete the default queues by calling:

- **deleteAdminQueueDefinition()** to delete the administration queue
- **deleteAdminReplyQueueDefinition()** to delete the administration reply queue
- **deleteDeadLetterQueueDefinition()** to delete the dead letter queue
- **deleteSystemQueueDefinition()** to delete the default local queue

These methods work successfully even if the queues do not exist.

Delete the queue manager definition by calling **deleteQueueManagerDefinition()**

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteAdminQueueDefinition();
 qmConfig.deleteAdminReplyQueueDefinition();
 qmConfig.deleteDeadLetterQueueDefinition();
 qmConfig.deleteSystemQueueDefinition();
 qmConfig.deleteQueueManagerDefinition();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

You can delete the default queue and queue manager definitions together by calling **deleteStandardQMDefinitions()**. This method is provided for convenience and is equivalent to:

```
deleteDeadLetterQueueDefinition();
deleteSystemQueueDefinition();
deleteAdminQueueDefinition();
deleteAdminReplyQueueDefinition();
deleteQueueManagerDefinition();
```

4. Close the MQeQueueManagerConfigure instance

When you have deleted the queue and queue manager definitions, you can **close** the MQeQueueManagerConfigure instance.

The complete example looks like this:

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

C

The steps in deleting a queue manager are essentially the same between the :

1. Remove all Connection Definitions.
2. Remove all Queues, including any ″system″ queues, for example the dead letter queue. Ensure all queues are empty.
3. Remove the queue manager.

You require an administrator to perform these functions. We also recommend stopping the queue manager first.

Note: Deleting the queue mananger will free the queue manager handle for you. MQeAdministratorHndl hAdmin:

```
/* Create the new administrator based on the exisitng QM Handle */
rc = mqeAdministrator_new(&exceptBlock,
        &hAdmin,hQueueManager);
if (MQERETURN_OK == rc) {

    if (MQERETURN_OK == rc) {
        /* delete any conncetion definitins for example :*/
        rc = mqeAdministrator_Connection_delete(hAdmin,
                &exceptBlock,
                hRemoteQM);
    }

    /* delete all the local queues here - remember to do "special*/
  /*queues" for example ... */
    if (MQERETURN_OK == rc) {
        rc = mqeAdministrator_LocalQueue_delete(hAdmin,
                &exceptBlock,
                MQE_DEADLETTER_QUEUE_NAME,
                hLocalQMName);
    }

    /* Finally delete the queue manager */
    if (MQERETURN_OK == rc) {
        rc = mqeAdministrator_QueueManager_delete(hAdmin,
                 &exceptBlock);
    }
    /* free of the amdinsitrator */
    (void)mqeAdministrator_free(hAdmin, &exceptBlock);
}
```

Servlet

As well as running as a standalone server, a queue manager can be encapsulated in a servlet to run inside a Web server . A servlet queue manager has nearly the same capabilities as a server queue manager. MQeServlet provides an example implementation of a servlet. As with the server, servlets use ini files to hold start up parameters. A servlet uses many of the same WebSphere MQ Everyplace components as the server.

The main component not required in a servlet is the connection listener, this function is handled by the Web server itself. Web servers only handle http data streams so any WebSphere MQ Everyplace client that wishes to communicate with an WebSphere MQ Everyplace servlet must use the http adapter (com.ibm.mqe.adapters.MQeTcpipHttpAdaper). When you configure connections to queue managers running in servlets, you must specify the name of the servlet in the parameters field of the connection. The following definitions configure a connection on servlet /servlet/MQe with queue manager `PayrollQM`:

Connection name
    `PayrollQM`

Channel
    `com.ibm.mqe.communications.MQeChannel`

Note: The com.ibm.mqe.MQeChannel class has been moved and is now known as com.ibm.mqe.communications.MQeChannel. Any references to the old class name in administration messages is replaced automatically with the new class name.

Channel Adapter
    `com.ibm.mqe.adapters.MQeTcpipAdapter:192.168.0.10:80`

Parameters
    `/servlet/MQe`

Options

Alternatively, if the relevant aliases have been set up, you can configure the connection as follows:

Connection name
    `PayrollQM`

Channel
    `DefaultChannel`

Adapter
    `Network:192.168.0.10:80`

Parameters
    `/servlet/MQe`

Options

Web servers can run multiple servlets. It is possible to run multiple different WebSphere MQ Everyplace servlets within a Web server, with the following restrictions:

- Each servlet must have a unique name
- Only one queue manager is allowed per servlet
- Each WebSphere MQ Everyplace servlet must run in a different Java Virtual Machine (JVM)

The WebSphere MQ Everyplace servlet extends javax.servlet.http.HttpServlet and overrides methods for starting, stopping and handling new requests. The following code fragment starts a servlet:

```
/**
 * Servlet initialization......
 */
public void init(ServletConfig sc) throws ServletException
{
  // Ensure supers constructor is called.
  super.init(sc);

  try
  {
    // Get the the server startup ini file
    String startupIni;
    if ((startupIni = getInitParameter("Startup")) == null)
      startupIni = defaultStartupInifile;

    // Load it
    MQeFields sections = MQeQueueManagerUtils.loadConfigFile(startupIni);

    // assign any class aliases
    MQeQueueManagerUtils.processAlias(sections);

    // Uncomment the following line to start trace before the queue
    // manager is started
    //      MQeQueueManagerUtils.traceOn("MQeServlet Trace", null);

    // Start connection manager
    channelManager = MQeQueueManagerUtils.processChannelManager(sections);

    // check for any pre-loaded classes
    loadTable = MQeQueueManagerUtils.processPreLoad(sections);

    // setup and activate the queue manager
    queueManager = MQeQueueManagerUtils.processQueueManager(sections,
    channelManager.getGlobalHashtable( ));

    // Start ChannelTimer  (convert time-out from secs to millisecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt
                   ("TimeInterval");
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer(channelManager, timeInterval);

    // Servlet initialization complete
    mqe.trace(1300, null);
  }
  catch (Exception e)
  {
    mqe.trace(1301, e.toString());
    throw new ServletException(e.toString());
  }
}
```

The main differences compared to a server startup are:

- The servlet overrides the **init** method of the superclass. This method is called by the Web server to start the servlet. Typically this occurs when the first request for the servlet arrives.
- The name of the startup ini file cannot be passed in from the command line. The example expects to obtain the name using the servlet method **getInitParameter()** which takes the name of a parameter and returns a value. The WebSphere MQ Everyplace servlet uses a *`Startup`* parameter that it expects to contain an ini file name. The mechanism for configuring parameters in a Web server is Web server dependant.
- A listener is not started as the Web server handles all network requests on behalf of the servlet.
- As there is no listener a mechanism is required to time-out connections that have been inactive for longer than the time-out period. A simple timer class MQeChannelTimer is instantiated to perform this function. The *`TimeInterval`* value is the only parameter used from the `[Listener]` section of the ini file.

A servlet relies on the Web server for accepting and handling incoming requests. Once the Web server has decided that the request is for an WebSphere MQ Everyplace servlet, it passes the request to WebSphere MQ Everyplace using the **doPost()** method. The following code handles this request:

```
/**
 * Handle POST......
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
       throws IOException
{
  // any request to process ?
  if (request == null)
    throw new IOException("Invalid request");
  try
  {
    int max_length_of_data = request.getContentLength();
  // data length
    byte[] httpInData = new byte[max_length_of_data];
  // allocate data area
    ServletOutputStream httpOut = response.getOutputStream();
  // output stream
    ServletInputStream  httpIn  = request.getInputStream();
  // input stream

    // get the request
    read( httpIn, httpInData, max_length_of_data);

    // process the request
    byte[] httpOutData = channelManager.process(null, httpInData);

    // appears to be an error in that content-
   length is not being set
    // so we will set it here
    response.setContentLength(httpOutData.length);
    response.setIntHeader("content-length", httpOutData.length);

    // Pass back the response
    httpOut.write(httpOutData);
  }
  catch (Exception e)
  {
    // pass it on ...
    throw new IOException( "Request failed" + e );
  }
}
```

This method:

1. Reads the http input data stream into a *`byte array`*. The input data stream may be buffered so the **read()** method is used to ensure that the entire data stream is read before continuing.

   Note: WebSphere MQ Everyplace only handles requests with the **doPost()** method, it does not accept requests using the **doGet()** method
2. The request is passed to WebSphere MQ Everyplace through a connection manager. From this point, all processing of the request is handled by core WebSphere MQ Everyplace classes such as the queue manager.
3. Once WebSphere MQ Everyplace has completed processing the request, it returns the result wrapped in http headers as a byte array. The byte array is passed to the Web server and is transmitted back to the client that originated the request.

Security

The queue manager fully supports the security functions supplied with WebSphere MQ Everyplace. Any messages stored in a queue defined with security characteristics are encoded using those characteristics. Any connections set up between a queue manager and a secure queue use the security characteristics of the queue, or an existing connection with equal or higher security.

Messages can be individually protected by attaching security characteristics to them directly. The correct characteristics must be presented whenever dealing with a message protected in this manner.

Connection security

Because of the way channel security works, when a specific attribute rule is specified for a target queue, it forces the local queue manager to create an instance of the same attribute rule, `examples.rules.AttributeRule` and `com.ibm.mqe.MQeAttributeRule` are treated as the same rule. If this is not a desirable behaviour, you can specify a null rule for the target queue. In this case, com.ibm.mqe.`MQeAttributeDefaultRule` takes effect.

See Chapter 8, “Security”, on page 79 for a detailed discussion of WebSphere MQ Everyplace security.

Aliases

WebSphere MQ Everyplace provides two forms of aliasing, queue manager aliasing and queue aliasing. Both provide a level of indirection between the applications logical view of an item and the real item. The WebSphere MQ Everyplace System Programming Guide provides detailed information on using queue and queue manager aliasing.

Chapter 7. Message Delivery

WebSphere MQ Everyplace networks are composed of connected queue managers and can include gateways. They can span multiple physical networks and route messages between them. In general they provide synchronous and asynchronous access to queues with a programming model that is independent of queue location.

This chapter describes different types of the message delivery process in detail, under the following headings:

- Asynchronous message delivery
- Synchronous message delivery
- Assured and Non-assured message delivery
- Synchronous assured message delivery

Asynchronous message delivery

An asynchronous put to a remote queue places the message on the backing store associated with the local definition of that queue, along with its destination queue manager name, queue name, and the compressor, authenticator, and cryptor characteristics that match the target destination of the message. The message's dump method is called as it is saved to persistent storage in a secure format that is defined by its destination queue. The queue manager controls message delivery. It identifies or establishes a connection with appropriate characteristics to the queue manager for the next hop, then creates or reuses a transporter to the target queue manager. The transporter dumps the message and transmits the resulting byte string. The target queue manager and queue name are not part of that message flow.

If appropriate, the message is encrypted and compressed over the connection. If it has reached its destination queue manager, it is decrypted and decompressed. A new message is created, using the restore method, and the resultant message is placed on the destination queue. If the message has not reached its destination queue manager, it is decrypted and decompressed. It is then re-encrypted, compressed, and placed on a store-and-forward queue for onward transmission, if a store-and-forward queue exists. In both cases it is held on its respective queue in a secure format, as defined by its destination queue.

A characteristic of asynchronous message delivery is that messages are passed to the queue manager at intermediate hops, being queued for onward transmission. Messages are taken off the intermediate queues first in order of priority, then in order of arrival on the queue. Duplicate messages, created when you resend a message, are also taken off the intermediate queues in the order of their arrival on the queue.

Synchronous message delivery

Synchronous message delivery is similar to the asynchronous case described above, but the queue manager involvement in intermediate hops takes place at a much lower level, involving the transporter and connections. An end-to-end connection is established, using the adapters defined in the protocol specifications at each intermediate node, to identify the next link. At the end of the last link, where no further relevant file descriptors exist, the message gets passed to the higher layers of the queue manager for processing. Thus the sending node does not queue the message but passes it along the connection, through intermediate hops, and then gives it to the destination queue manager to place it on the target queue.

The link into WebSphere MQ uses a bridge queue on the gateway, which transforms the message into a WebSphere MQ format. This mechanism means that synchronous WebSphere MQ Everyplace style messaging from a device is possible to WebSphere MQ, with the connection terminating at the gateway. The message is delivered in real time from the gateway, through a client channel, to a WebSphere MQ server. From there its destination can require it to be routed asynchronously along WebSphere MQ message channels.

In a similar manner, a device capable of only synchronous messaging can send messages to an asynchronous WebSphere MQ Everyplace queue, provided that a suitable intermediary is available.

Assured and non-assured message delivery

Message delivery using synchronous message transmission can be assured or non-assured.

Assured message delivery

Asynchronous transmission introduces the concept of *assured message delivery*. When delivering messages asynchronously, WebSphere MQ Everyplace delivers each message once, and once-only, to its destination queue. However, this assurance is only valid if the definition of the remote queue and remote queue manager match the current characteristics of the remote queue and remote queue manager. If a remote queue definition and the remote queue do not match, then it is possible that a message may become undeliverable. In this case the message is not lost, but remains stored on the local queue manager.

Non-assured message delivery

Non-assured delivery of a message takes place in a single network flow. The queue manager sending the message creates or reuses a channel to the destination queue manager.

The message to be sent is dumped to create a byte-stream, and this byte stream is given to the channel for transmission. Once program control has returned from the channel the sender queue manager knows that the message has been successfully given to the target queue manager, that the target has logged the message on a queue, and that the message has been made visible to WebSphere MQ Everyplace applications.

However, a problem can occur if the sender receives an exception over the channel from the target. The sender has no way of knowing if the exception occurred before or after the message was logged and made visible. If the exception occurred before the message was made visible it is safe for the sender to send the message again. However, if the exception occurred after the message was made visible, there is a danger of introducing duplicate messages into the system since an WebSphere MQ Everyplace application could have processed the message before it was sent the second time.

The solution to this problem involves transmitting an additional confirmation flow. If the sender application receives a successful response to this flow, then it knows that the message has been delivered once and once-only.

Synchronous assured message delivery

Put message

You can perform assured message delivery using synchronous message transmission, but the application must take responsibility for error handling.

The `confirmID` parameter of the `putMessage` method dictates whether a confirm flow is expected or not. A value of `zero` means that message transmission occurs in one flow, while a value of greater than zero means that a confirm flow is expected. The target queue manager logs the message to the destination queue as usual, but the message is locked and invisible to WebSphere MQ Everyplace applications, until a confirm flow is received. When you put messages with the `confirmID`, the messages are ordered by confirm time, not arrival time.

An WebSphere MQ Everyplace application can issue a `put` message confirmation using the `confirmPutMessage` method. Once the target queue manager receives the flow generated by this command, it unlocks the message, and makes it visible to WebSphere MQ Everyplace applications. You can confirm only one message at a time. It is not possible to confirm a batch of messages.

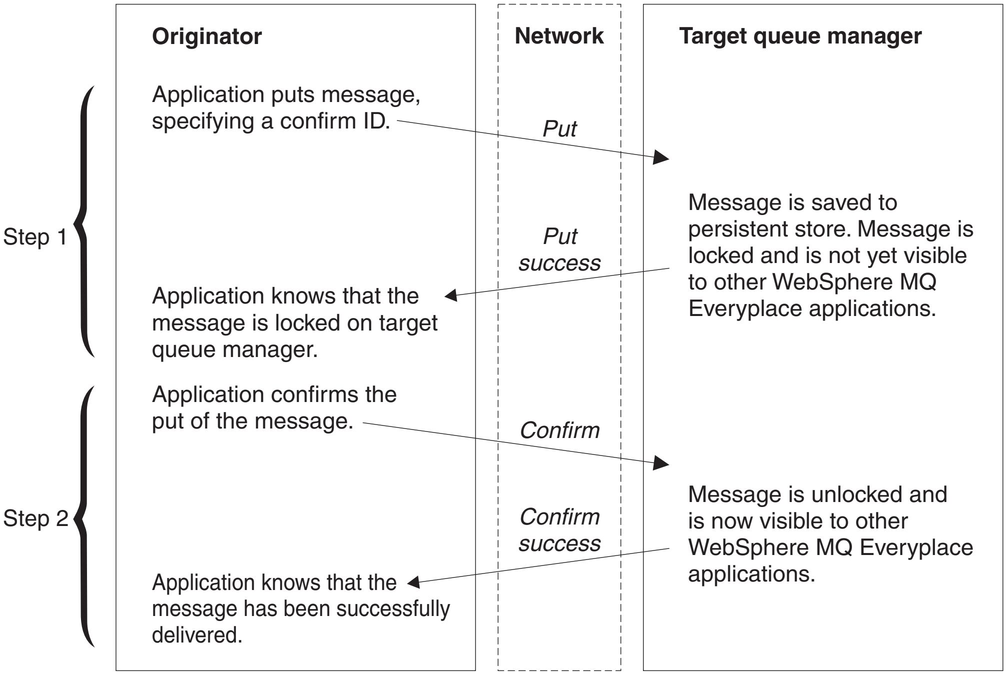

Originator
Network
Target queue manager
Step 1
Application puts message, specifying a confirm ID.
Put
Message is saved to persistent store. Message is locked and is not yet visible to other WebSphere MQ Everyplace applications.
Put success
Application knows that the message is locked on target queue manager.
Step 2
Application confirms the put of the message.
Confirm
Message is unlocked and is now visible to other WebSphere MQ Everyplace applications.
Confirm success
Application knows that the message has been successfully delivered.

Figure 5. Assured put of synchronous messages

The `confirmPutMessage()` method requires you to specify the `UniqueID` of the message, not the `confirmID` used in the prior put message command. The `confirmID` is used to restore messages that remain locked after a transmission failure. This is explained in detail on page 77.

A skeleton version of the code required for an assured `put` is shown below:

Java codebase

```
long confirmId = MQe.uniqueValue();

try
{
  qmgr.putMessage( "RemoteQMgr", "RemoteQueue",
       msg, null, confirmId );
}
catch( Exception e )
{
  /* handle any exceptions*/
}

try
{
   qmgr.confirmPutMessage( "RemoteQMgr", "RemoteQueue",
                         msg.getMsgUIDFields() );
}
catch ( Exception e )
{
  /* handle any exceptions   */
}
```

C codebase

```
/* generate confirm Id */
MQEINT64 confirmId;
rc = mqe_uniqueValue(&exceptBlock,
       &confirmId);

/* put message to queue using this confirm Id */
if(MQERETURN_OK == rc) {
    rc = mqeQueueManager_putMessage(hQMgr,
          &exceptBlock,
          hQMgrName, hQName,
          hMsg, NULL, confirmId);
    /* now confirm the message put */
    if(MQERETURN_OK == rc) {
         /* first get the message uid fields */
         MQeFieldsHndl hFilter;
         rc = mqeFieldsHelper_getMsgUidFields(hMsg,
             &exceptBlock,
             &hFilter);
         if(MQERETURN_OK == rc) {
            rc = mqeQueueManager_confirmPutMessage(hQMgr,
             &exceptBlock,
             hQMgrName,
             hQName, hFilter);
         }
    }
}
```

If a failure occurs during step 1 in Figure 5 on page 71 the application should retransmit the message. There is no danger of introducing duplicate messages into the WebSphere MQ Everyplace network since the message at the target queue manager is not made visible to applications until the confirm flow has been successfully processed.

If the WebSphere MQ Everyplace application retransmits the message, it should also inform the target queue manager that this is happening. The target queue manager deletes any duplicate copy of the message that it already has. The application sets the `MQe.Msg_Resend` field to do this.

If a failure occurs during step 2 in Figure 5 on page 71 the application should send the confirm flow again. There is no danger in doing this since the target queue manager ignores any confirm flows it receives for messages that it has already confirmed. This is shown in the following example, taken from examples.application.example6.

Java codebase

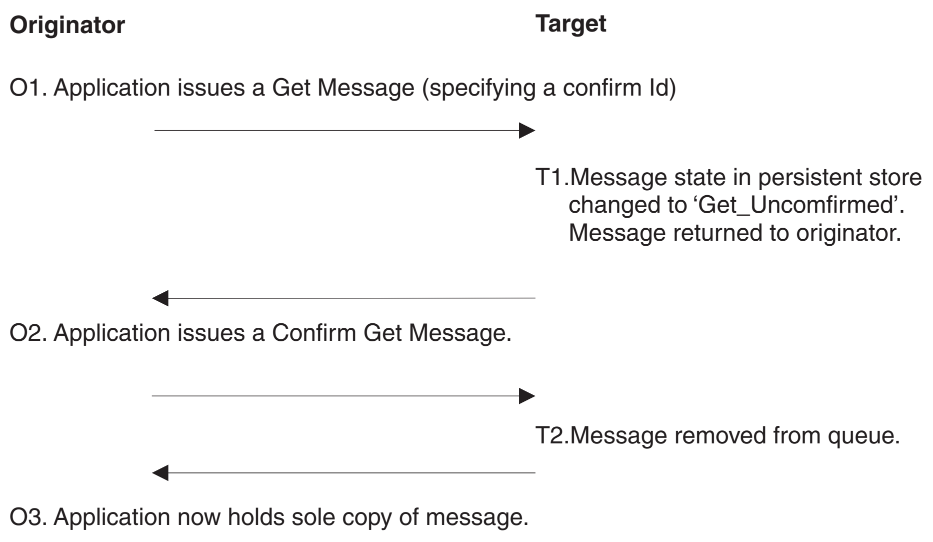
```
boolean msgPut     = false;
 /* put successful?  */
boolean msgConfirm = false;
 /* confirm successful?  */
int maxRetry       = 5;
 /* maximum number of retries  */

long confirmId = MQe.uniqueValue();

int retry = 0;
while( !msgPut &&
    retry < maxRetry )
{
  try
  {
    qmgr.putMessage( "RemoteQMgr",
       "RemoteQueue",
       msg, null,
       confirmId );
   msgPut = true;
/* message put successful             */
  }
  catch( Exception e )
  {
    /* handle any exceptions */
    /* set resend flag for
  retransmission of message */
    msg.putBoolean( MQe.Msg_Resend, true );
    retry ++;
  }
}

if ( !msgPut )
 /* was put message successful?*/
    /* Number of retries has
  exceeded the maximum allowed,
  /*so abort the put*/
   /* message attempt */
 return;

retry = 0;
while( !msgConfirm &&
     retry < maxRetry )
{
  try
  {
    qmgr.confirmPutMessage( "RenoteQMgr",
        "RemoteQueue",
                         msg.getMsgUIDFields());
    msgConfirm = true;
/* message confirm successful*/
  }
  catch ( Exception e )
  {
    /* handle any exceptions*/
    /* An Except_NotFound
  exception means */
 /*that the message has already   */
    /* been confirmed */
    if ( e instanceof MQeException &&
        ((MQeException)e).code() == Except_NotFound )
      putConfirmed = true;
  /* confirm successful */
    /* another type of exception -
  need to reconfirm message */
    retry ++;
  }
}
```

C codebase

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_putMessage(hQMgr,
          &exceptBlock,
          hQMgrName,
          hQName, hMsg,
          NULL, confirmId);

/* if the put attempt fails,
  retry up to the maximum number*/
/*of retry times permitted,
  setting the re-send flag. */
while (MQERETURN_OK != rc
     && --maxRetry > 0 ) {
    rc = mqeFields_putBoolean(hMsg, &exceptBlock,
           MQE_MSG_RESEND, MQE_TRUE);
    if(MQERETURN_OK == rc) {
       rc = mqeQueueManager_putMessage(hQMgr, &exceptBlock,
              hQMgrName, hQName,
              hMsg, NULL, confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
            &exceptBlock,
            &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmPutMessage(hQMgr,
             &exceptBlock,
             hQMgrName, hQName,
             hFilter);
    }
     while (MQERETURN_OK != rc
        && --maxRetry > 0 ) {
             rc = mqeQueueManager_confirmPutMessage(hQMgr,
                  &exceptBlock,
                   hQMgrName,
                   hQName,
                   hFilter);
    }
}
```

Get message

Assured message `get` works in a similar way to `put`. If a `get` message command is issued with a `confirmId` parameter greater than zero, the message is left locked on the queue on which it resides until a confirm flow is processed by the target queue manager. When a confirm flow is received, the message is deleted from the queue. Figure 6 on page 75 describes a get of synchronous messages:



O1. Application issues a Get Message (specifying a confirm Id)

T1.Message state in persistent store changed to 'Get_Uncomfirmed'. Message returned to originator.

O2. Application issues a Confirm Get Message.

T2.Message removed from queue.

O3. Application now holds sole copy of message.

Figure 6. Assured get of synchronous messages

The following code is taken from `examples.application.example6`

Java codebase

```
boolean msgGet     = false;
/* get successful?  */
boolean msgConfirm = false;
/* confirm successful?  */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries  */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry)
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
          "RemoteQueue",
          filter, null,
                            confirmId );
    msgGet = true;
 /* get succeeded  */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    /* if the exception is of type
   Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable
   then throw the exception */

    if ( e instanceof MQeException )
      if ( ((MQeException)e).code() ==
        Except_Q_NoMatchingMsg )
        throw e;
    retry ++;
 /* increment retry count  */
  }
}

if ( !msgGet )
 /* was the get successful?       */
   /* Number of retry attempts has
  exceeded the maximum allowed, so abort  */
   /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
          "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
 /* confirm succeeded  */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    retry ++;  /* increment retry count */
  }
}
```

C codebase

```
MQEINT32 maxRetry = 5;

rc = mqeQueueManager_getMessage(hQMgr,
            &exceptBlock,
             hQMgrName,
             hQName, hMsg,
             NULL, confirmId);

/* if the get attempt fails, retry
  up to the maximum number of*/
/*retry times permitted,
 setting the re-send flag. */
while (MQERETURN_OK != rc  &&
            --maxRetry > 0 ) {
    rc = mqeFields_getBoolean(hMsg,
          &exceptBlock,
           MQE_MSG_RESEND,
           MQE_TRUE);
    if(MQERETURN_OK == rc) {
        rc = mqeQueueManager_getMessage(hQMgr,
              &exceptBlock,
               hQMgrName,
               hQName, hMsg,
               NULL,
               confirmId);
    }
}

if(MQERETURN_OK == rc) {
    MQeFieldsHndl hFilter;
    maxRetry = 5;
    rc = mqeFieldsHelper_getMsgUidFields(hMsg,
              &exceptBlock,
              &hFilter);
    if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_confirmGetMessage(hQMgr,
                &exceptBlock,
                 hQMgrName,
                 hQName,
                 hFilter);
    }
     while (MQERETURN_OK != rc  &&
             --maxRetry > 0 ) {
              rc = mqeQueueManager_confirmPutMessage(hQMgr,
                  &exceptBlock,
                   hQMgrName,
                   hQName,
                   hFilter);
    }
}
```

The value passed as the `confirmId` parameter also has another use. The value is used to identify the message while it is locked and awaiting confirmation. If an error occurs during a `get` operation, it can potentially leave the message locked on the queue. This happens if the message is locked in response to the `get` command, but an error occurs before the application receives the message. If the application reissues the `get` in response to the exception, then it will be unable to obtain the same message because it is locked and invisible to WebSphere MQ Everyplace applications.

However, the application that issued the `get` command can restore the messages using the `undo` method. The application must supply the `confirmId` value that it supplied to the `get` message command. The `undo` command restores messages to the state they were in before the `get` command.

Java codebase

```
boolean msgGet     = false;
/* get successful?  */
boolean msgConfirm = false;
/* confirm successful?  */
MQeMsgObject msg   = null;
int maxRetry       = 5;
/* maximum number of retries  */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr",
          "RemoteQueue",
          filter, null,
                           confirmId );
    msgGet = true;
 /* get succeeded  */
  }
  catch ( Exception e )
  {
    /* handle any exceptions   */
    /* if the exception is of type
   Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable
   then throw the exception */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code() == Except_Q_NoMatchingMsg )
        throw e;
    retry ++;  /* increment retry count  */
    /* As a precaution, undo the message
   on the queue. This will remove  */
    /* any lock that may have been put on
   the message prior to the       */
    /* exception occurring   */
    myQM.undo( qMgrName, queueName, confirmId );
  }
}

if ( !msgGet )
```

```
  /* was the get successful?       */
     /* Number of retry attempts has
   exceeded the maximum allowed, so abort  */
     /* get message operation   */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr",
          "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;
 /* confirm succeeded  */
  }
  catch ( Exception e )
  {
    /* handle any exceptions     */
    retry ++;
 /* increment retry count  */
  }
}
```

C codebase

```
MQeFieldsHndl hMsg;
rc = mqeQueueManager_getMessage(hQMgr, &exceptBlock,
           &hMsg, hQMgrName,
            hQName, hFilter,
            NULL, confirmId);
/* if unsuccessful, undo the operation */
if(MQERETURN_OK != rc) {
    rc = mqeQueueManager_undo(hQMgr, &exceptBlock,
           hQMgrName, hQName,
           confirmId);
}
```

The `undo` command also has relevance for the `putMessage` and `browseMessagesAndLock` commands. As with `get` message, the `undo` command restores any messages locked by the `browseMessagesandLock` command to their previous state.

If an application issues an `undo` command after a failed `putMessage` command, then any message locked on the target queue awaiting confirmation is deleted.

The `undo` command works for operations on both local and remote queues.

Chapter 8. Security

WebSphere MQ Everyplace provides an integrated set of security features that enable the protection of data when held locally and when it is being transferred. There are three different categories of security:

Local security
Local security provides protection for any WebSphere MQ Everyplace data.

Queue-based security
Queue-based security automatically protects WebSphere MQ Everyplace message data between the initiating queue manager and queue, on the queue, and between the queue and the receiving queue manager. This protection is independent of whether the target queue is owned by a local or a remote queue manager. Using queue-based security does not require any change to application code and, therefore, is not described any further in this chapter. The WebSphere MQ Everyplace Configuration Guide describes how to add security attributes to a queue.

Message-level security
Message-level security provides protection for message data between an initiating and receiving WebSphere MQ Everyplace application.

Note: Throughout the world there are varying government regulations concerning levels and types of cryptography. You must always use a level and type of cryptography that complies with the appropriate local legislation. This is particularly relevant when using a mobile device that is moved from country to country. WebSphere MQ Everyplace provides facilities for this, but it is the responsibility of the application programmer to implement it.

In this chapter, security is explained under the following headings:

- Security features
- Local security
- Message-level security
- Mini-certificate issuance service
- Private registry service
- Public registry service

Security features

Queue based security is handled internally by WebSphere MQ Everyplace and does not require any specific action by the initiator or recipient of the message. Local and Message-level security must be initiated by an application.

All three categories protect Message data by the application of an MQeAttribute , or a descendent. Depending on the category, the attribute is either explicitly or implicitly applied.

Every attribute can contain any or all of the following objects:

- Authenticator
- Cryptor
- Compressor
- Key

- Target Entity Name

The way these objects are used depends on the category of WebSphere MQ Everyplace security. Each category of security is described in detail later in this chapter.

WebSphere MQ Everyplace also provides the following services to assist with security:

Private registry services
WebSphere MQ Everyplace private registry provides a repository in which public and private objects can be stored. It provides (login) PIN protected access so that access to a private registry is restricted to the authorized user. It also provides additional services so that functions can use the entity's private key, (for digital signature, and RSA decryption) without the private credentials leaving the PrivateRegistry instance.

These services are used by queue-based security and message-level security using MQeTrustAttribute.

Public registry services
WebSphere MQ Everyplace public registry provides a publicly accessible repository for mini-certificates.

These services can be used by queue-based and message-level security.

Mini-certificate issuance service
WebSphere MQ Everyplace provides SupportPac ES03, ″WebSphere MQ Everyplace WTLS Mini-Certificate Server″, which includes a default *mini-certificate issuance service* that you can configure to issue mini-certificates to a carefully controlled set of entity names.

These services can be used by queue-based and message-level security.

These services are described in more detail later in the chapter.

Local security

Local security protects WebSphere MQ Everyplace message or MQeFields data locally. This is achieved by creating an attribute with an appropriate symmetric cryptor and compressor, creating and setting up an appropriate *key*, by providing a password. The key is explicitly attached to the attribute, and the attribute is attached to the WebSphere MQ Everyplace message. WebSphere MQ Everyplace provides the MQeLocalSecure Java class and C API to assist with the setup of local security, but in all cases it is the responsibility of the local security user (WebSphere MQ Everyplace internally or a WebSphere MQ Everyplace application) to set up an appropriate attribute and manage the password key.

Local security provides protection for WebSphere MQ Everyplace data, MQeFields objects, including Java message objects, for example MQeMsgObject. The protected data is returned in a byte array. To apply local security to a data object you must:

1. Create an attribute with an appropriate authenticator, cryptor, and compressor.
2. Set up an appropriate *key*, by providing a password.
3. Explicitly attach the key to the attribute, the attribute to the data, MQeFields object, and invoke the **dump()** method on the data object.

The authenticator determines how access to the data is controlled. It is invoked every time a piece of data is acessed. The cryptor determines the cryptographic strength protecting the data confidentiality. The compressor determines the amount of storage required by the message.

WebSphere MQ Everyplace provides the MQeLocalSecure class to assist with the use of local security. However, it is the responsibility of the local security user to setup an appropriate attribute and provide the password. MQeLocalSecure provides the function to protect the data and to save and restore it from backing storage. If an application chooses to attach an attribute to a message without using MQeLocalSecure, it also needs to save the data after using **dump** and must retrieve the data before using **restore**.

Usage scenario

Consider a scenario where mobile agents working on many different customer sites want to ensure that the confidential data of one customer is not accidentally shared with another. Local security features, using different keys, and possibly different cryptographic strengths, provide a simple method for protecting different customer data held on a single machine .

A simple extension of this scenario could be that the protected local data is accessed using a key that is pulled from a secure queue on an WebSphere MQ Everyplace server node. The agents client has to authenticate itself to access the server queue and pull the local key data, but never knows the actual key.

One of the advantages of taking this approach is that an audit trail is easily accumulated for all access to customer specific data.

Secure feature choices

When using local security, WebSphere MQ Everyplace provides attribute choices for authentication, encryption, and compression. The algorithms supported by WebSphere MQ Everyplace for authentication, encryption, and compression are listed in Table 5.

Table 5. Authentication, encryption and compression support

Function	Algorithm
Authentication	WTLS mini-certificate (NTAuthenticator or UserIdAuthenticator, Java only)
	Validation Windows NT, Windows 2000, AIX, or Solaris identity
	WinCEAuthenticator (C only)
Compression	LZW (Java only)
	RLE (Java and C)
	GZIP (Java only)
Encryption	Triple DES (Java only)
	DES (Java only)
	MARS (Java only)
	RC4 (Java and C)
	RC6 (Java only)
	XOR (Java only)

You can use your own implementations of authenticators, provided that your cryptor is symmetric.

Selection criteria

You should use an authenticator if you need to provide additional controls to prevent access to the local data by unauthorized users. In some ways using an authenticator is unnecessary since providing the key password automatically limits access to those who know this secret.

Queue-based security, uses mini-certificate based mutual authentication, and message-level protection.

The choice of cryptor is driven by the strength of protection required. The stronger the encryption, the more difficulty an attacker would face when trying to get illegal access to the data. Data protected with symmetric ciphers that use 128 bit keys is acknowledged as more difficult to attack than data protected using ciphers that use shorter keys. However, in addition to cryptographic strength, the selection of a cryptor may also be driven by many other factors. An example is that some financial solutions require the use of triple DES in order to get audit approval.

You should use a compressor if you need to optimize the size of the protected data. However, the effectiveness of the compressor depends on the content of the data. The Java MQeRleCompressor and the C MQE_RLE_COMPRESSOR perform run length encoding. This means that the compressor routines compress or expand repeated bytes. Hence it is effective in compressing and decompressing data with many repeated bytes. MQeLZWCompressor uses the LZW scheme. The simplest form of the LZW algorithm uses a dictionary data structure in which various words, or data patterns, are stored against different codes. This compressor is likely to be most effective where the data has a significant number of repeating words, or data patterns. The MQeGZIPCompressor uses the same compression algorithm as the **gzip** command on UNIX. This searches for repeating patterns in the data and replaces subsequent occurrences of a pattern with a reference back to the first occurrence of the pattern.

Usage guide for Java

1. The following code protects an MQeFields object using MQeLocalSecure

```
 try
{
 .../* SIMPLE UNPROTECT FRAGMENT */
 .../* instantiate a DES cryptor */
 MQeDESCryptor desC = new MQeDESCryptor( );
 .../* instantiate an attribute using the DES cryptor */
 MQeAttribute desA = new MQeAttribute( null, desC, null);
 .../* instantiate a (a helper) LocalSecure object */
 MQeLocalSecure ls = new MQeLocalSecure( );
.../* open LocalSecure obj
  identifying target file and directory */
ls.open( ".\\", "TestSecureData.txt" );
/*instantiate a MQeFields object */
MQeFields myData =new MQeFields();
/*add some test data */
myData.putAscii("testdata","0123456789abcdef....");
.../* use LocalSecure write to protect data*/
ls.write( myData.dump(), desA, "It_is_a_secret" ) );
...
}
catch ( Exception e )
{
e.printStackTrace(); /* show exception */
```

```
}

try
    {
    .../* SIMPLE UNPROTECT FRAGMENT     */
    .../* instantiate a DES cryptor     */
    MQeDESCryptor des2C  = new MQeDESCryptor( );
    .../* instantiate an attribute using the DES cryptor  */
    MQeAttribute  des2A  = new MQeAttribute( null, des2C, null);
    .../* instantiate a (a helper) LocalSecure object */
    MQeLocalSecure ls2   = new MQeLocalSecure( );
    .../* open LocalSecure obj identifying
      target file and directory */
    ls2.open( ".\\", "TestSecureData.txt" );
    .../* use LocalSecure read to restore
      from target and decode data*/
    String outData       = MQe.byteToAscii( ls2.read( desA2,
                                          "It_is_a_secret"));
    .../* show results....  */
    trace ( "i: test data out = " + outData);
    ...
    }
  catch ( Exception e )
    {
    e.printStackTrace();
  /* show exception  */
    }
```

2. The following code protects an MQeMsgObject locally without using MQeLocalSecure.

```
try
 {
   .../*SIMPLE PROTECT FRAGMENT */
   .../*instantiate a DES cryptor */
  MQeDESCryptor desC = new MQeDESCryptor();
   .../*instantiate an Attribute using the DES cryptor */
  MQeAttribute attr = new MQeAttribute(null,desC,null);
   .../*instantiate a base Key object */
  MQeKey localkey = new MQeKey();
  .../*set the base Key object local key */
  localkey.setLocalKey("my secret key");
  .../*attach the key to the attribute */
  attr.setKey(localkey);
   /*instantiate an MQeFields object */
  MQeFields myData = new MQeFields();
   /*attach the attribute to the data object */
  myData.setAttribute(attr);
   /*add some test data */
  myData.putAscii("testdata", "0123456789abcdef....");
  trace ("i:test data in = " + myData.getAscii("testdata"));
   /*encode the data */
  byte [] protectedData = myData.dump();
  trace ("i:protected test data = " + MQe.byteToAscii(protectedData));
 }
catch (Exception e )
 {
     e.printStackTrace();  /*show exception */
 }

try
 {
  .../*SIMPLE UNPROTECT FRAGMENT */
  .../*instantiate a DES cryptor */
  MQeDESCryptor desC2 = new MQeDESCryptor();
  .../*instantiate an Attribute using the DES cryptor */
  MQeAttribute attr2 = new MQeAttribute(null,desC2,null);
  .../*instantiate a base Key object */
  MQeKey localkey2 = new MQeKey();
  .../*set the base Key object local key */
 localkey2.setLocalKey("my secret key");
  .../*attach the key to the attribute */
 attr2.setKey(localkey2 );
  /*instantiate a new data object */
 MQeFields myData2 = new MQeFields();
  /*attach the attribute to the data object */
 myData2.setAttribute(attr2 );
  /*decode the data */
 myData2.restore(protectedData );
  /*show the unprotected test data */
 trace ("i:test data out = " + myData2.getAscii("testdata"));
 }
catch (Exception e )
 {
   e.printStackTrace();   /*show exception */
   }
```

Usage guide for C

1. The following code protects an MQeFields structure using MQeLocalSecure:

```
/* write to a file */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hDir = NULL, hFile = NULL;
MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQeLocalSecureHndl hLocalSecure = NULL;
MQeFieldsHndl     hData = NULL;
MQEBYTE outBuf[128];
MQEINT32 bufLen = 128;

MQERETURN rc;

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
       &hKeySeed,
        "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
       hAttr, NULL,
       MQE_RC4_CRYPTOR_CLASS_NAME,
       NULL, hKeySeed);
/* create a dir string */
rc = mqeString_newChar8( &exceptBlock, &hDir, ".\\");
/* create a file name string */
rc = mqeString_newChar8( &exceptBlock,
       &hFile,
       "localSecureFile.txt");
/* create an MQeLocalSecure */
rc = mqeLocalSecure_new( &exceptBlock, &hLocalSecure);
/* open file */
rc = mqeLocalSecure_open(hLocalSecure, &exceptBlock, hDir, hFile);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* add some test data */
rc = mqeString_newChar8(&exceptBlock,
       &hFieldName,
       "testdata");
rc = mqeString_newChar8(&exceptBlock,
       &hFieldData,
       "0123456789abcdef....");
rc = mqeFields_putAscii(hData, &exceptBlock,
       hFieldName, hFieldData);
/* dump (protect) data Fields */
```

```
rc = mqeFields_dump(hData, &exceptBlock,
      outBuf, &buflen);
/* write to .\\ocalSecureFile.txt */
rc = mqeLocalSecure_write(hLocalSecure, &exceptBlock,
        outBuf, bufLen, hAttr, NULL);

/* read from a file */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl   hKeySeed = NULL, hDir = NULL, hFile = NULL;
MQeStringHndl   hFieldName = NULL, hFieldData = NULL;
MQeExceptBlock  exceptBlock;
MQeLocalSecureHndl hLocalSecure = NULL;
MQERETURN rc;
MQEBYTE outBuf[128];
MQEINT32 bufLen = 128;

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
       &hKeySeed,
       "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
       &hAttr, NULL,
       MQE_RC4_CRYPTOR_CLASS_NAME,
       NULL, hKeySeed);
/* create a dir string */
rc = mqeString_newChar8( &exceptBlock,
       &hDir, ".\\");
/* create a file name string */
rc = mqeString_newChar8( &exceptBlock,
       &hFile,
       "localSecureFile.txt");
/* create an MQeLocalSecure */
rc = mqeLocalSecure_new( &exceptBlock,
       &hLocalSecure);
/* open file */
rc = mqeLocalSecure_open(hLocalSecure, &exceptBlock,
       hDir, hFile);
/* read from .\\ocalSecureFile.txt */
rc = mqeLocalSecure_read(hLocalSecure,
       &exceptBlock, outBuf,
       &Buflen, hAttr, NULL);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* restore data Fields */
rc = mqeFields_restore(hData, &exceptBlock,
       outBuf, bufLen);
/* read test data */
rc = mqeString_newChar8(&exceptBlock, &hFieldName,
       "testdata");
rc = mqeFields_getAscii(hData, &exceptBlock,
       &hFieldData, hFieldName);
```

2. The following code protects an MQeFields structure without using MQeLocalSecure:

```
/* dump to a buffer */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hFieldName =
         NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQeFieldsHndl     hData = NULL;
MQEBYTE           outBuf[128];
MQEINT32          bufLen = 128;
MQERETURN rc;

/* create a key seed string */
```

```
rc = mqeString_newChar8(&exceptBlock,
       &hKeySeed,
       "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
       &hAttr, NULL,
       MQE_RC4_CRYPTOR_CLASS_NAME,
       NULL, hKeySeed);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* set the attribute to the data Fields */
rc = mqeFields_setAttribute(hData, &exceptBlock, hAttr);
/* add some test data */
rc = mqeString_newChar8(&exceptBlock,
       &hFieldName,
       "testdata");
rc = mqeString_newChar8(&exceptBlock,
       &hFieldData,
       "0123456789abcdef....");
rc = mqeFields_putAscii(hData, &exceptBlock,
       hFieldName, hFieldData);
/* dump (protect) data Fields */
rc = mqeFields_dump(hData, &exceptBlock,
      outBuf, &bufLen);

/* restor from a buffer */
MQeFieldsAttrHndl hAttr = NULL;
MQeStringHndl     hKeySeed = NULL, hFieldName =
          NULL, hFieldData = NULL;
MQeExceptBlock    exceptBlock;
MQERETURN rc;
MQEBYTE           outBuf[128];
MQEINT32          bufLen = 128;

...
/* assume protected data is in inBuf
and its length is in bufLen */

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock,
       &hKeySeed,
       "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeFieldsAttr_new(&exceptBlock,
       &hAttr, NULL,
        MQE_RC4_CRYPTOR_CLASS_NAME,
        NULL, hKeySeed);
/* create a data Fields */
rc = mqeFields_new(&exceptBlock, &hData);
/* set the attribute to the data Fields */
rc = mqeFields_setAttribute(hData, &exceptBlock, hAttr);
/* restore data Fields */
rc = mqeFields_restore(hData, &exceptBlock,
       inBuf, bufLen);
/* read test data */
rc = mqeString_newChar8(&exceptBlock,
       &hFieldName, "testdata");
rc = mqeFields_getAscii(hData, &exceptBlock,
       &hFieldData, hFieldName);
```

Message-level security

Message-level security facilitates the protection of message data between an initiating and receiving WebSphere MQ Everyplace application. Message-level security is an application layer service. It requires the initiating WebSphere MQ Everyplace application to create a message-level attribute and provide it when using **putMessage()** to put a message to a target queue.

The receiving application must set up and pass a matching message-level attribute to the receiving queue manager so that the attribute is available when the application invokes **getMessage()** to get the message from the target queue.

Like local security, message-level security exploits the application of an attribute on a message, an MQeFields object descendent. The initiating application's queue manager handles the application's **putMessage()** with the message Java dump method or C API, which invokes the attached attribute's Java **encodeData()** method or C API to protect the message data. The receiving application's queue manager handles the application's **getMessage()** with the message's Java 'restore' method or C API, which in turn uses the supplied attribute's **decodeData()** method to recover the original message data.

Usage scenario

Message-level security is typically most useful for:

- Solutions that are designed to use predominantly asynchronous queues.
- Solutions for which application level security is important, that is solutions whose normal message paths include flows over multiple nodes perhaps connected with different protocols. Message-level security manages trust at the application level, which means security in other layers becomes unnecessary.

A typical scenario is a solution service that is delivered over multiple open networks. For example over a mobile network and the internet, where, from outset asynchronous operation is anticipated. In this scenario, it is also likely that message data is flowed over multiple links that may have different security features, but whose security features are not necessarily controlled or trusted by the solution owner. In this case it is very likely the solution owner does not wish to delegate trust for the confidentiality of message data to any intermediate, but would prefer to manage and control trust management directly.

WebSphere MQ Everyplace message-level security provides solution designers with the features that enable the strong protection of message data in a way that is under the direct control of the initiating and recipient applications, and that ensures the confidentiality of the message data throughout its transfer, end to end, application to application.

Secure feature choices

WebSphere MQ Everyplace supplies two alternative attributes for message-level security.

MQeMAttribute

This suits business-to-business communications where mutual trust is tightly managed in the application layer and requires no trusted third party. It allows use of all available WebSphere MQ Everyplace symmetric cryptor and compressor choices. Like local security it requires the attribute's key to be preset before it is supplied as a parameter on **putMessage()** and **getMessage()**. This provides a simple and powerful method for message-level protection that enables use of strong encryption to protect message confidentiality, without the overhead of any public key infrastructure (PKI).

MQeMTrustAttribute

Note: The MQeMTrustAttribute does not apply to the C codebase.
This provides a more advanced solution using digital signatures and exploiting the default public key infrastructure to provide a digital envelope style of protection. It uses ISO9796 digital signature or validation so that the receiving application can establish proof that the message came from the purported sender. The supplied attribute's cryptor protects message confidentiality. SHA1 digest guarantees message integrity and RSA encryption and decryption, ensuring that the message can only be restored by the intended recipient. As with MQeMAttribute, it allows use of all available WebSphere MQ Everyplace symmetric cryptor and compressor choices. Chosen for size optimization, the certificates used are mini-certificates which conform to the WTLS Specification approved by the WAP forum. WebSphere MQ Everyplace provides a default public key infrastructure to distribute the certificates as required to encrypt and authenticate the messages.

A typical MQeMTrustAtribute protected message has the format:

```
RSA-enc{SymKey}, SymKey-enc {Data, DataDigest, DataSignature}
```

where:

RSA-enc:
RSA encrypted with the intended recipient's public key, from his mini-certificate

SymKey:
Generated pseudo-random symmetric key

SymKey-enc:
Symmetrically encrypted with the `SymKey`

Data: Message data

DataDigest:
Digest of message data

DigSignature:
Initiator's digital signature of message data

Selection Criteria

MQeMAttribute relies totally on the solution owner to manage the content of the key seed that is used to derive the symmetric key used to protect the confidentiality of the data. This key seed must be provided to both the initiating and recipient applications. While it provides a simple mechanism for the strong protection of message data without the need of any PKI, it clearly depends of the effective operational management of the key seed.

MQeMTrustAttribute exploits the advantages of the WebSphere MQ Everyplace default PKI to provide a digital envelope style of message-level protection. This not only protects the confidentiality of the message data flowed, but checks its integrity and enables the initiator to ensure that only the intended recipient can access the data. It also enables the recipient to validate the originator of the data, and ensures that the signer cannot later deny initiating the transaction. This is known as *non-repudiation*.

Solutions that wish to simply protect the end-to-end confidentiality of message data will probably decide that MQeMAttrribute suits their needs, while solutions for which one to one (authenticatable entity to authenticatable entity) transfer and non-repudiation of the message originator are important may find MQeMTrustAttribute is the correct choice.

Usage guide

The following code fragments provide examples of how to protect and unprotect a message using MQeMAttribute, in both Java and C, and alsoMQeMTrustAttribute, which is Java specific.

Message-level security using MAttribute for Java

Note:

```
    /*SIMPLE PROTECT FRAGMENT */
{
     MQeMsgObject msgObj = null;
     MQeMAttribute attr = null;
     long confirmId = MQe.uniqueValue();
     try{
        trace(">>>putMessage to target Q using MQeMAttribute"
               +" with 3DES Cryptor and key=my secret key");
         /* create the cryptor */
        MQe3DESCryptor tdes = new MQe3DESCryptor();
         /* create an attribute using the cryptor */
        attr = new MQeMAttribute(null,tdes,null );
         /* create a local key */
        MQeKey localkey = new MQeKey();
         /* give it the key seed */
        localkey.setLocalKey("my secret key");
         /* set the key in the attribute */
        attr.setKey(localkey );
         /* create the message */
        msgObj = new MQeMsgObject();
        msgObj.putAscii("MsgData","0123456789abcdef...");
         /* put the message using the attribute */
        newQM.putMessage(targetQMgrName, targetQName,
                  msgObj, attr, confirmId );
        trace(">>>MAttribute protected msg put OK...");
       }
    catch (Exception e)
       {
       trace(">>>on exception try resend exactly once...");
       msgObj.putBoolean(MQe.Msg_Resend, true );
       newQM.putMessage(targetQMgrName, targetQName,
                     msgObj, attr, confirmId );
       }
}

    /*SIMPLE UNPROTECT FRAGMENT */
{
       MQeMsgObject msgObj2 = null;
       MQeMAttribute attr2 = null;
       long confirmId2 = MQe.uniqueValue();
    try{
       trace(">>>getMessage from target Q using MQeMAttribute"+
          " with 3DES Cryptor and key=my secret key");
        /* create the attribute - we do not have to specify the cryptor, */
        /* the attribute can get this from the message itself  */
       attr2 = new MQeMAttribute(null,null,null );
        /* create a local key */
       MQeKey localkey = new MQeKey();
        /* give it the key seed */
       localkey.setLocalKey("my secret key");
```

```
                /* set the key in the attribute */
               attr2.setKey(localkey );
                /* get the message using the attribute */
               msgObj2 = newQM.getMessage(targetQMgrName, targetQName,
                                          null, attr2, confirmId2 );
               trace(">>>unprotected MsgData = "
                   + msgObj2.getAscii("MsgData"));
               }
            catch (Exception e)
              {
               /*exception may have left */
              newQM.undo(targetQMgrName,
          /*message locked on queue */
                       targetQName, confirmId2 );
          /*undo just in case */
              e.printStackTrace();
          /*show exception reason */
              }
             ...
        }
```

Message-level security using MAttribute for C

```
   /* putMessage */
   MQeMsgAttrHndl    hAttr = NULL;
   MQeStringHndl     hKeySeed = NULL, hQMgrName =
             NULL, hQName = NULL;
   MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
   MQeExceptBlock    exceptBlock;
   MQeFieldsHndl     hData = NULL;
   MQeQueueManagerHndl hQMgr = NULL;
   MQERETURN rc;

   ...
   /* assume queue manager handle in hQMgr,
 /*QMgr name in hQMgrName, and queue name in hQName */

   /* create a key seed string */
   rc = mqeString_newChar8(&exceptBlock, &hKeySeed,
          "my secret key");
   /* create a new attribute with a RC4 cryptor */
   rc = mqeMsgAttr_new(&exceptBlock, &hAttr, NULL,
         MQE_RC4_CRYPTOR_CLASS_NAME,
         NULL, hKeySeed);
   /* create a data Fields */
   rc = mqeFields_new(&exceptBlock, &hData);
   /* add some test data */
   rc = mqeString_newChar8(&exceptBlock, &hFieldName,
         "MsgData");
   rc = mqeString_newChar8(&exceptBlock, &hFieldData,
          "0123456789abcdef....");
   rc = mqeFields_putAscii(hData, &exceptBlock,
          hFieldName, hFieldData);
   /* send message */
   rc = mqeQueueManager_putMessage(hQMgr, &exceptBlock,
             hQMgrName, hQName,
             hData, hAttr, 0);

   /* getMessage */
   MQeMsgAttrHndl hAttr = NULL;
   MQeStringHndl     hKeySeed = NULL, hQMgrName =
             NULL, hQName = NULL;
   MQeStringHndl     hFieldName = NULL, hFieldData = NULL;
   MQeExceptBlock    exceptBlock;
   MQeQueueManagerHndl hQMgr = NULL;
   MQERETURN rc;

   ...
```

```
/* assume queue manager handle in hQMgr, QMgr
name in hQMgrName, and queue name in hQName */

/* create a key seed string */
rc = mqeString_newChar8(&exceptBlock, &hKeySeed,
       "my secret key");
/* create a new attribute with a RC4 cryptor */
rc = mqeMsgAttr_new(&exceptBlock, &hAttr, NULL,
      MQE_RC4_CRYPTOR_CLASS_NAME,
      NULL, hKeySeed);
/* get message */
rc = mqeQueueManager_getMessage(hQMgr, &exceptBlock,
         &hData, hQMgrName,
         hQName, NULL, hAttr, 0);
/* get test data */
rc = mqeString_newChar8(&exceptBlock, &hFieldName,
       "MsgData");
rc = mqeFields_getAscii(hData, &exceptBlock,
       &hFieldData, hFieldName);
```

Message-level security using MTrustAttribute (Java only)

For an explanation about MQePrivateRegistry and MQePublicRegistry, used in the following example, refer to “Private registry service” on page 97and “Public registry service” on page 100.

```
    /*SIMPLE PROTECT FRAGMENT */
{
    MQeMsgObject msgObj = null;
    MQeMTrustAttribute attr = null;
    long confirmId = MQe.uniqueValue();
try {
    trace(">>>putMessage from Bruce1 intended for Bruce8"
        + " to target Q using MQeMTrustAttribute
     with MARSCryptor ");
     /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
     /* create an attribute using the cryptor */
    attr = new MQeMTrustAttribute(null, mars, null);
     /* open the private registry belonging to the sender */
    String EntityName = "Bruce1";
    String PIN = "12345678";
    Object Passwd = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry();
    sendreg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
                     PIN, Passwd, null, null );
     /* set the private registry in the attribute */
    attr.setPrivateRegistry(sendreg );
     /* set the target (recipient) name in the attribute */
    attr.setTarget("Bruce8");
     /* open a public registry to get the target's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
     /* set the public registry in the attribute */
    attr.setPublicRegistry(pr);
     /* set a home server, which is used to find the certificate*/
     /* if it is not already in the public registry */
    attr.setHomeServer(MyHomeServer +":8082");
     /* create the message */
    msgObj =new MQeMsgObject();
    msgObj.putAscii("MsgData","0123456789abcdef...");
     /* put the message using the attribute */
    newQM.putMessage(targetQMgrName, targetQName,
                     msgObj, attr, confirmId );
    trace(">>>MTrustAttribute protected msg put OK...");
    }
catch (Exception e)
    {
```

```
    trace(">>>on exception try resend exactly once...");
    msgObj.putBoolean(MQe.Msg_Resend, true);
    newQM.putMessage(targetQMgrName, targetQName,
                     msgObj, attr, confirmId );
    }
}


    /*SIMPLE UNPROTECT FRAGMENT */
{
    MQeMsgObject msgObj2 = null;
    MQeMTrustAttribute attr2 = null;
    long confirmId2 = MQe.uniqueValue();
try {
    trace(">>>getMessage from Bruce1 intended for Bruce8"
        + " from target Q using MQeMTrustAttribute with MARSCryptor ");
     /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
     /* create an attribute using the cryptor */
    attr2 = new MQeMTrustAttribute(null, mars, null);
     /* open the private registry belonging to the target */
    String EntityName =  "Bruce8";
    String PIN =  "12345678";
    Object Passwd =  "It_is_a_secret";
    MQePrivateRegistry getreg = new MQePrivateRegistry();
    getreg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
                    PIN, Passwd, null, null );
     /* set the private registry in the attribute */
    attr2.setPrivateRegistry(getreg);
     /* open a public registry to get the sender's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
     /* set the public registry in the attribute */
    attr2.setPublicRegistry(pr);
     /* set a home server, which is used to find the certificate*/
     /* if it is not already in the public registry */
    attr2.setHomeServer(MyHomeServer +":8082");
     /* get the message using the attribute */
    msgObj2 = newQM.getMessage(targetQMgrName,
                               targetQName, null, attr2, confirmId2 );
    trace(">>>MTrustAttribute protected msg = "
          + msgObj2.getAscii("MsgData"));
    }
catch (Exception e)
    {
     /*exception may have left */
    newQM.undo(targetQMgrName,            /*message locked on queue */
             targetQName, confirmId2 ); /*undo just in case */
    e.printStackTrace();               /*show exception reason */
    }
}
```

Non-repudiation

The MQeMTrustAttribute digitally signs messages. This enables the recipient to validate the creator of the message, and ensures that the creator cannot later deny creating the message. This is known as *non-repudiation*. This process depends on the fact that only one public key can validate the signature successfully generated by a particular private key. This validation proves that the signature was created with the corresponding private key. The only way the alleged creator can deny creating the message is to claim that someone else had access to the private key.

When a message is created with the MQeMTrustAttribute, it uses the private key from the sender's private registry to create the digital signature and it stores the sender's name in the message. When the message is read with the queue manager's **getMessage()** method, it uses the sender's public certificate to validate the digital signature. The message is read successfully only if the signature validates successfully, proving that the message was created by the entity whose name was stored in the message as the sender.

When the MQeMTrustAttribute is specified as a parameter to the queue manager's **getMessage()** method, the attribute validates the digital signature but by the time the message is returned to the user's application all the information relating to the signature has been discarded. If non-repudiation is important to you, you must keep a record of this information. The simplest way to do this is to keep a copy of the encrypted message, because that includes the digital signature. You can do this by using the **getMessage()** method without an attribute. This returns the encrypted message which you can then save, for example in a local queue. You can decrypt the message by applying the attribute to access the contents of the message.

The following code fragment provides an example of how to save an encrypted message.

Saving a copy of an encrypted message

```
/*SIMPLE FRAGMENT TO SAVE ENCRYPTED MESSAGE*/
{
MQeMsgObject msgObj2 = null;
MQeMTrustAttribute attr2 = null;
long confirmId2 = MQe.uniqueValue();
long confirmId3 = MQe.uniqueValue();
try {
    trace(">>>getMessage from Bruce1
     intended for Bruce8"
   + " from target Q using MQeMTrustAttribute
   with MARSCryptor ");
    /* read the encrypted message without an attribute */
    MQeMsgObject tmpMsg1 = newQM.getMessage(targetQMgrName,
        targetQName, null, null, confirmId2 );
    /* save the encrypted message -
   we cannot put it directly */
    /* to another queue because of
   the origin queue manager  */
    /* data. Embed it in another message */
    MQeMsgObject tmpMsg2 = new MQeMsgObject();
    tmpMsg2.putFields("encryptedMsg", tmpMsg1);
    newQM.putMessage(localQMgrName, archiveQName,
        tmpMsg2, null, confirmId3);
    trace(">>>encrypted message saved locally");
    /* now decrypt and read the message & */
    /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
    /* create an attribute using the cryptor */
    attr2 = new MQeMTrustAttribute(null, mars, null);
    /* open the private registry belonging to the target */
    String EntityName =  "Bruce8";
    String PIN =  "12345678";
    Object Passwd =  "It_is_a_secret";
    MQePrivateRegistry getreg = new MQePrivateRegistry();
    getreg.activate(EntityName,
     ".\\MQeNode_PrivateRegistry",
     PIN, Passwd, null, null );
    /* set the private registry in the attribute */
    attr2.setPrivateRegistry(getreg);
    /* open a public registry to
   get the sender's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".\\");
    /* set the public registry in the attribute */
    attr2.setPublicRegistry(pr);
    /* set a home server, which is
   used to find the certificate*/
    /* if it is not already in the public registry */
    attr2.setHomeServer(MyHomeServer +":8082");
    /* decrypt the message by unwrapping it */
    msgObj2 = tmpMsg1.unwrapMsgObject(attr2);
    trace(">>>MTrustAttribute protected msg = "
  + msgObj2.getAscii("MsgData"));

catch (Exception e)
  {   /*exception may have left */
    newQM.undo(targetQMgrName,
  /*message locked on queue */
        targetQName, confirmId2 );
  /*undo just in case */
    e.printStackTrace();
  /*show exception reason */
   }
}
```

Mini-certificate issuance service

The ES03 WebSphere MQ Everyplace SupportPac, ″WebSphere MQ Everyplace WTLS Mini-Certificate Server″ is available as a separate free download from *http://www.ibm.com/software/ts/mqseries/txppacs/*. WebSphere MQ Everyplace includes a default *mini-certificate issuance service* that can be configured to satisfy private registry auto-registration requests. With the tools provided, a solution can setup and manage a mini-certificate issuance service so that it issues mini-certificates to a carefully controlled set of entity names. These are a prerequisite for MQeMTrustAttribute-based message-level security. The characteristics of this issuance service are:

- Management of the set of registered authenticatable entities.
- Issuance of mini-certificates. The mini-certificate conforms to the WAP WTLS specification.
- Management of the mini-certificate repository.

The tools provided in the ES03 SupportPac enable a mini-certificate issuance service administrator to authorize mini-certificate issuance to an entity by registering its entity name and registered address and defining a one-time-use *certificate request PIN*. This would normally be done after off line checking to validate the authenticity of the requestor. The certificate request PIN can be posted to the intended user, as bank card PINs are posted when a new card is issued. The user of the private registry (for example the WebSphere MQ Everyplace application or WebSphere MQ Everyplace queue manager) can then be configured to provide this certificate request PIN at startup time. When the private registry triggers auto-registration, the mini-certificate issuance service validates the resulting new certificate request, issues the new mini-certificate and then resets the registered certificate request PIN so it cannot be reused. All auto-registration of new mini-certificate requests is processed on a secure channel.

We recommend that you refer to the MQe_MiniCertificateServer documentation included in the ES03 SupportPac, ″WebSphere MQ Everyplace WTLS Mini-Certificate Server″, for more details of how to install and use the WTLS digital certificate issuance service for WebSphere MQ Everyplace.

Renewing mini-certificates

The certificates issued for an entity by the mini-certificate issuance service are valid for one year from the date of issue and it is advisable to renew them before they expire. Renewed certificates are obtained from the same mini-certificate issuance service. Before requesting a renewal, the request must be authorized with the issuance service and a one-time-use certificate request PIN obtained, in just the same way as for the initial certificate issuance. When you use the server to obtain the PIN for renewal, remember that you are updating the entity, not adding it.

When a certificate is issued for an entity, a copy of the mini-certificate server's own certificate is issued with it. This is needed to check the validity of other certificates. With versions of WebSphere MQ Everyplace earlier than 1.2, the certificate server's certificate could expire before the entity's certificate. If this happens you can renew the server's certificate by requesting a renewal of the entity's certificate; a new copy of the mini-certificate server's certificate will be returned along with the entity's certificate. From mini-certificate server Version 1.2, the mini-certificate server's certificate will expire later than the entity's certificate.

The class com.ibm.mqe.registry.MQePrivateRegistryConfigure contains a method **renewCertificates()** which can be used to request renewed certificates. This is used in the example program examples.certificates.RenewWTLSCertificates, which implements a command-line program that requests renewed certificates from the issuance service

The program has four compulsory parameters:

```
RenewWTLSCertificates <entity> <ini file> <MCS addr> <MCS Pin>
```

where:

entity is the name of the entity for which a renewed certificate is required. This should be either a queue manager, a queue or other authenticatable entity. The name of a queue should be specified as `<queue manager>+<queue>`, for example `myQM+myQueue`.

ini file
is the name of a configuration file that contains a section for the registry. This is typically the same configuration file that is used for the queue manager. For a queue, this typically the configuration file for the queue manager that owns the queue.

MCS addr
is the host name and port address of the mini-certificate server (for example: `myServer:8085`)

MCS Pin
is the one-time use PIN issued by the mini-certificate server administrator to authorize this renewal request.

Obtaining new credentials (private and public keys)

When you renew a certificate, you get an updated certificate for your existing public key. This allows you to continue to use your existing private and public key pair. If you want to change your private and public key pair, you must request new credentials. This includes a request to the mini-certificate issuance service for a new public certificate embodying the new public key. Before requesting a certificate for the new credentials, the request must be authorized with the issuance service and a one-time-use certificate request PIN must be obtained, in the same way as for the initial certificate issuance. (When you use the server to obtain the PIN for the new certificate, remember that you are updating the entity, not adding it.)

The class com.ibm.mqe.registry.MQePrivateRegistryConfigure contains a method **getCredentials()** which can be used to request new credentials. This is used in the example program examples.install.GetCredentials, which implements a GUI program that requests new credentials from the issuance service.

Note: When new credentials are issued, the existing ones are archived in the registry. You will no longer be able to decrypt messages created using your earlier credentials. The new certificate will not validate a digital signature (used with MQeMTrustAttribute) created with your earlier credentials.

Listing mini-certificates

It can be useful to list the certificates in a registry, for example to check on their expiry dates. You can do this using methods in the class com.ibm.mqe.attributes.MQeListCertificates. These are used in the example program examples.certificates.ListWTLSCertificates, which implements a command-line program that lists certificates.

The program has one compulsory and three optional parameters:

```
ListWTLSCertificates <reg Name>[<ini file>] [<level>] [<cert names>]
```

where:

regName
is the name of the registry whose certificates are to be listed. It can be a private registry belonging to a queue manager, a queue or another entity; it can be a public registry, or (for the administrator) it can be the mini-certificate server’s registry. If you want to list the certificates in a queue’s registry, you must specify its name as `<queue manager>+<queue>`, for example `myQM+myQueue`. If you want to list the certificates in a public registry, it must have the name `MQeNode_PublicRegistry`, it will not work for a public registry with any other name. The name of the mini-certificate server’s registry is `MiniCertificateServer`.

ini file
is the name of a configuration file that contains a section for the registry. This is typically the same configuration file that is used for the queue manager or mini-certificate server. For a queue, this is typically the configuration file for the queue manager that owns the queue. This parameter should be specified for all registries except public registries, for which it can be omitted.

level is the level of detail for the listing. This can be:

-b or -brief	prints the names of the certificate, one name per line
-n or -normal	prints the names of the certificates, one per line, followed by their type (old or new format)
-f or -full	prints the names of the certificates, their type, and some of the contents

This parameter is optional and if omitted the ″normal″ level of detail is used.

cert names
is a list of names of the certificates to be listed. It starts with the flag `-cn` followed by names of the certificates, for example: `-cn ExampleQM putQM`. If this parameter is used, only the named certificates are listed. If this parameter is omitted, all the certificates in the registry are listed.

Updated mini-certificate format for WebSphere MQ Everyplace Version 2.0.0.5

The mini-certificates used by WebSphere MQ Everyplace are based on the WTLS certificates used by WAP. The certificates used by WebSphere MQ Everyplace Versions 1.0 and 1.1 were based on the latest draft of the WTLS specification that was available at the time of development. A standard for the certificates has since been approved. In WebSphere MQ Everyplace Version 2.0.0.5, updated mini-certificates that conform to the approved standard have been introduced.

You can upgrade your certificates to the new format by running the mini-certificate server from WebSphere MQ Everyplace Version 2.0.0.5 and renewing the certificates. The renewed certificates are in the new format.

Private registry service

Note: The private registry service does not apply to the C codebase.
This section describes the private registry service provided by WebSphere MQ Everyplace.

Private registry and the concept of authenticatable entity

Queue-based security, that uses mini-certificate based mutual authentication and message-level security, that uses digital signature, have triggered the concept of authenticatable entity. In the case of mutual authentication it is normal to think about the authentication between two users but, messaging generally has no concept of users. The normal users of messaging services are applications and they handle the user concept.

WebSphere MQ Everyplace abstracts the concept of target of authentication from user to authenticatable entity. This does not exclude the possibility of authenticatable entities being people, but this would be application selected mapping.

Internally, WebSphere MQ Everyplace defines all queue managers that can either originate or be the target of mini-certificate dependent services as authenticatable entities. WebSphere MQ Everyplace also defines queues defined to use mini-certificate based authenticators as authenticatable entities. So queue managers that support these services can have one (the queue manager only), or a set (the queue manager and every queue that uses certificate based authenticator) of authenticatable entities.

WebSphere MQ Everyplace provides configurable options to enable queue managers and queues to auto-register as an authenticatable entity. WebSphere MQ Everyplace private registry service, MQePrivateRegistry provides services that enable a WebSphere MQ Everyplace application to auto-register authenticatable entities and manage the resulting credentials.

All application registered authenticatable entities can be used as the initiator or recipient of message-level services protected using MQeMTrustAttribute.

Private registry and authenticatable entity credentials

To be useful every authenticatable entity needs its own credentials. This provides two challenges, firstly how to execute registration to get the credentials, and secondly where to manage the credentials in a secure manner. WebSphere MQ Everyplace private registry services help to solve these two problems. These services can be used to trigger auto-registration of an authenticatable entity creating its credentials in a secure manner and they can also be used to provide a secure repository.

Private registry (a descendent of base registry) adds to base registry many of the qualities of a secure or cryptographic token. For example, it can be a secure repository for public objects (mini-certificates) and private objects (private keys). It provides a mechanism to limit access to the private objects to the authorized user. It provides support for services (for example digital signature, RSA decryption) in such a way that the private objects never leave the private registry. Also, by providing a common interface, it hides the underlying device support.

Auto-registration

WebSphere MQ Everyplace provides default services that support auto-registration. These services are automatically triggered when an authenticatable entity is configured; for example when a queue manager is started, or when a new queue is defined, or when an WebSphere MQ Everyplace application uses MQePrivateRegistry directly to create a new authenticatable entity. When registration is triggered, new credentials are created and stored in the authenticatable entity’s private registry. Auto-registration steps include generating a new RSA key pair, protecting and saving the private key in the private registry; and packaging the public key in a new-certificate request to the default mini-certificate server. Assuming the mini-certificate server is configured and available, and the authenticatable entity has been pre-registered by the mini-certificate server (is authorized to have a certificate), the mini-certificate server returns the authenticatable entity’s new mini-certificate, along with its own mini-certificate and these, together with the protected private key, are stored in the authenticatable entity’s private registry as the entity’s new credentials.

While auto-registration provides a simple mechanism to establish an authenticatable entity’s credentials, in order to support message-level protection, the entity requires access to its own credentials (facilitating digital signature) and to the intended recipient’s public key (mini-certificate).

Usage scenario

The primary purpose of WebSphere MQ Everyplace’s private registry is to provide a private repository for WebSphere MQ Everyplace authenticatable entity credentials. An authenticatable entity’s credentials consist of the entity’s mini-certificate (encapsulating the entity’s public key), and the entity’s keyring protected private key.

Typical usage scenarios need to be considered in relation to other WebSphere MQ Everyplace security features:

Queue-based security with MQeWTLSCertAuthenticator

Whenever queue-based security is used, where a queue attribute is defined with MQeWTLSCertAuthenticator, mini-certificate based mutual authentication, the authenticatable entities involved are WebSphere MQ Everyplace owned. Any queue manager that is to be used to access messages in such a queue, any queue manager that owns such a queue and the queue itself are all authenticatable entities and need to have their own credentials. By using the correct configuration options and setting up and using an instance of WebSphere MQ Everyplace mini-certificate issuance service, auto-registration can be triggered when the queue managers and queues are created, creating new credentials and saving them in the entities' own private registries.

Message-level security with MQeMTrustAttribute

Whenever message-level security is used with MQeMTrustAttribute, the initiator and recipient of the MQeMTrustAttribute protected message are application owned authenticatable entities that must have their own credentials. In this case, the application must use the services of MQePrivateRegistry (and an instance of WebSphere MQ Everyplace mini-certificate issuance service) to trigger auto-registration to create the entities' credentials and to save them in the entities' own private registries.

Secure feature choices

WebSphere MQ Everyplace does not provide support for any alternative secure repository for an authenticatable entity's credentials. If queue-based security with MQeWTLSCertAuthenticator or message-level security using MQeMTrustAttribute are used, private registry services must be used.

Usage guide

Prior to using queue-based security, WebSphere MQ Everyplace owned authenticatable entities must have credentials. This is achieved by completing the correct configuration so that auto-registration of queue managers is triggered. This requires the following steps:

1. Setup and start an instance of WebSphere MQ Everyplace mini-certificate issuance service.
2. Using MQe_MiniCertificateServer, add the name of the queue manager as a valid authenticatable entity, and the entity's one-time-use certificate request PIN.
3. Configure MQePrivateClient1.ini and MQePrivateServer1.ini so that when queue managers are created using SimpleCreateQM, auto-registration is triggered. This section explains which keywords are required in the registry section of the `ini` files, and where to use the entity's one-time-use certificate request PIN.

Prior to using message-level security to protect messages using MQeMTrustAttribute, the application must use private registry services to ensure that the initiating and recipient entities have credentials. This requires the following steps:

1. Setup and start an instance of WebSphere MQ Everyplace mini-certificate issuance service.
2. Add the name of the application entity, and allocate the entity a one-time-use certificate request PIN.
3. Use a program similar to the pseudo-code fragment below to trigger auto-registration of the application entity . This creates the entity's credentials and saves them in its private registry.

```
/* SIMPLE MQePrivateRegistry FRAGMENT*/
   try
      {
      /* setup PrivateRegistry parameters  */
      String EntityName      = "Bruce";
      String EntityPIN           = "11111111";
      Object KeyRingPassword      = "It_is_a_secret";
      Object CertReqPIN           = "12345678";
```

```
    Object CAIPAddrPort         = "9.20.X.YYY:8082";
    /* instantiate and activate a
 Private Registry. */
    MQePrivateRegistry preg  = new MQePrivateRegistry( );
    preg.activate( EntityName,
/* entity name               */
                   ".//MQeNode_PrivateRegistry",
/* directory root  */
                   EntityPIN,
/* private reg access PIN     */
                   KeyRingPassword,
/* private credential keyseed */
                   CertReqPIN,
/* on-time-use Cert Req PIN   */
                   CAIPAddrPort );
/* addr and port MiniCertSvr  */
    trace(">>> PrivateRegistry activated OK ...");
    }
 catch (Exception e)
    {
    e.printStackTrace( );
    }
```

Public registry service

This section describes the public registry service provided by WebSphere MQ Everyplace.

WebSphere MQ Everyplace provides default services facilitating the sharing of authenticatable entity *public credentials* (mini-certificates) between WebSphere MQ Everyplace nodes. Access to these mini-certificates is a prerequisite for message-level security. WebSphere MQ Everyplace public registry, also a descendent of base registry, provides a publicly accessible repository for mini-certificates. This is analogous to the personal telephone directory service on a mobile phone, the difference being that it is a set of mini-certificates of the authenticatable entities instead of phone numbers. WebSphere MQ Everyplace public registry is not a purely passive service. If accessed to provide a mini-certificate that is does not hold, and if the public registry is configured with a valid home server, the public registry automatically attempts to get the requested mini-certificate from the public registry of the home server. It also provides a mechanism to share a mini-certificate with the public registry of other WebSphere MQ Everyplace nodes. Together these services provide the building blocks for an intelligent automated mini-certificate replication service that can facilitates the availability of the right mini-certificate at the right time.

Usage scenario

A typical scenario for the use of the public registry would be to use these services so that the public registry of a particular WebSphere MQ Everyplace node builds up a store of the most frequently needed mini-certificates as they are used.

A simple example of this is to setup an WebSphere MQ Everyplace client to automatically get the mini-certificates of other authenticatable entities that it needs, from its WebSphere MQ Everyplace home server, and then save them in its public registry.

Secure feature choices

It is the Solution creator’s choice whether to use the public registry active features for sharing and getting mini-certificates between the public registries of different WebSphere MQ Everyplace nodes.

The alternative to this intelligent replication may be to have an out-of-band utility to initialize an WebSphere MQ Everyplace node's public registry with all required mini-certificates before enabling any secure services that uses them.

Selection criteria

Out-of-band initialization of the set of mini-certificates available in an WebSphere MQ Everyplace node's public registry may have advantages over using the public registry active features in the case where the solution is predominantly asynchronous and the synchronous connection to the WebSphere MQ Everyplace node's home server may be difficult. But in the case where this connection is more likely to be available, the public registry's active mini-certificate replication services are useful tools to automatically maintain the most useful set of mini-certificates on any WebSphere MQ Everyplace node public registry.

Usage guide

```
/*SIMPLE MQePublicRegistry shareCertificate FRAGMENT */
   try {
   String EntityName = "Bruce";
   String EntityPIN = "12345678";
   Object KeyRingPassword = "It_is_a_secret";
   Object CertReqPIN = "12345678";
   Object CAIPAddrPort = "9.20.X.YYY:8082";
/*instantiate and activate PublicReg */
   MQePublicRegistry pubreg = new MQePublicRegistry();
   pubreg.activate("MQeNode_PublicRegistry",".\\");
/* auto-register Bruce1,Bruce2...Bruce8 */
/* ... note that the mini-certificate issuance service must */
/* have been configured to allow the auto-registration    */
   for (int i = 1; i < 9; i++)
   {
   EntityName = "Bruce"+(new Integer(i)).toString();
   MQePrivateRegistry preg = new MQePrivateRegistry();
/* activate() will initiate auto-registration */
   preg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
    EntityPIN, KeyRingPassword, CertReqPIN, CAIPAddrPort);
/* save MiniCert from PrivReg in PubReg*/
    pubreg.putCertificate(EntityName,
     preg.getCertificate(EntityName ));
/*before share of MiniCert */
   pubreg.shareCertificate(EntityName,
    preg.getCertificate(EntityName ),"9.20.X.YYY:8082");
   preg.close();
   }
   pubreg.close();
   }
   catch (Exception e)
   {
   e.printStackTrace();
   }
```

Notes:

1. It is not possible to activate a registry instance more than once, hence the example above demonstrates the recommended practice of accessing a private registry by creating a new instance of MQePrivateRegistry, activating the instance, performing the required operations and closing the instance.
2. If you want to share certificates using a public registry on the home-server, the public registry must be called `MQeNode_PublicRegistry`.

Chapter 9. Java Message Service

The WebSphere MQ Everyplace classes for Java Message Service (JMS) are a set of Java classes that implement the Sun JMS interfaces to enable JMS programs to access WebSphere MQ Everyplace systems. This chapter describes how to use the WebSphere MQ Everyplace classes for JMS.

The initial release of JMS classes for WebSphere MQ Everyplace Version 2.0.0.5, supports the point-to-point model of JMS, but does not support the publish or subscribe model.

The use of JMS as the API to write WebSphere MQ Everyplace applications has a number of benefits, because JMS is open standard:

- The protection of investment, both in skills and application code
- The availability of people skilled in JMS application programming
- The ability to write messaging applications that are independent of the JMS implementations

This chapter contains information under the following headings:

- Using JMS with WebSphere MQ Everyplace
- Writing JMS programs
- Restrictions in this version of WebSphere MQ Everyplace
- Using Java Naming and Directory Interface (JNDI)
- Mapping JMS messages to WebSphere MQ Everyplace messages

More information about the benefits of the JMS API is on Sun’s Web site at http://java.sun.com.

Using JMS with WebSphere MQ Everyplace

This section describes how to set up your system to run the example programs, including the Installation Verification Test (IVT) example which verifies your WebSphere MQ Everyplace JMS installation. To use JMS with WebSphere MQ Everyplace you must have the following jar files, in addition to MQeBase.jar, on your class path:

jms.jar
This is Sun’s interface definition for the JMS classes

MQeJMS.jar
This is the WebSphere MQ Everyplace implementation of JMS

Obtaining jar files

WebSphere MQ Everyplace does not ship with Sun’s JMS interface definition, which is contained in jms.jar, and this must be downloaded before JMS can be used. At the time of writing, this can be freely downloaded from http://java.sun.com/products/jms/docs.html
The JMS Version 1.0.2b jar file is required.

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be on the classpath. If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath. If Java 2 is being used, a 1.2 or later JRE, the JRE might contain these classes. You can use WebSphere MQ Everyplace without JNDI, but at the cost of a small degree of provider dependence.WebSphere MQ Everyplace-specific classes must be used for the ConnectionFactory and Destination objects. You can download JNDI jar files from http://java.sun.com/products/jndi

Testing the JMS class path

You can use the example program `examples.jms.MQeJMSIVT` to test your JMS installation. Before you run this program, you need a WebSphere MQ Everyplace queue manager that has a `SYSTEM.DEFAULT.LOCAL.QUEUE`. In addition to the JMS jar files mentioned above, you also need the following or equivalent jar files on your class path to run `examples.jms.MQeJMSIVT`:

- MQeBase.jar
- MQeExamples.jar

You can run the example from the command line by typing:

```
 java examples.jms.MQeJMSIVT -i
   <ini file name>
```

where <ini file name> is the name of the initialisation (ini) file for the WebSphere MQ Everyplace queue manager. You can optionally add a ″-t″ flag to turn tracing on:

```
java examples.jms.MQeJMSIVT -t -i
   <ini file name>
```

The example program checks that the required jar files are on the class path by checking for classes that they contain. It creates a *QueueConnectionFactory* and configures it using the ini file name that you passed in on the command line. It starts a connection, which:

1. Starts the WebSphere MQ Everyplace queue manager
2. Creates a JMS Queue representing the queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager
3. Sends a message to the JMS Queue
4. Reads the message back and compares it to the message it sent

The `SYSTEM.DEFAULT.LOCAL.QUEUE` should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The output from the program should look like this:

```
using ini file '<.ini file name>'
  to configure the connection
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
Creating and configuring QueueConnectionFactory
Creating connection
From the connection data, JMS
 provider is IBM WebSphere MQ Everyplace Version 2.0.0.0
Creating session
Creating queue
Creating sender
Creating receiver
Creating message
Sending message
Receiving message
```

```
HEADER FIELDS
----------------------------------------
 JMSType:          jms_text
 JMSDeliveryMode:  2
 JMSExpiration:    0
 JMSPriority:      4
 JMSMessageID:     ID:00000009524cf094000000f052fc06ca
 JMSTimestamp:     1032184399562
 JMSCorrelationID: null
 JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
 JMSReplyTo:       null
 JMSRedelivered:   false

 PROPERTY FIELDS (read only)
-----------------------------------------
 JMSXRcvTimestamp : 1032184400133

 MESSAGE BODY (read only)
-----------------------------------------
A simple text message from the MQeJMSIVT program

Retrieved message is a TextMessage; now checking
for equality with the sent message
Messages are equal. Great!
Closing connection
connection closed
IVT finished
```

Running other WebSphere MQ Everyplace JMS example programs

WebSphere MQ Everyplace provides two other example programs for the JMS classes. The program `examples.jms.PTPSample01` is similar to the IVT examples described above, but there is a command line argument to tell it not to use the *Java Naming and Directory Interface* (JNDI) and it does not have the same checks on the class path. The program requires the same JMS and WebSphere MQ Everyplace jar files on the class path as `examples.jms.MQeJMSIVT`, that is jms.jar, MQeJMS.jar, MQeBase.jar, and MQeExamples.jar. It also requires the jndi.jar file, even if it does not use JNDI, because the program imports javax.naming. The section on Using JNDI provides more information on the jndi.jar file. You can run the example from the command line by typing:

```
 java examples.jms.PTPSample01 -nojndi -i <ini file name>
```

where <ini file name > is the name of the initialisation (ini) file for the WebSphere MQ Everyplace queue manager. By default, the program will use the `SYSTEM.DEFAULT.LOCAL.QUEUE` on this queue manager. You can specify a different queue by using the -q flag:

```
java examples.jms.PTPSample01 -i <ini file name> -q <queue name>
```

You can also turn tracing on by adding the -t flag:

```
java examples.jms.PTPSample01 -t -i <ini file name> -q <queue name>
```

The `examples.jms.PTPSample02` program uses message listeners and filters. This program creates a *QueueReceiver* with a ″blue″ filter and creates a message listener for it. It creates a second QueueReceiver with a ″red″ filter and message listener. It then sends four messages to a queue, two with the filter property colour set to blue and two with the filter property colour set to red, and checks that the message listeners receive the correct messages. The program has the same command line parameters as `examples.jms.PTPSample01`.

Writing JMS programs

This section provides information on writing WebSphere MQ Everyplace JMS applications. It provides a brief introduction to the JMS model and information on programming some common tasks that application programs may need to perform.

The JMS model

JMS defines a generic view of a message service. It is important to understand this view, and how it maps onto the underlying WebSphere MQ Everyplace system. The generic JMS model is based around the following interfaces that are defined in Sun's javax.jms package:

Connection
This provides a connection to the underlying messaging service and is used to create *Sessions*.

Session
This provides a context for producing and consuming messages, including the methods used to create *MessageProducers* and *MessageConsumers*.

MessageProducer
This is used to send messages.

MessageConsumer
This is used to receive messages.

Destination
This represents a message destination.

Note: A connection is thread safe, but sessions, message producers, and message consumers are not. While the JMS specification allows a Session to be used by more than one thread, it is up to the user to ensure that Session resources are not concurrently used by multiple threads. The recommended strategy is to use one Session per application thread.

Therefore, in WebSphere MQ Everyplace terms:

Connection
This provides a connection to a WebSphere MQ Everyplace queue manager. All the Connections in a JVM must connect to the same queue manager, because WebSphere MQ Everyplace supports a single queue manager per JVM. The first connection created by an application will try and connect to an already running queue manager, and if that fails will attempt to start a queue manager itself. Subsequent connections will connect to the same queue manager as the first connection.

Session
This does not have an equivalent in WebSphere MQ Everyplace

Message producer and message consumer
These do not have direct equivalents in WebSphere MQ Everyplace. The MessageProducer invokes the `putMessage()` method on the queue manager. The MessageConsumer invokes the `getMessage()` method on the queue manager.

Destination
This represents a WebSphere MQ Everyplace queue.

Note: WebSphere MQ Everyplace JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put messages to or receive messages from a synchronous remote queue.

The generic JMS interfaces are subclassed into more specific versions for Point-to-point and Publish or Subscribe behaviour. WebSphere MQ Everyplace implements the Point-to-point subclasses of JMS. The Point-to-point subclasses are:

QueueConnection
Extends Connection

QueueSession
Extends Session

QueueSender
Extends MessageProducer

QueueReceiver
Extends MessageConsumer

Queue
Extends destination

We recommend writing application programs that use only references to the interfaces in javax.jms. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- Queue

These are known as "administered objects", that is, objects that can be administered and stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation. However, on small devices looking up objects in a JNDI namespace may be impractical or represent an unnecessary overhead. We, therefore, provide two versions of the `QueueConnectionFactory` and Queue classes. The parent classes, `MQeQueueConnectionFactory.class`, `MQeJMSQueue.class`, provide the base JMS functionality but cannot be stored in JNDI, while subclasses, MQeJNDIQueueConnectionFactory.class, and the MQeJMSJNDIQueue.class, add the necessary functionality for them to be stored and retrieved from JNDI.

Building a connection

You normally build connections indirectly using a connection factory. A JNDI namespace can store a configured factory, therefore insulating the JMS application from provider-specific information. See the section Using JNDI, below, for details on how to store and retrieve objects using JNDI.

If a JNDI namespace is not available, you can create factory objects at runtime. However, this reduces the portability of the JMS application because it requires references to WebSphere MQ Everyplace specific classes. The following code creates a QueueConnectionFactory. The factory uses a WebSphere MQ Everyplace queue manager that is configured with an initialisation (ini) file:

```
QueueConnectionFactory factory;
factory = new com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory();
((com.ibm.mqe.jms.MQeJNDIQueueConnectionFactory)factory).
setIniFileName(<initialisation file>)
```

The section on Configuring WebSphere MQ Everyplace JMS objects provides more information about configuring connection factories.

Using the factory to create a connection: Use the createQueueConnection() to create a QueueConnection:

```
QueueConnection connection;
connection = factory.createQueueConnection();
```

Starting the connection

Under the JMS specification, connections are not active upon creation. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. Use the following command to start the connection:

```
connection.start();
```

Obtaining a session

Once a connection has been created, you can use the `createQueueSession()` method on the QueueConnection to obtain a session. The method takes two parameters:

1. A boolean that determines whether the session is ″transacted″ or ″non-transacted″.
2. A parameter that determines the ″acknowledge″ mode. This is used when the session is ″non-transacted″.

The simplest case is that where acknowledgements are used and are handled by JMS itself with `AUTO_ACKNOWLEDGE`, as shown in the following code fragment:

```
QueueSession session;
boolean transacted = false;
session = connection.createQueueSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

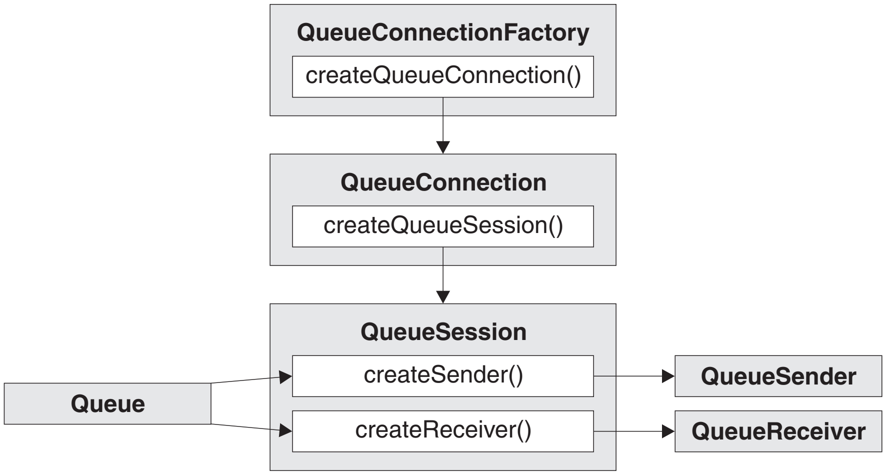


Figure 7. Obtaining a session once a connection is created

Sending a message

Messages are sent using a MessageProducer. For point-to-point this is a QueueSender that is created using the `createSender()` method on QueueSession. A QueueSender is normally created for a specific Queue, so that all messages sent using that sender are sent to the same destination. Queue objects can be either created at runtime, or built and stored in a JNDI namespace. Refer to “Using Java Naming and Directory Interface (JNDI)” on page 115, for details on how to store and retrieve objects using JNDI.

JMS provides a mechanism to create a Queue at runtime that minimizes the implementation-specific code in the application. This mechanism uses the QueueSession.createQueue() method, which takes a string parameter describing the destination. The string itself is still in an implementation-specific format, but this is a more flexible approach than directly referencing the implementation classes.

For WebSphere MQ Everyplace JMS the string is the name of the WebSphere MQ Everyplace queue. This can optionally contain the queue manager name. If the queue manager name is included, the queue name is separated from it by a plus sign ′+′, for example:

```
ioQueue = session.createQueue("myQM+myQueue");
```

This will create a JMS Queue representing the WebSphere MQ Everyplace queue ″myQueue″ on queue manager ″myQM″. If no queue manager name is specified the local queue manager is used, i.e. the one that JMS is connected to. For example:

```
String queueName = "SYSTEM.DEFAULT.LOCAL.QUEUE";

...

ioQueue = session.createQueue(queueName);
```

This will create a JMS Queue representing the WebSphere MQ Everyplace queue `SYSTEM.DEFAULT.LOCAL.QUEUE` on the queue manager that the JMS Connection is using.

Message types: JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the implementation-specific class names for the message types, methods are provided on the Session object for message creation. In the sample program, a text message is created in the following manner:

```
System.out.println("Creating a TextMessage");
TextMessage outMessage = session.createTextMessage();
System.out.println("Adding Text");
outMessage.setText(outString);
```

The message types that can be used are:
- BytesMessage
- ObjectMessage
- TextMessage

Note: Note that Version 2.0 of WebSphere MQ Everyplace does not support these two message types:
- MapMessage
- StreamMessage

The message types are described in more detail later in this section.

Receiving a message

Messages are received by using a QueueReceiver. This is created from a Session by using the `createReceiver()` method. This method takes a Queue parameter that defines where the messages are received from. See ″Sending a message″ above for details of how to create a Queue object. The sample program creates a receiver and reads back the test message with the following code:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue);
Message inMessage = queueReceiver.receive(1000);
```

The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. You can omit this parameter, in which case the call blocks indefinitely.

If you do not want any delay, use the `receiveNoWait()` method. The receive methods return a message of the appropriate type. For example, if a TextMessage is put on a queue, when the message is received the object that is returned is an instance of TextMessage . To extract the content from the body of the message, it is necessary to cast from the generic Message class, which is the declared return type of the receive methods, to the more specific subclass, such as TextMessage . If the received message type is not known, you can use the ″instanceof″ operator to determine which type it is. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully. The following code illustrates the use of ″instanceof″, and extraction of the content from a TextMessage:

```
if (inMessage instanceof TextMessage){
    String replyString = ((TextMessage)inMessage).getText();
     ...
} else {
    //Print error message if Message was not a TextMessage.
    System.out.println("Reply message was not a TextMessage");
}
```

Message selectors: JMS provides a mechanism to select a subset of the messages on a queue so that only the subset is returned by a receive call. When creating a WebSphere MQ Everyplace, a string can be provided that contains an Structured Query Language (SQL) expression to determine which messages to retrieve. The selector can refer to fields in the JMS message header as well as fields in the message properties (these are effectively application-defined header fields).

This version of WebSphere MQ Everyplace JMS supports a restricted set of selectors, which are equivalent to the filter mechanism within WebSphere MQ Everyplace itself. The message selectors supported by WebSphere MQ Everyplace JMS are described in the JMS Messages section below.

The following example shows how to select on a user-defined property named myProp:

```
queueReceiver = session.createReceiver(ioQueue, "myProp ='blue'");
```

Note: The JMS specification does not permit the selector associated with a receiver to be changed. Once a receiver is created, the selector is fixed for the lifetime of that receiver. This means that if you require different selectors, you must create new receivers.

Asynchronous delivery: An alternative to making calls to `QueueReceiver.receive()` is to register a method that is called automatically when a suitable message is available. The following fragment illustrates the mechanism:

```
import javax.jms.*;
public class MyClass implements MessageListener
{
    // The method that will be called by JMS when a message is available.
    public void onMessage(Message message)
    {
        System.out.println("message is "+message);
        //application specific processing here
          ...
    }
}
  ...
//In Main program (possibly of some other class)
```

```
MyClass listener = new MyClass();
queueReceiver.setMessageListener(listener);
//main program can now continue with other application specific
//behaviour.
```

Note: Use of asynchronous delivery with a QueueReceiver marks the entire Session as asynchronous. All other QueueReceivers belonging to the same Session should also use asynchronous delivery. It is an error to make an explicit call to the receive methods of a QueueReceiver that is associated with a Session that is using asynchronous delivery.

WebSphere MQ Everyplace strictly interprets the JMS specification requirement that Sessions are single threaded. This has the following consequences:

- The connection must be in stopped mode to set up a session with more than one message listener. The reason for this is that when a connection is actively delivering messages, once the first message listener for a session has been registered, the session is now controlled by the thread of control that delivers messages to it. At that point a client thread of control cannot be used to further configure the session.
- Once an asynchronous listener is active the commit() method may only be called from the message listener. In other circumstances the listener must first be stopped in order to make the thread of control available to the client.

Handling errors

Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and handle them appropriately. Unlike normal Java Exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important detail from the underlying transport. When a JMSException is thrown as a result of WebSphere MQ Everyplace raising an exception, the exception is usually included as the embedded exception in the JMSException. The standard implementation of JMSException does not include the embedded exception in the output of its `toString()` method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in the following fragment:

```
try {
    ...code which may throw a JMSException
} catch (JMSException je) {
    System.err.println("caught "+je);
    Exception e = je.getLinkedException();
    if (e != null) {
        System.err.println("linked exception:"+e);
    }
}
```

Exception listener: For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to `receive()` methods. To cope with this situation, it is possible to register an ExceptionListener, which is an instance of a class that implements the `onException()` method. When a serious error occurs, this method is called with the `JMSException` passed as its only parameter. Further details are in Sun’s JMS documentation.

JMS messages

JMS messages are composed of the following parts:

Header All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

Properties Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

Body JMS defines several types of message body which cover the majority of messaging styles currently in use. JMS defines five types of message body:

Text A message containing a `java.lang.String`

Object
A message that contains a Serializable java object

Bytes A stream of uninterpreted bytes for encoding a body to match an existing message format

Stream
A stream of Java primitive values filled and read sequentially, not supported in this version of WebSphere MQ Everyplace JMS

Map A set of name-value pairs, where names are Strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined. Map is not supported in this version of WebSphere MQ Everyplace JMS.

The `JMSCorrelationID` header field is used to link one message with another. It typically links a reply message with its requesting message.

Message selectors: A Message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, via message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be boolean, byte, short, int, long, float, double, and String.
- The JMSX and JMS_ name prefixes are reserved.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWriteableException is thrown. If `clearProperties()` is called, the properties can then be both read from, and written to.

A property value may duplicate a value in a message's body, or it may not. JMS does not define a policy for what should or should not be made into a property. However, for best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection. A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages whose headers match the selector are delivered. Message selectors cannot reference message body values. A message selector matches a message when the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, which can contain:

Literals

- A string literal is enclosed in single quotes. A doubled single quote represents a single quote. Examples are ′literal′ and ′literal′′s′. Like Java string literals, these use the Unicode character encoding.
- An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java long are supported.
- An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. Numbers in the range of Java double are supported. Note that rounding errors may affect the operation of message selectors including approximate numeric literals.
- The boolean literals `TRUE` and `FALSE`.

Identifiers

- An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method `Character.isJavaLetter` returns `true`. This includes ″_″ and ″$″. A letter or digit is any character for which the method `Character.isJavaLetterOrDigit` returns `true`.
- Identifiers cannot be the names `NULL`, `TRUE`, or `FALSE`.
- Identifiers cannot be `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, and `IS`.
- Identifiers are either header field references or property references.
- Identifiers are case-sensitive.
- Message header field references are restricted to:
  - JMSDeliveryMode
  - JMSPriority
  - JMSMessageID
  - JMSTimestamp
  - JMSCorrelationID
  - JMSType

  JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values may be null, and if so, are treated as a NULL value.
- Any name beginning with ″JMSX″ is a JMS-defined property name
- Any name beginning with ″JMS_″ is a provider-specific property name
- Any name that does not begin with ″JMS″ is an application-specific property name
- If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.

White space

This is the same as is defined for Java, space, horizontal tab, form feed, and line terminator.

Logical operators

Currently supports `AND` only.

Comparison operators

- Only equals (′=′) is currently supported.
- Only values of the same type can be compared.

- If there is an attempt to compare different types, the selector is always false.
- Two strings are equal if they contain the same sequence of characters.
- The `IS NULL` comparison operator tests for a null header field value, or a missing property value. The `IS NOT NULL` comparison operator is not supported.

Note that Arithmetic operators are not currently supported.

The following message selector selects messages with a message type of car and a colour of blue: `"JMSType ='car 'AND colour ='blue'"`

When selecting Header fields WebSphere MQ Everyplace will interpret exact numeric literals so that they match the type of the field in question, that is a selector testing the JMSPriority or JMSDeliveryMode Header fields will interpret an exact numeric literal as an int, whereas a selector testing JMSExpiration or JMSTimestamp will interpret an exact numeric literal as a long. However, when selecting message properties WebSphere MQ Everyplace will always interpret an exact numeric literal as a long and an approximate numeric literal as a double. Application specific properties intended to be used for message selection should therefore be set using the `setLongProperty` and `setDoubleProperty` methods respectively.

Restrictions in this version of WebSphere MQ Everyplace

This version of WebSphere MQ Everyplace JMS implements the Point-to-Point subset of JMS with a few restrictions. It does not implement any of the optional classes:

- The application server classes ConnectionConsumer, ServerSession, and ServerSessionPool
- The XA classes:
  - XAConnection
  - XAConnectionFactory
  - XAQueueConnection
  - XAQueueConnectionFactory
  - XAQueueSession
  - XASession
  - XATopicConnection
  - XATopicConnectionFactory
  - XATopicSession

It does not implement the TemporaryQueue class, which means that the QueueRequestor class will not work or the MapMessage and StreamMessage classes.

In the `QueueConnectionFactory`, the `createQueueConnection()` method that takes a username and password as parameters is not implemented, WebSphere MQ Everyplace does not have the concept of a user. The method with no parameters is implemented.

When a message is read from a queue but not acknowledged, the message is returned to the queue for redelivery. In this case the JMSRedelivered header field should be set in the message. WebSphere MQ Everyplace JMS does not set this header field.

WebSphere MQ Everyplace JMS can put messages to a local queue or an asynchronous remote queue and it can receive messages from a local queue. It cannot put to or receive messages from a synchronous remote queue.

Using Java Naming and Directory Interface (JNDI)

One of the advantages of using JMS is the ability to write applications which are independent of the JMS implementations, allowing you to plug in a JMS implementation which is appropriate for your environment. However, certain JMS objects must be configured in a way which is specific to the JMS implementation you have chosen. These objects are the connection factories and destinations, queues, and they are often referred to as ″administered objects″. In order to keep the application programs independent of the JMS implementation, these objects must be configured outside of the application programs. They would typically be configured and stored in a JNDI namespace. The application would lookup the objects in the namespace and would be able to use them straight away, because they have already been configured.

There may be situations, such as on a small device, where it would not be desirable to use JNDI. In these cases the objects could be configured directly in the application. The cost of not using JNDI would be a small degree of implementation-dependence in the application.

Storing and Retrieving objects with JNDI

Before using JNDI to either store or retrieve objects, an ″initial context″ must be set up, as shown in this fragment taken from the MQeJMSIVT_JNDI example program:

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;

...
java.util.Hashtable environment =new java.util.Hashtable();
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
environment.put(Context.PROVIDER_URL, url);
Context ctx = new InitialContext(environment );
```

where:

icf defines a factory class for the JNDI context. This depends upon the JNDI provider that you are using. The documentation supplied by the JNDI provider should tell you what value to use for this. See also the examples below.

url defines the location of the namespace. This will depend on the type of namespace you are using. If you are using the file system, this will be a file url that identifies a directory in your file system. If you are using LDAP this will be a ldap url that identifies a LDAP server and location in the directory tree of that server. The documentation supplied by the JNDI provider should describe the correct format for the url.

For more details about JNDI usage, see Sun’s JNDI documentation.

Note: Some combinations of the JNDI packages and LDAP service providers can result in an LDAP error 84. To resolve the problem, insert the following line before the call to InitialContext.

```
environment.put(Context.REFERRAL,"throw");
```

Once an initial context is obtained, objects can be stored in and retrieved from the namespace. To store an object, use the `bind()` method:

```
ctx.bind(entryName, object);
```

where ’entryName’ is the name under which you want the object stored, and ’object’ is the object to be stored, for example to store a factory under the name ″ivtQCF″:

```
ctx.bind("ivtQCF", factory);
```

To store an object in a JNDI namespace, the object must satisfy either the javax.naming.Referenceable interface or the java.io.Serializable interface, depending on the JNDI provider you use. The `MQeJNDIQueueConnectionFactory` and `MQeJMSJNDIQueue`classes implement both of these interfaces. To retrieve an object from the namespace, use the`lookup()` method:

```
object = ctx.lookup(entryName);
```

where `entryName` is the name under which you want the object stored , for example, to retrieve a `QueueConnectionFactory` stored under the name ″ivtQCF″:

```
QueueConnectionFactory factory;
factory = (QueueConnectionFactory)ctx.lookup("ivtQCF");
```

Using the example programs with JNDI

The example program `examples.jms.MQeJMSIVT_JNDI` can be used to test your installation using JNDI. This is very similar to the `examples.jms.MQeJMSIVT` program, except that it uses JNDI to retrieve the connection factory and the queue that it uses. Before you can run this program you must store these two administered objects in a JNDI namespace:

Table 6. Administered objects for a JNDI namespace

Entry name	Java class	Description
ivtQCF	`MQeJNDIQueueConnectionFactory`	A `QueueConnectionFactory` configured to use a WebSphere MQ Everyplace queue manager
ivtQ	`MQeJMSJNDIQueue`	A Queue configured to represent a WebSphere MQ Everyplace queue which is local to the queue manager used by the ivtQCF entry

The program examples.jms.CreateJNDIEntry or the MQeJMSAdmin tool , explained in the following section, can be used to create these entries. Larger installations may have a *Lightweight Directory Access Protocol (LDAP)* directory available, but for smaller installations a file system namespace may be more appropriate. When you have decided on a namespace you must obtain the corresponding JNDI class files to support the namespace and add these to your classpath. These will vary depending on your choice of namespace and the version of Java you are using.

You must always have the javax.naming.* classes on your classpath. If you are using Java 1 (for example a 1.1.8 JRE) you must obtain a copy of the jndi.jar file and add it to your classpath. If you are using Java 2 (a 1.2 or later JRE) the JRE may contain these classes itself.

If you want to use an LDAP directory, you must obtain JNDI classes that support LDAP, for example Sun's ldap.jar or IBM's ibmjndi.jar, and add these to your classpath. Some Java 2 JREs may already contain Sun's classes for LDAP. See also the section below about LDAP support for Java classes.

If you want to use a file system directory, you must obtain JNDI classes that support the file system, for example Sun's fscontext.jar (which requires providerutil.jar as well) and add these to your classpath. The CreateJNDIEntry example program requires the MQeJMS.jar file on your classpath, in addition to the JNDI jar files. It takes the following command line arguments:

```
java examples.jms.CreateJNDIEntry -url<providerURL>
          [-icf<initialContextFactory>][-ldap]
          [-qcf<entry name><MQe queue manager ini file>]
          [-q<entry name><MQe queue name>]
```

An alternative arguement to use is:

```
java  examples.jms.CreateJNDIEntry -h
```

In the previous two examples:

-url<providerURL>
The URL of the JNDI initial context (obligatory parameter)

-icf<initialContextFactory>
The initialContextFactory for JNDI that defaults to the file system:

```
com.sun.jndi.fscontext.RefFSContextFactory
```

-ldap This should be specified if you are using an LDAP directory

-qcf<entry name><MQe queue manager ini file>
The name of a JNDI entry to be created for a JMS `QueueConnectionFactory` and the name of an initialisation (ini) file for a WebSphere MQ Everyplace queue manager to be used to configure it

-h Displays a help message

The url, -url, must be specified and either a QueueConnectionFactory (-qcf) or a Queue (-q), or both, must be specified. The context factory, -icf, is optional and defaults to a file system directory. The LDAP flag, -ldap, should be specified if an LDAP directory is being used, this prefixes the entry name with "cn=", which is required by LDAP.

For example, if a queue manager with the initialisation file *d:\MQe\exampleQM\exampleQM.ini* exists, and you are using a JNDI directory based in the file system at *d:\MQe\data\jndi\*, type (all on one line):

```
java  examples.jms.CreateJNDIEntry -url file://d:/MQe/data/jndi -qcf  ivtQCF
d:\MQe\exampleQM\exampleQM.ini
```

Note that forward slashes are used in the url, even if the file system itself uses back slashes. The url directory must already exist. To add an entry for the queue you would type (all on one line):

```
java  examples.jms.CreateJNDIEntry -url file://
d:/MQe/data/jndi -q ivtQ SYSTEM.DEFAULT.LOCAL.QUEUE
```

You could use another local queue instead of the SYSTEM.DEFAULT.LOCAL.QUEUE.

You could also specify the queue name as `exampleQM+SYSTEM.DEFAULT.LOCAL.QUEUE`, where `exampleQM` is the name of the queue manager. If the name of the queue manager is not specified, the local queue manager is used.

Both entries could be added at the same time by typing:

```
java  examples.jms.CreateJNDIEntry
    -url file://d:/MQe/data/jndi -qcf ivtQCF
d:\MQe\exampleQM\exampleQM.ini -q ivtQ
        SYSTEM.DEFAULT.LOCAL.QUEUE
```

Again, you should type all of this command on one line. A maximum of one connection factory and one queue can be added at a time.

When the JNDI entries have been created, you can run the `example.jms.MQeJMSIVT_JNDI` program. This requires the same jar files on the classpath as the MQeJMSIVT program, that is:

- `jms.jar`, Sun's interface definition for the JMS classes
- `MQeJMS.jar`, the WebSphere MQ Everyplace implementation of JMS
- `MQeBase.jar`
- `MQeExamples.jar`

It also requires the JNDI jar files, as used for the `CreateJNDIEntry` example program. The example can be run from the command line by typing:

```
java  examples.jms.MQeJMSIVT_JNDI
     -url<providerURL>
```

where <providerURL> is the specified URL of the JNDI initial context. By default the program uses the file system context for JNDI:

```
com.sun.jndi.fscontext.RefFSContextFactory
```

If necessary you can specify an alternative context:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
        -icf<initialContextFactory>
```

You can optionally add a `-t` flag to turn tracing on:

```
java  examples.jms.MQeJMSIVT_JNDI -url<providerURL>
        -icf<initialContextFactory> -t
```

To use the entries in the file system directory created in the CreateJNDIEntry example above, type:

```
java examples.jms.MQeJMSIVT_JNDI -url file://d:/MQe/data/jndi
```

The example program checks that the required jar files are on the classpath by checking for classes that they contain. It looks up the QueueConnectionFactory and the Queue in the JNDI directory. It starts a connection, which starts the WebSphere MQ Everyplace queue manager, sends a message to the Queue, reads the message back and compares it to the message it sent. The queue should not contain any messages before running the program, otherwise the message read back will not be the one that the program sent. The first lines of output from the program should look like this:

```
using context factory
   'com.sun.jndi.fscontext.RefFSContextFactory' for the directory
using directory url 'file://d:/MQe/data/jndi'
```

```
checking classpath
found JMS interface classes
found MQe JMS classes
found MQe base classes
found jndi.jar classes
found com.sun.jndi.fscontext.RefFSContextFactory classes
Looking up connection factory in jndi
Looking up queue in jndi
Creating connection
```

The rest of the output should be similar to that from the example without JNDI. You can also run the two other example programs, `examples.jms.PTPSample01` and `example .jms.PTPSample02`, using JNDI. These programs requires the same JMS and WebSphere MQ Everyplace jar files on the classpath as the `MQeJMSIVT_JNDI` program, that is:

- `jms.jar`
- `MQeJMS.jar`
- `MQeBase.jar`
- `MQeExamples.jar`

They also require the jndi.jar file and the jar files for the JNDI provider you are using, for example, file system or LDAP. The examples can be run from the command line by typing:

```
 java  examples.jms.PTPSsample01 -url<providerURL>
```

As in the previous example, `providerURL` is the URL of the JNDI initial context. By default, the program uses the file system context for JNDI, that is `com.sun.jndi.fscontext.RefFSContextFactory`. If necessary you can specify an alternative context:

```
java examples.jms.PTPSsample01 -url<providerURL>
        -icf<initialContextFactory>
```

You can optionally add a ″-t″ flag to turn tracing on: java examples.jms. PTPSsample01 -url <providerURL><-icf initialContextFactory> -t . To use the entries in the file system directory created in the CreateJNDIEntry example above, you would type:

```
java examples.jms.PTPSample01 -url file://d:/MQe/data/jndi
```

The program `examples.jms.PTPSample02` uses message listeners and filters. It creates a `QueueReceiver` with a filter ″colour=′blue′″ and creates a message listener for it. It creates a second `QueueReceiver` with a filter ″colour=′red′″ and also creates a message listener. It sends four messages to a queue, two with the property ″colour″ set to ″red″ and two with the property ″colour″ set to ″blue″, and checks that the message listeners receive the correct messages. The program has the same command line parameters as the `PTPSample01` program and can be run in the same way. Simply substitute `PTPSample02` for `PTPSample01`.

Mapping JMS messages to WebSphere MQ Everyplace messages

This section describes how the JMS message structure is mapped to a WebSphere MQ Everyplace message. It is of interest to programmers who wish to transmit messages between JMS and traditional WebSphere MQ Everyplace applications.

As described earlier, the JMS specification defines a structured message format consisting of a header, three types of property and five types of message body, while WebSphere MQ Everyplace defines a single free-format message object, `MQeMsgObject`. WebSphere MQ Everyplace defines some constant field names that messaging applications require, for example UniqueID, MessageID, and Priority, while applications can put data into a WebSphere MQ Everyplace message as `<name, value>` pairs.

To send JMS messages using WebSphere MQ Everyplace, we define a constant format for storing the information contained in a JMS message within an `MQeMsgObject`. This adds three top-level fields and four `MQeFields` objects to an `MQeMsgObject`, as shown in the following example.

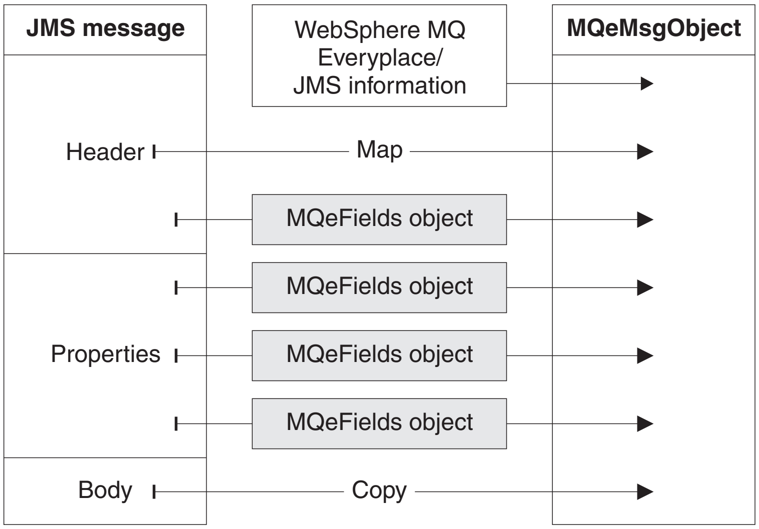


Figure 8. Mapping a JMS message to a WebSphere MQ EveryplaceMQeMsgObject

The following sections describe the contents of these fields.

Naming MQeMsgObject fields

An `MQeMsgObject` stores data as a `<name, value>` pair. The field names used to map JMS message data to the MQeMsgObject are defined in com.ibm.mqe.MQe and com.ibm.mqe.jms.MQeJMSMsgFieldNames:

MQeJMS field names

```
MQe.MQe_JMS_VERSION
MQeJMSMsgFieldNames.MQe_JMS_CLASS
```

JMS message field names

```
MQeJMSMsgFieldNames.MQe_JMS_HEADER
MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES
MQeJMSMsgFieldNames.MQe_JMS_BODY
```

JMS header field names

```
MQeJMSMsgFieldNames.MQe_JMS_DESTINATION
MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE
MQeJMSMsgFieldNames.MQe_JMS_MESSAGEID
MQeJMSMsgFieldNames.MQe_JMS_TIMESTAMP
MQeJMSMsgFieldNames.MQe_JMS_CORRELATIONID
MQeJMSMsgFieldNames.MQe_JMS_REPLYTO
MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED
MQeJMSMsgFieldNames.MQe_JMS_TYPE
MQeJMSMsgFieldNames.MQe_JMS_EXPIRATION
MQeJMSMsgFieldNames.MQe_JMS_PRIORITY
```

WebSphere MQ Everyplace JMS information

Two <name, value> pairs holding information required for WebSphere MQ Everyplace to recreate the JMS message are added directly to the `MQeMsgObject`:

`MQe.MQe_JMS_VERSION`
This contains a *short* describing the version number of the WebSphere MQ Everyplace JMS implementation used to store the message. The current version number is 1. The presence or absence of a field named `MQe.MQe_JMS_VERSION` is used to determine if an `MQeMsgObject` contains a WebSphere MQ Everyplace JMS message.

MQeJMSMsgFieldNames.MQe_JMS_CLASS
This contains a *String* describing the type of JMS message body stored in the `MQeMsgObject`. It defines the strings in Table 24.

Table 7. Strings in MQeJMSMsgFieldNames.MQe_JMS_CLASS

JMS message type	MQe.MQe_JMS_CLASS
Bytes message	jms_bytes
Map message	jms_map
Null message	jms_null
Object message	jms_object
Stream message	jms_stream
Text message	jms_text

JMS header files

JMS Header fields are stored within an `MQeMsgObject` using the following rules:

1. If a JMS header field is identical to a defined `MQeMsgObject` field then the header value is mapped directly to the appropriate field in the `MQeMsgObject`.
2. If a JMS header field does not map directly to a defined field but can be represented using existing fields defined by WebSphere MQ Everyplace then the JMS header value is converted as appropriate and then set in the `MQeMsgObject`.
3. If WebSphere MQ Everyplace has not defined an equivalent field by then, the header field is stored within an MQeFields object, which is then embedded in the `MQeMsgObject`. This ensures that the JMS header field in question can be restored when the JMS message is recreated.

The header fields that map directly to `MQeMsgObject` fields are:

Table 8. Header fields that map directly to MQeMsgObject fields

JMS header field	`MQeMsgObject` defined field
JMSTimestamp	MQe.Msg_Time
JMSCorrelationID	MQe.Msg_CorrelID
JMSExpiration	MQe.Msg_ExpireTime
JMSPriority	MQe.Msg_Priority

Two JMS header fields, `JMSReplyTo` and `JMSMessageID`, are converted prior to being stored in MQeMsgObject fields.

`JMSReplyTo` is split between `MQe.Msg_ReplyToQMgr` and `MQe.Msg_ReplyToQ`, while `JMSMessageID` is the `String "ID:"` followed by a 24-byte hashcode generated from a combination of `MQe.Msg_OriginQMgr and MQe.Msg_Time`.

The remaining four JMS header fields, `JMSDeliveryMode`, `JMSRedelivered`, and `JMSType` have no equivalents in WebSphere MQ Everyplace. These fields are stored within an MQeFields object in the following manner:

- As an int field named `MQe.MQe_JMS_DELIVERYMODE`
- As a boolean field named `MQe.MQe_JMS_REDELIVERED`
- As a String field named `MQe.MQe_JMS_JMSTYPE`

This MQeFields object is then stored within the `MQeMsgObject` as `MQe.MQe_JMS_HEADER`. Finally, JMSDestination is recreated when the message is received and, therefore does not need to be stored in the `MQeMsgObject`.

JMS properties

When storing JMS property fields in an `MQeMsgObject`, the <name, value> format used by the JMS properties corresponds very closely to the format of data in an `MQeFields` object:

Table 9. JMS property fields and the MQeFields object

Property type	Corresponding MQeFields object
Application-specific	`MQe.MQe_JMS_PROPERTIES`
Standard (`JMSX_name`)	`MQe.MQe_JMSX_PROPERTIES`
Provider-specific (`JMS_provider_name`)	`MQe.MQe_JMS_PS_PROPERTIES`

Three `MQeFields` objects, corresponding to the three types of JMS property, application-specific, standard, and provider-specific are used to store the <name, value> pairs stored as JMS message properties.

These three `MQeFields` objects are then embedded in the `MQeMsgObject` with the following names:

- `MQe.MQe_JMS_PROPERTIES`, application-specific
- `MQe_MQe_JMSX_PROPERTIES`, standard properties
- `MQe.MQe_JMS_PS_PROPERTIES`, provider-specific

Note that WebSphere MQ Everyplace does not currently set any provider specific properties. However, this field is used to enable WebSphere MQ Everyplace to handle JMS messages from other providers, for example WebSphere MQ.

JMS message body

Regardless of the JMS message type, WebSphere MQ Everyplace stores the JMS message body internally as an array of bytes. For the currently supported message types, this byte array is created as follows:

Table 10. JMS message body

JMS message type	Conversion
Bytes message	ByteArrayOutputStream.toByteArray();
Object message	<serialized object>.toByteArray();
Text message	String.getBytes(″UTF-8″);

When the JMS message body is stored in an `MQeMsgObject`, this *byte* array is added directly to the `MQeMsgObject` with the name `MQe.MQe_JMS_BODY`.

The following code fragment creates a WebSphere MQ Everyplace JMS text message by adding the required fields to an `MQeMsgObject`:

```
// create an MQeMsgObject
 MQeMsgObject msg = new MQeMsgObject();

 // set the JMS version number
 msg.putShort(MQe.MQe_JMS_VERSION, (short)1);
 // and set the type of JMS message this MQeMsgObject contains
 msg.putAscii(MQeJMSMsgFieldNames.MQe_JMS_CLASS, "jms_text");

 // set message priority and exipry time - these are mapped to
  JMSPriority and JMSExpiration
 msg.putByte(MQe.Msg_Priority, (byte)7);
 msg.putLong(MQe.Msg_ExpireTime, (long)0);

 // store JMS header fields with no WebSphere MQ Everyplace
  equivalents in an MQeFields object
 MQeFields headerFields = new MQeFields();
 headerFields.putBoolean(MQeJMSMsgFieldNames.MQe_JMS_REDELIVERED,
          false);
 headerFields.putAscii(MQeJMSMsgFieldNames.MQe_JMS_TYPE,
        "testMsg");
 headerFields.putInt(MQeJMSMsgFieldNames.MQe_JMS_DELIVERYMODE,
 Message.DEFAULT_DELIVERY_MODE);
 msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_HEADER,
       headerFields);

 // add an integer application-specific property
 MQeFields propField = new MQeFields();
 propField.putInt("anInt", 12345);
 msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PROPERTIES,
       propField);

 // the provider-specific and JMSX properties are blank
 msg.putFields(MQeJMSMsgFieldNames.MQe_JMSX_PROPERTIES,
      new MQeFields());
 msg.putFields(MQeJMSMsgFieldNames.MQe_JMS_PS_PROPERTIES,
      new MQeFields());

 // finally add a text message body
 String msgText =
   "A test message to WebSphere MQ Everyplace JMS";
 byte[] msgBody = msgText.getBytes("UTF8");
 msg.putArrayOfByte(MQeJMSMsgFieldNames.MQe_JMS_BODY,
          msgBody);

 // send the message to a WebSphere MQ Everyplace Queue
 queueManager.putMessage(null,
          "SYSTEM.DEFAULT.LOCAL.QUEUE",
          msg, null, 0);
```

Now, we use JMS to receive the message and print it:

```
// first set up a QueueSession, then...
 Queue queue = session.createQueue
      ("SYSTEM.DEFAULT.LOCAL.QUEUE");
 QueueReceiver receiver = session.createReceiver(queue);

 // receive a message
 Message rcvMsg = receiver.receive(1000);

 // and print it out
 System.out.println(rcvMsg.toString());
```

This gives:

```
  HEADER FIELDS
 -----------------------------
  JMSType:          testMsg
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      7
  JMSMessageID:     ID:00000009524cf094000000f07c3d2266
  JMSTimestamp:     1032876532326
 JMSCorrelationID: null
 JMSDestination:   null:SYSTEM.DEFAULT.LOCAL.QUEUE
  JMSReplyTo:       null
  JMSRedelivered:   false

  PROPERTY FIELDS (read only)
 ------------------------------
  JMSXRcvTimestamp : 1032876532537
  anInt : 12345

  MESSAGE BODY (read only)
------------------------------------------------------------------
 A test message to WebSphere MQ Everyplace JMS
```

Note that JMS sets some of the JMS message fields, for examole `JMSMessageID`, `JMSXRcvTimestamp` internally.

WebSphere MQ Everyplace JMS classes

WebSphere MQ Everyplace classes for Java Message Service consist of a number of Java classes and interfaces that are based on the Sun javax.jms package of interfaces and classes. They are contained in the com.ibm.mqe.jms package. The following classes are provided:

Table 11. WebSphere MQ Everyplace JMS classes

Class	**Implements**
MQeBytesMessage	BytesMessage
MQeConnection	Connection
MQeConnectionFactory	ConnectionFactory
MQeConnectionMetaData	ConnectionMetaData
MQeDestination	Destination
MQeJMSEnumeration	Java.util.Enumeration from QueueBrowser
MQeJMSJNDIQueue	Queue
MQeJMSQueue	Queue
MQeMessage	Message
MQeMessageConsumer	MessageConsumer
MQeMessageProducer	MessageProducer
MQeObjectMessage	ObjectMessage
MQeQueueBrowser	QueueBrowser
MQeQueueConnection	QueueConnection
MQeJNDIQueueConnectionFactory	QueueConnectionFactory
MQeQueueConnectionFactory	QueueConnectionFactory
MQeQueueReceiver	QueueReceiver
MQeQueueSender	QueueSender

Table 11. WebSphere MQ Everyplace JMS classes (continued)

Class	Implements
MQeQueueSession	QueueSession
MQeSession	Session
MQeTextMessage	TextMessage

Note that MessageListener and ExceptionListener are implemented by applications.

Chapter 10. Error and error handling

This chapter describes what happens if an error occurs within the Java and C codebases, under the following headings:

- Error handling in the Java codebase
- Error handling in the C codebase

Error handling in the Java codebase

Errors within the Java codebase are handled using exceptions. The WebSphere MQ Everyplace Java Programming Reference documents all of the exception codes that the WebSphere MQ Everyplace Java code can return in the following classes:

- com.ibm.mqe.MQeExceptionCodes
- com.ibm.mqe.mqbridge.MQeBridge.ExceptionCodes

Error handling in the C codebase

The C codebase indicates errors using *Return* and *Reason* codes. The C code does not have any exception handling mechanism, as in C++. WebSphere MQ Everyplace does not use the operating system error handling functions. An `MQeExceptBlock` handles errors and returns values from the functions. An application is free to install any operating system exception handlers that it requires.

The specific nature of an error condition is returned using two values, `MQERETURN` and `MQEREASON`. `MQERETURN` determines the general area in which the application failed, and distinguishes between warnings and errors. You can ignore warnings, but you must not ignore errors. With errors, your application needs to solve the problem in order to continue safely.

`MQERETURN` and `MQEREASON` are both returned in the `MQeExceptBlock`. The `MQERETURN` value is also the return value from the function.

Structure of the codes

The MQe_nativeReturnCodes.h header file lists all of the return and reason codes. They are divided into funtion area and then by error or warning. For example, `MQERETURN_QUEUE_MANAGER_ERROR` and `MQERETURN_QUEUE_MANAGER_WARNING`. Warnings indicate that a situation may be ignored.

Exception block

The `MQeExceptBlock` structure is used to pass the return code and reason code, generated by a function call, back to the user. If a function call does not return `MQERETURN_OK`, use the ERC macro to get the reason code.

WebSphere MQ Everyplace ships two macros:

EC	This macro resolves to the return code in the exception block structure.
ERC	This macro resolves to the reason code in the exception block structure.

The convention within WebSphere MQ Everyplace is that a pointer to an exception block is passed first on a new function. A pointer to the object handle is passed second, followed by any additional parameters. On subsequent calls, the object handle is the first parameter passed, and the pointer to the exception block is second, followed by any additional parameters.

The structure of the exception block, as shown in the following example, is MQeExceptBlock_st.

```
struct MQeExceptBlock_st
 {
   MQERETURN  ec;
      /* return code*/
   MQEREASON  erc;
      /* reason code*/
   MQEVOID*   reserved;
      /* reserved for internal use only*/
 }
```

Obtaining an Exception Block

We recommend that you allocate the Exception Block on the stack, rather than the heap. This simplifies possible memory allocations, although there are no restrictions on allocating space on the heap. The following code demonstates how to do this:

```
MQERETURN rc
MQeExceptBlock exceptBlock;
/*.....initialisation*/
rc = mqeFunction_anyFunction(&exceptBlock,
/*parameters go here*/);
if (MQERETURN_OK ! = rc) {
printf("An error has occured, return code =
   %d, reason code =%d \n",
   exceptBlock.ec exceptBlock.erc);
}else {
}
```

Using exception blocks

All API calls need to take exception blocks. The C Bindings codebase permits NULL to be passed to an API call. However, this feature is deprecated in the C codebase and, therefore, not recommended.

You should use a different exception block for each thread in the application.

Note: If an error is not corrected, subsequent API calls can put the system in an unpredictable state.

Useful macros

A number of macros help to access the exception block:

SET_EXCEPT_BLOCK
Sets the return and reason codes to specific values, for exampe:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK(&exceptBlock,
      MQERETURN_OK,
      MQEREASON_NA);
```

SET_EXCEPT_BLOCK_TO_DEFAULT
Sets return and reason codes to non-error values, for example:

```
MQeExceptBlock exceptBlock;
SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
```

EC Accesses the return code, for example:

```
MQeExceptBlock exceptBlk;
/*WebSphere MQ Everyplace API call */
MQERETURN returncode;
returnCode = EC(&exceptBlock);
```

ERC Accesses the reason code, for example:

```
MQeExceptBlock exceptBlk;
/*WebSphere MQ Everyplace API call*/
MQEREASON reasoncode;
MQEREASON reasonCode = ERC(&exceptBlock);
```

NEW_EXCEPT_BLOCK
Can create a temporary exception block. This is useful for temporary clean-up operations, for example:

Chapter 11. Deployment of applications from Webshpere Studio Device Developer

This chapter describes how to develop and deploy applications to devices from WebSphere Studio Device Developer (WSDD). To fully understand the concepts outlined here we recommend that you have Java programming skills, knowledge of J2ME and MIDlets, and basic knowledge of WebSphere MQ Everyplace.

The example application aims to aid your understanding of the WebSphere MQ Everyplace interface. The code can be split into 3 parts:

The message service

This runs WebSphere MQ Everyplace, controls a queue manager and performs functions such as queue creation and message sending. This is the core of the examples and allows them to be written with minimal calls to the WebSphere MQ Everyplace API. This also means that to see the code required to create a local queue for example, a user can simply look at the relevant function within MQeMessageService.

Example 1: The message pump

This is a very simple application consisting of a single server and client. The client is set to send a message to the server every 3 seconds which, when received by the server, will be displayed to the user. Queues are asynchronous. Implementations of the client are available for both MIDP and J2SE, while the server is only available for J2SE.

Example 2: The text application

This is slightly more complex than the first example, consisting of 2 servers and a client. When initiating, the client is required to register with the registration server. The registration server adds the client to a store-and-forward queue on the gateway server and replies with a success or failure message. The client can then send user-defined messages to the gateway server (which it will display). The aim of this application is to show how a seperate server can be used to create resources necessary for a new client on the system to aid scalability of large WebSphere MQ Everyplace networks.

This chapter presents information under the following headings:

- Getting started
- Debugging
- Available runnable classes
- MIDlets
- Cleaning up after applications
- Problems with SmartLinker
- Additional help

Getting started

This section details how to set up both of the devices that you use and WebSphere Studio Device Developer (WSDD) to work with those devices.

Palm: What you need to get started

The following are prerequisites required for writing and testing WebSphere MQ Everyplace applications for the Palm:

- A Palm device or Palm Emulator (you can download POSE from http://www.palmos.com/dev/tools/emulator/)
- A copy of a J2ME virtual machine installed on the Palm, for example the Sun's K Virtual Machine (KVM) and IBM's J9, available from http://java.sun.com and http://www.embedded.oti.com
- A cradle to synchronize the palm with your pc
- Something to generate .prc files, that is palm executables, to run on the plam, such as Sun's J2ME Wireless Toolkit (available at http://java.sun.com) and IBM's WSDD, available at http://www.embedded.oti.com, which includes J9 as standard
- WebSphere MQ Everyplace JARs/classes

This documentation concentrates on J9 and WSDD.

Palm: Getting started with WSDD

You must complete several tasks before using the palm device or palm emulator to run WebSphere MQ Everyplace MIDlets:

1. Get the virtual machine onto the unit. The .prc files required for this are located in the C:\IBM\wsdd\wsdd4.0\ive\runtimes\palmos\68k\ive\bin directory or the equivilant location for your installation. You need the following files:
   - j9_vm_bundle.prc
   - j9pref.prc
   - midp15.prc
   - j9_dbg_bundle.prc (only if you are planning to debug an application)
2. Once you have installed these files on your palm device (it should come with instructions on how to do this), use WSDD to create a new MIDlet suite (in the java perspective - [File][New][Other][J2ME for J9][Create MIDlet Suite]).
3. Import the source for the example application into the src directory. Include the WebSphere MQ Everyplace library in the list of libraries to use, that is right-click the name of the project in the packages window and select [Properties] [Java Build Path] and the Libraries tab. Use the 'Add External JARs' option to add the WebSphere MQ Everyplace MIDP jar to the list. Note the following files are not meant for use under MIDP:
   - mqeexampleapp.msgpump.NormalClient
   - mqeexampleapp.msgpump.NormalServer
   - mqeexampleapp.msgpump.InputThread
   - mqeexampleapp.textapp.Client
   - mqeexampleapp.textapp.GatewayServer
   - mqeexampleapp.textapp.RegistrationServer

   These should be run in Foundation or J2SE to act as commandline implementations of the clients and servers. There are no MIDP servers as it is not an environment that servers are designed to run in.
4. Set WSDD to run files on that device. With normal palms, an installation program is provided to enable the installation of new programs from a desktop computer (e.g. C:\Palm\Instapp.exe). This needs to be set in WSDD in [Window][Preferences][Device Developer][PalmOS Java Configuration] under PalmOS Install Tool. You also meed to set the other options in this menu:

PalmOS Emulator
This is required if you want to use POSE or a similar PalmOS emulator

PilRC resource compiler
This creates the PRC files from the jad and jar. The WSDD help describes the java options in more detail.

Palm: Building for the Palm in WSDD

Once WSDD has been set up to work with the palm, try building and running the example application on your palm device. To do this:

1. Double-click the wsddbuild.xml file from within your project. If you created a J2ME for J9 project and not a normal java one, it will appear after all the packages.
2. Select the builds tab from the bottom of the window. Currently, your list of builds should be empty. This window specifies the platforms you are building the project for, that is palm, pocketpc, windows, and so on.
3. Click **Add Build** and select the palm option from the pulldown platforms menu.
4. Click **Next** and enter any creator ID and a name for the application.
5. Click **Next** again until you reach the final *select launcher* screen. If you are using a palm device, select the manual option. If you are using the emulator, select the emulator option.
6. Click **Finish** and select the launch tab. Your device should now be a launch option.

PocketPC: What you need to get started

To run WebSphere MQ Everyplace applications on the PocketPC you need:

- A Pocket PC device. Emulators exist, but they are not as true to the original device as the Palm emulators are to the Palm.
- A copy of a J2ME virtual machine installed on the device. A cradle to sync the pocket pc with your desktop
- J9 for the Pocket PC comes with WSDD and is located in C:\IBM\wsdd\wsdd4.0\ive\runtimes\pocketpc\arm\ive. The files required from here are:
  - bin\iverel15.dll
  - bin\j9.exe
  - bin\j9dbg15.dll
  - bin\j9dyn15.dll
  - bin\j9hook15.dll
  - bin\j9midp15.dll
  - bin\j9prt15.dll
  - bin\j9thr15.dll
  - bin\j9vm15.dll
  - bin\j9w.exe
  - bin\j9zlib15.dll
  - bin\swt-win32-ce-2023.dll
  - lib\jclMidp
  - lib\jclMidp.jxe

These are specified in the WSDD help file. Create a similar directory structure on the device, for example, program files or wsdd with bin and lib subdirectories.

Then copy the files to the relevant places. Note that the example application functions under MIDP,hence the need for the jclMidp.jxe file. The Palm: What you need to get started section provides details on downloading WSDD.

PocketPC: Getting started with Websphere Studio Device Developer

To run applications on the Pocket PC from WSDD, you need to tell WSDD where the various files you copied to your device are located. This is done in [Window][Preferences][Device Developer][PocketPC Java Configuration]. Set the three options to: \Program Files\WSDD \My Documents\WSDD \Windows\Start Menu, assuming that you copied the J9 files to '\Program Files\WSDD' earlier

PocketPC: Building for the Pocket PC in Websphere Studio Device Developer

This procedure is almost identical to that described in the Building for the Palm in WSDD section. However, with the final choice for launcher, choose 'MIDlet Suite on PocketPC Device' rather than the manual option. This means that the application automatically copies to the relevant device and runs automatically..

Debugging

This section describes how to set the example application debugging using WSDD, both locally and remotely, on various devices.

Debugging on the Palm using WSDD

To debug on the palm using WSDD:

1. Install the j9_dbg_bundle.prc file on the target device before attempting to debug. This is located in C:\IBM\wsdd\wsdd4.0\ive\runtimes\palmos\68k\ive\bin.
2. On on the target device, run prefs and navigate to the J9 Java VM section. Ensure that 'Enable Debug' is selected, otherwise you cannot debug an application.
3. When the application launches (via wsddbuild.xml - launches), select **debug** rather than **run**.

Debugging on the PocketPC using WSDD

The files specified in the PocketPC: What you need to get started section include the necessary components to debug remotely. Simply launch the application using the debug command rather than the run command, as described in Debugging on the palm using WSDD.

Debugging locally using WSDD

Select **debug** rather than **run** to start the application.

Available runnable classes

The following classes can be run the prompt:

mqeexampleapp.msgpump.NormalClient
A J2SE client for the Message Pump

mqeexampleapp.msgpump.NormalServer
A J2SE server for the Message Pumpv

mqeexampleapp.textapp.Client
A J2SE client for the Text App

mqeexampleapp.textapp.RegServer
A J2SE registration server for the Text App

mqeexampleapp.textapp.GatewayServer
A J2SE gateway server for the Text App

MIDlets

The following MIDlets are available in this example application:

mqeexampleapp.msgpump.MidpClient
The MIDP client for the Message Pump

mqeexampleapp.textapp.MidpClient
The MIDP client for the Text App

mqeexampleapp.messageservice.RMSclea
A simple utility to clear all RMS stores within the MIDlet suite

Giving parameters to the MIDlet

A useful feature of MIDlets is that they can retrieve parameters from their jad file. The example applications take advantage of this to allow simple changes to the MIDP clients without having to alter the code. Unfortunately, you cannot perform the necessary changes to the jad file in WSDD. To use this feature, open the jad of your project in a text editor. User defined parameters are specified as follows:

```
parameter: value
```

Use the following parameters for the two example applications:

1. MsgPump Client

   Pump_SecurityLevel
   Specifies the security level that the application should use:
   - 0 for no security
   - 1 for message based security
   - 2 for queue based security

   Pump_ServerQueue
   Specifies the name of the queue that messages should be sent to

   Pump_ServerIP
   Specifies the IP of the server that will sent messages

   Pump_ServerPort
   Specifies the port that the server will be listening on

   Pump_ServerQueueManager
   Specifies the name of the queue manager that messages will be sent to
2. TextApp Client

   pp_Registration_ServerIP
   The IP address of the registration server

   App_Gateway_ServerIP
   The IP address of the gateway server

An example jad file may look something like this:

```
MIDlet-Version:
MIDlet-Name: ExampleAppv2NewestMQe MIDlet-Jar-Size:
MIDlet-Jar-URL:
MIDlet-1: pumpclient,,mqeexampleapp.msgpump.MidpClient
```

```
MIDlet-2: clear,,mqeexampleapp.messageservice.RMSclear
MIDlet-3: textapp,,mqeexampleapp.textapp.MidpClient
MIDlet-Vendor:

Pump_SecurityLevel: 0
Pump_ServerQueue: A_Queue
Pump_ServerPort: 8083
Pump_ServerQueueManager: QM_Mr_Server
Pump_ServerIP: 10.0.0.101

App_Registration_ServerIP: 10.0.0.100
App_Gateway_ServerIP: 10.0.0.131
```

Any value not specified in this manner defaults to its usual value.

Cleaning up after the applications

As with all WebSphere MQ Everyplace queue managers, registries and messages are left on the system after the queue manager has been shut down. This design allows queue managers to restart witout losing their messages or recreating all their queues and registry settings. If, one of the examples crashes on starting, the data they leave behind should automatically be removed to prevent them from being restarted with an incomplete registry. If the example does not crash, or you wish to start the queue manager from scratch, you can use the following methods to remove the registry from the system:

J2SE The example application uses the MQeDiskFieldsAdapter, which will save registry settings to the Hard Drive. These are located in c:/MQe/QM/. Delete this directory to remove the remaining information.

MIDP The example application uses the MQeMidpFieldsAdapter for MIDP environments. This means that the registry is stored in the record stores of the MIDlet Suite. You can remove them using the RMSclear MIDlet located in the exampleapp.messageservice package.

Note: If you delete a MIDlet Suite from a device, its record store is also removed.

Problems with SmartLinker

A program called SmartLinker is used to strip unused classes and methods before packing a project into a .jxe file. Although this gives the benefit of a much smaller application, it also causes dynamically loaded classes to be stripped from the application when the .jxe is built.

An example of this is the various adapters that are dynamically loaded for different environments. Because these adapters are not explicitly refered to anywhere in the code, they are removed and so a NoClassDefFoundException is thrown.

The cleanest way to solve this problem is to specify in the jxeLinkOptions file that you wish to include a specific class. You can do this in WSDD in the following manner:

1. In the packages view of your project, open the directory for the device you are creating the jxe for (e.g. palm68k) and open the jxeLinkOptions file (e.g. ExampleApp.jxeLinkOptions.

2. Select the in or exclusion tab and pic [include whole classes] from the pulldown menu. This screen shows all the classes that the user has specified will definitely be included.
3. To add a new class to the list, select [new] and enter the class in the [Rule pattern] box, for example . com.ibm.mqe.adapter.MQeMidpFieldsAdapter. The following files require inclusion in this manner for the MIDP clients to work:
   - mqeexampleapp.messageservice.QueueManagerRules
   - com.ibm.mqe.adapters.MQeMidpFieldsAdapter
   - com.ibm.mqe.adapters.MQeMidpHttpAdapter
   - com.ibm.mqe.MQeAttributeRule
   - com.ibm.mqe.messagestore.MQeMessageStore
   - com.ibm.mqe.registry.MQeFileSession

Additional help

WSDD help provides additional information under: Websphere Studio Device Developer Product Documentation\WSDD Product Documentation \Tasks\Working with Palm OS Targets

Chapter 12. Open Services Gateway Initiative

Open Services Gateway Initiative (OSGi) is an application framework capable of deploying java applications or services, which can be dynamically employed, updated, or removed. Therefore, it can be a very useful means for providing service updates and ensuring that all the required classes for an application are made available as and when required. WebSphere MQ Everyplace provides an example bundle that provides WebSphere MQ Everyplace messaging within this framework.

In this chapter, WebSphere MQ Everyplace and OSGi are explained under the following headings:

- WebSphere MQ Everyplace example bundle contents
- Using WebSphere MQ Everyplace within OSGi
- Running the example bundles
- Providing user-defined Rules and dynamic class loading

WebSphere MQ Everyplace example bundle contents

WebSphere MQ Everyplace provides one main bundle for OSGi development and two example application bundles that provide hints on how to create a WebSphere MQ Everyplace client or server application within OSGi. No bundle exports or imports a service; they all rely on package dependency. The following table details the bundles and their dependencies.

Table 12. Bundles and dependencies

Bundle name	Description	Export packages	Import packages
MQeBundle.jar	Bundle containing all the required WebSphere MQ Everyplace classes excluding mqbridge functionality	com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.attributes com.ibm.mqe.communications com.ibm.mqe.messagestore com.ibm.mqe.mqemqmessage com.ibm.mqe.registry com.ibm.mqe.trace	
MQeServerBundle.jar	Example bundle containing a WebSphere MQ Everyplace Server application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework
MQeClientBundle.jar	Example bundle containing a WebSphere MQ Everyplace Client application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework

Both example application bundles, MQeClientBundle.jar and MQeServerBundle.jar contain bundle activators which start and stop the application when the framework starts or stops the bundle. The bundles are in MQE_HOME/Java/Jars.

Using WebSphere MQ Everyplace within OSGi

When developing your own bundles, importing the correct WebSphere MQ Everyplace packages into your bundles manifest file ensures that the WebSphere MQ Everyplace bundle is also installed into the framework when your bundle is installed.

One major factor in developing a bundle is that only one WebSphere MQ Everyplace queue manager is able to be run within an OSGi runtime. This means that there may be conflicts if several bundles are installed and each requires its own queue manager. Careful design of the bundle application is required to eliminate this problem. However, there should be no limit on the number of bundles that can use the same queue manager.

Running the Example Bundles

As an example of how to use WebSphere MQ Everyplace within the OSGi environment, we providedtwo example application bundles that are designed to work together in a simple scenario. The scenario consists of a client application and a server application. The Server simply sits and waits for messages and prints out any that it receives, the Client just sends one message. Within this scenario it is possible to have multiple Clients sending to the same Server or the same Client can be stopped and restarted to send another message to the Server. These bundles are explained in more detail below:

Server application (MQeServerBundle.jar)

When this bundle is started an MQeQueueManager is created and started with a listener and default queues in memory. The Application code is then run in a new thread and waits for incoming messages using a message listener; any received messages are displayed in the console. This thread continues to listen until the bundle is stopped, at which time it stops and then deletes the MQeQueueManager.

Client Application (MQeClientBundle.jar)

When this bundle is started it checks to see if an MQeQueueManager is already running in the JVM, if so it assumes it is running in the same runtime as the server and so uses that queue manager. If no queue manager is detected then a new one is defined and started in memory and a connection definition and remote queue definition are setup to the server. Client application code is then run in a new thread which sends a single message to the server. No checks are made to ensure the message is received. If we created a new QueueManager for the Client, it is stopped and deleted when the bundle is stopped.

The source for the classes included in the bundles can be seen in the MQe\Java\examples\osgi directory. More details are given in the java doc for these classes.

Some points to note when running the applications:

- Each application was written with two parts in mind. The first is setup of the underlying WebSphere MQ Everyplace messaging infrastructure and the second is the main application. This is why each one has a separate class providing function for each part.
- The MQeClientBundle.class and MQeServerBundle.class are both started in their own threads by the bundle activator start method. This way the start method is

not delayed in completing as the tasks of sending and receiving messages can take some time. This ensures a smooth transition of the bundles state from resolved to started.

Note: The Client and Server share the same MQeAdmin class in their bundles. This class could have been placed in its own bundle to avoid the duplication but for simplicities sake we have not done this.

- The Server must always be started before any Clients. Each Server must run in its own runtime. A single client can share the server's runtime or can reside in its own.

Running the example

However, you run the examples the following bundle is required by both the client and server and must be present on the bundle server. See the Configuration Guide for details on submitting bundles to a server.

MQeBundle.jar

To run the example you will need to first start the Server:

1. Import the MQeServerBundle.jar bundle onto the Bundle Server.
2. Start a new SMF runtime and Install and start the MQeServerBundle bundle on it. This should also install the three prerequisite bundles.
3. The server then starts listening, you should see output on the console including:

   ```
   'MQeServerBundle - registering message listener'
   ```

   This means the server is ready for messages.

Next you need to run a client to send a message. There are two methods for runnning the client bundle:

Method 1

In the same SMF runtime as the server:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Install and start the MQeClientBundle bundle on the runtime.
3. The client now starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

Method 2

In separate SMF runtimes:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Start a new SMF runtime and Install and start the MQeClientBundle bundle on it. This should also install the three prerequisite bundles.
3. The client starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

By default the example expects both client and server to be on the same machine running with the receiver listening on port 8085. However, you can change the port and address of the server, that is run the server on a separate machine. Before the server is started, you can tell it which port to run on by setting the java system property, ″examples.osgi.server.port″. This can be set in the Runtime IDE by selecting ″show runtime properties″ from the drop down menu.

To tell the client the address and port that the server is listening on, set the ″examples.osgi.server.address″ and ″examples.osgi.server.port″ system properties before the client is started.

Note: The server ignores the address property if it is not present. Also, if the client has already been run and you want to change the address and port, the runtime needs to be terminated and restarted to ensure that old MQeConnectionDefinition information is wiped from memory.

Providing user-defined rules and dynamic class loading

The OSGi runtime controls package visibility across bundles. If a bundle does not explicitly import a package, then it will not have access to classes within that package when it comes to dynamically loading them. This is especially important to WebSphere MQ Everyplace, because it has been designed with this flexibility in mind. Without some small changes to the bundles, developers cannot use 3rd party or their own Rules or Adapters. There are two ways to remove this problem:

1. OSGi version 3 includes a DynamicImport-Package statement for the bundles manifest file. This has been included in the MQeBundle.jar and as long as the user-defined class’s package is exported from its bundles manifest. WebSphere MQ Everyplace will be able to have access to this class.

   Note: This functionality is available to SMF version 3.1.0 or higher.
2. 2. Create a new MQeLoader and add all the user-defined classes before they are used, most likely within the bundles activator, for example:

   ```
   String MyRule = "UserQMRule";
   MQeLoader loader = new MQeLoader();
   loader.addClass(MyRule, Class.forName(MyRule));
   MQe.setLoader(loader);
   ```

Take care when using the second method, that the loader within WebSphere MQ Everyplace is not replaced with another loader from another bundle during the application runtime.

Appendix A. WebSphere MQ Everyplace Java programming examples

This appendix provides WebSphere MQ Everyplace programming examples for java developpers.

Examples

The examples previously described form a small part of the set of examples provided with WebSphere MQ Everyplace. Each example demonstrates how to use or extend a feature of WebSphere MQ Everyplace. Most are described in the relevant sections of this Guide. They are all listed and briefly described in the following sections

examples.adapters

This package provides two example classes that conform to the MQSeries Everyplace adapters specification.

MQeDiskFieldsAdapter
This example class is identical in functionality to the disk fields adapter found in com.ibm.mqe.adapters. It supports the reading and writing of data on the local file store.

WESAuthenticationGUIAdapter
Wrappers the WESAuthenticationAdapter found inside com.ibm.mqe.adapters. This example enhances the WESAuthenticationAdapter by displaying a dialog box that prompts the user for login information when connecting to a Websphere Everyplace proxy.

See Chapter 1, *Adapters* of the WebSphere MQ Everyplace System Programming Guide for more information on adapters in WebSphere MQ Everyplace.

examples.administration.commandline package

This package contains a suite of example tools for creating base WebSphere MQ Everyplace objects from the command line. Each program is a simple example of how to send administration messages and how to interpret the replies.

Using these tools and a script you can reliably set up exactly the same configuration on a number of machines.

examples.administration.console package

This package contains a set of classes that implement a simple graphical user interface (GUI) for managing WebSphere MQ Everyplace resources.

Admin
Front end to the example administration GUI.

Additionally there is a suite of classes that provides the graphical user interface for each WebSphere MQ Everyplace managed resource.

The GUI can be invoked in any of the following ways:

- Using the batch file ExamplesAdminConsole.bat
- From the command line:
  ```
  java examples.administration.console.Admin
  ```
- From a button on the example server `examples.awt.AwtMQeServer`

Refer to the WebSphere MQ Everyplace Configuration Guide, for more details information about using the WebSphere MQ Everyplace administration functions.

examples.administration.simple package

This package contains a set of examples that show how to use some of the administrative features of WebSphere MQ Everyplace in your programs. As with the application examples, these examples can work with either a local or a remote queue manager.

Example1
Create and delete a queue.

Example2
Add a connection definition for a remote queue manager.

Example3
Inquire on the characteristics of a queue manager and the queues it owns.

ExampleAdminBase
The base class that all administration examples inherit from.

For details of WebSphere MQ Everyplace administration functions, refer to the WebSphere MQ Everyplace Configuration Guide.

examples.application package

This package contains a set of examples that demonstrate various ways to interact with a queue manager. These include putting a message to and getting a message from a queue. All the examples can be used with either a local queue manager or a remote queue manager. Before you can use any of these applications, the queue managers that are to be used must be created. You can use the CreateExampleQM.bat batch file on Windows, or the CreateExampleQM shell script on UNIX, to create queue managers `ExampleQM` (see “Post install test” on page 22).

Example1
Simple put and get of a message.

Example2
Put several messages and then get the second one using a match field.

Example3
Use a message listener to detect when new messages arrive.

Example5
Lock messages then get, unlock, and delete them.

Example6
Simple put and get of a message using assured message delivery.

Example7
Simple put and get of a message through a Websphere Everyplace proxy.

ExampleBase
The base class that all application examples inherit from.

These examples can be run as follows:

Windows
Using batch file ExamplesMQeClientTest.bat

```
ExamplesMQeClientTest <JDK> <example no>
  <remoteQMgrName> <localQMgr ini file>
```

UNIX Using shell script ExamplesMQeClientTest

```
ExamplesMQeClientTest  <example no>
<remoteQMgrName> <localQMgr ini file>
```

where

<*JDK*> is the name of the Java environment (see "Development environment" on page 11 for details). The default is IBM

Note: This parameter is not used on UNIX.

<*example no*>
is the number of the example to run (suffix of the name of the example). The default is 1 (Example1).

<*remoteQMgrName*>
is the name of the queue manager that the application should work with. This can be the name of the local or a remote queue manager. If it is a remote queue manager, a connection must be configured that defines how the local queue manager can communicate with the remote queue manager.

By default the local queue manager is used, as defined in ExamplesMQeClient.ini.

<*localQMgrIniFile*>
is an ini file containing startup parameters for a local queue manager. By default ExamplesMQeClient.ini is used.

For more details on how to write applications that interact with a queue manager see Chapter 6, "Queue manager operations", on page 43.

examples.attributes package

This package contains a set of classes that show how to write additional components to extend WebSphere MQ Everyplace security. However, they are not designed to be used for asynchronous messaging and do not provide very strong security.

NTAuthenticator
An authenticator that authenticates a user to the Windows NT security database. To authenticate correctly the user must have the following User Rights set on the target NT system:

- Act as part of the operating system
- Logon locally
- Logon as a service

The NT authenticator uses the Java native interface (JNI) to interact with Windows NT security. The code for this can be found in the examples.nativecode directory. The dll built from this code must be placed in the PATH of the NT machine that owns the target resource.

UnixAuthenticator
An authenticator that authenticates a user using the UNIX password or shadow password system. The UNIX authenticator uses the JNI to interact with the host system. The code for this can be found in the examples.nativecode directory. If your system supports the shadow password file, you must recompile this native code with the `USE_SHADOW` preprocessor flag defined. You must also ensure the code has sufficient privileges to read the shadow password file when it executes. This example does not work if your system uses a distributed logon service (such as Lightweight Directory Access Protocol (LDAP)).

LogonAuthenticator
Base logon authentication support.

UseridAuthenticator

Support for base *`userID`* authentication.

This example requires a UserIDS.txt file as input. This file must have the format:

```
  [UserIDs]

User1Name=User1Password

...

UserNName=UserNPassword
```

See Chapter 8, “Security”, on page 79 for more detailed information about the WebSphere MQ Everyplace security features.

examples.awt package

This package provides a toolkit for building applications that require a small graphical interface. It also contains example applications that provide a graphical front end to WebSphere MQ Everyplace functions.

AwtMQeServer
A graphical front end to the `examples.queuemanager.MQeServer` example. The MQeTraceResourceGUI class provides a resource bundle that contains internationalized strings for use by the GUI. MQeTraceResourceGUI is in package examples.trace.

You can use the batch file ExamplesAwtMQeServer.bat to run this application.

See “Server queue managers” on page 49 for more details about running a queue manager in a server environment.

AwtMQeTrace
A graphical front end to examples.trace.MQeTrace.

See Chapter 9, “Java Message Service”, on page 103 for more information about the WebSphere MQ Everyplace trace facility.

Classes **AwtDialog**, **AwtEvent**, **AwtFormat**, **AwtFrame**, and **AwtOutputStream** provide a toolkit for building small footprint awt-based graphical applications. These classes are used by many of the graphical WebSphere MQ Everyplace examples.

examples.certificates package

This package contains examples for managing mini-certificates (see "Mini-certificate issuance service" on page 94) for more information on these examples, and using mini-certificates.

ListWTLSCertificates
This example uses methods in the class com.ibm.mqe.attributes.MQeListCertificates to implement a command line program which lists mini-certificates in a registry, to varying levels of detail.

RenewWTLSCertificates
This example uses methods in the class com.ibm.mqe.registry.MQePrivateRegistryConfigure to implement a command line program which renews mini-certificates in a registry. This should only be used on a private registry.

examples.eventlog package

This package contains some examples that demonstrate how to log events to different facilities.

LogToDiskFile
Write events to a disk file.

LogToNTEventLog
Write events to the Windows NT event log. This class uses the JNI to interact with the Windows NT event log. The code for this is in the examples.nativecode directory.

LogToUnixEventLog
Write events to the UNIX event log (which is normally /var/adm/messages). This class uses the JNI to interact with the UNIX event logging system. The code for this can be found in the examples.nativecode directory. The syslog daemon on your system should be configured to report the appropriate events.

examples.install package

This package contains a set of classes for creating and deleting queue managers.

DefineQueueManager
A GUI that allows the user to select options when creating to queue manager. When the options have been selected, this example creates an ini file containing the queue manager startup parameters, and then creates the queue manager.

CreateQueueManager
A GUI program that requests the name and directory of an ini file that contains queue manager startup parameters. When the name and directory are provided, a queue manager is created.

SimpleCreateQM
A command line program that takes a parameter that is the name of an ini file that contains queue manager startup parameters. It also optionally takes a parameter that is the root directory where queues are stored. Provided a valid ini file is found, a queue manager is created.

DeleteQueueManager
A GUI program that takes the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, the queue manager is deleted.

SimpledDeleteQM
A command line program that takes a parameter that is the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, the queue manager is deleted.

GetCredentials
A GUI program that takes the name of an ini file that contains queue manager startup parameters. Provided a valid ini file is found, new credentials (private/public key pair and public certificate) are obtained for the queue manager. The mini-certificate server must be running and the request for a new certificate must have been authorized for this to succeed (see "Mini-certificate issuance service" on page 94).

All the configuration files use the resources and utilities provided in **ConfigResource**, and **ConfigUtils**.

For more details about creating and deleting queue managers, see Chapter 6, "Queue manager operations", on page 43.

examples.mqbridge.awt package

This package contains a set of classes that show how to use and extend the WebSphere MQ bridge. Some of the examples extend other WebSphere MQ Everyplace examples.

AwtMQBridgeServer
This is an example of a graphical interface for the underlying examples.mqbridge.queuemanager.MQBridgeServer class.

The MQBridgeServer class source code demonstrates how to add bridge functionality to your WebSphere MQ Everyplace server program, following these guidelines.

To start the bridge enabled server:

1. Instantiate the base WebSphere MQ Everyplace queue manager, and start it running.
2. Instantiate a com.ibm.mqe.mqbridge.MQeMQBridges object, and use its activate() method, passing the same .ini file information as you passed to the base WebSphere MQ Everyplace queue manager.

The bridge function is then usable.

To stop the bridge-enabled server:

1. Disable the bridge function by calling the MQeMQBridges.close() method. This stops all the current WebSphere MQ bridge operations cleanly, and shuts down all the WebSphere MQ bridge function.
2. Remove your reference to the MQeMQBridges object, allowing it to be garbage-collected.
3. Stop and close the base WebSphere MQ Everyplace queue manager.

ExamplesAwtMQBridgeServer.bat
This file provides an example of how to invoke the MQBridgeServer using the Awt server. It also shows how to control the initial settings of the AwtMQBridgeTrace module.

ExamplesAwtMQBridgeServer.ini
This file provides an example configuration file for a queue manager that supports WebSphere MQ bridge functionality.

Refer to the WebSphere MQ Everyplace Configuration Guide for more details about the WebSphere MQ bridge.

examples.mqbridge.administration.commandline package

This package contains a suite of example tools, similar to those in the examples.administration.commandline package, designed to administer the objects required for an WebSphere MQ bridge.

examples.mqbridge.application.GetFromMQ

The example programs in this package are useful for proving that WebSphere MQ Everyplace and WebSphere MQ can communicate with each other. These examples are WebSphere MQ bindings programs that use the Java classes and are driven by a simple command-line syntax.

GetFromMQ
This class destructively reads any message appearing on a specified WebSphere MQ queue, and provides timing statistics on when the message arrives. Optionally the message content can be dumped to the standard output screen.

This example is useful when testing a link between WebSphere MQ Everyplace and WebSphere MQ, to see what throughput is being achieved between the two systems. Scripts dealing with connectivity between WebSphere MQ Everyplace and WebSphere MQ can refer to and use this class.

PutFromMQ
This class puts a message to an WebSphere MQ queue, such that the user can specify the target queue and the target queue manager. It specifically uses the long form of the **MQQueueManager.accessQueue()** method to make use of any WebSphere MQ Everyplace queue manager alias definitions that might be defined on the WebSphere MQ queue.

examples.mqeexampleapp package

This package contains two example applications to aid your understanding of the MQe interface. The example code can be split into 3 parts:

The message service (examples.mqeexampleapp.messageservice)
This runs WebSphere MQ Everyplace, controls a queue manager and performs functions such as queue creation and message sending. This is the core of the examples and allows them to be written with minimal calls to the WebSphere MQ Everyplace API. This also means that to see the code required to create a local queue for example, a user can simply look at the relevant function within MQeMessageService.

Example 1: The message pump (examples.mqeexampleapp.msgpump)
This is a very simple application consisting of a single server and client.

The client is set to send a message to the server every 3 seconds which, when received by the server, will be displayed to the user. Queues are asynchronous. Implementations of the client are available for both MIDP and J2SE, while the server is only available for J2SE.

Example 2: The text application (examples.mqeexampleapp.textapp)
This is slightly more complex than the first example, consisting of 2 servers and a client. When initiating, the client is required to register with the registration server. The registration server adds the client to a store-and-forward queue on the gateway server and replies with a success or failure message. The client can then send user-defined messages to the gateway server (which it will display). The aim of this application is to show how a seperate server can be used to create resources necessary for a new client on the system to aid scalability of large WebSphere MQ Everyplace networks.

examples.nativecode package

Several of the examples require access to operating system facilities on Windows NT, or UNIX (AIX and Solaris). WebSphere MQ Everyplace accesses these functions using the JNI. For Windows, the code in the examples\native directory provides the JNI implementation required by `examples.attributes.NTAuthenticator` and `examples.eventlog.LogToNTEventLog`. For UNIX, the code in the file examples/native/JavaUnix.c provides the JNI implementation required by the `examples.attributes.UnixAuthenticator` and `examples.eventlog.LogToUnixEventLog`.

examples.queuemanager package

A queue manager can run in many different types of environment. This package contains a set of examples that allow a queue manager to run as a client, server, or servlet:

MessageWaiter
An example of how to wait for messages without using the deprecated **waitFormessage** method.

MQeClient
A simple client typically used on a device.

MQePrivateClient
A client that can be used with secure queues and secure messaging.

MQeServer
A server that can connect concurrently to multiple queue managers (clients or servers). This is typically used on a server platform. Batch file ExamplesAwtMQeServer.bat can be used to run the `examples.awt.AwtMQeServer` example which provides a graphical front end to this server.

MQePrivateServer
Similar to MQeServer but allows the use of secure queues and secure messaging.

MQeServlet
An example that shows how to run a queue manager in a servlet.

MQeChannelTimer
An example that polls the channel manager so that it can time-out idle channels.

MQeQueueManagerUtils
A set of helper methods that configure start various WebSphere MQ Everyplace components.

For more details about running queue managers in different environments see “Starting queue managers” on page 44. For details on queue managers that provide an environment for secure queues and messaging (MQePrivateClient and MQePrivateServer), see Chapter 8, “Security”, on page 79.

examples.rules package

You can control and extend the base WebSphere MQ Everyplace functionality using rules. Some components of WebSphere MQ Everyplace invoke rules classes. These rules provide a means of changing the functionality of the component. This package contains the following example rules classes:

ExamplesQueueManagerRules
Example queue manager rules class makes regular attempts to transmit any held messages.

See Chapter 7, “Message Delivery”, on page 69 for more details.

AttributeRule
Example attribute rule that controls the use of attributes.

The ″examples.mqbridge.rules″ package of the *Programming Interface Reference for Java programmers* provides details on how to use bridge rules.

examples.trace package

This package contains an example trace handler that can be used for debugging an application during development, and for tracing a completed application.

MQeTrace
The base WebSphere MQ Everyplace trace class.

AwtMQeTrace, which is in the examples.awt package, provides a graphical front end to the MQeTrace class.

MQeTraceResource
A resource bundle that contains trace messages that can be output by WebSphere MQ Everyplace.

MQeTraceResourceGUI
This class contains all the translatable text for the trace window controls.

Appendix B. Applying maintenance to WebSphere MQ Everyplace

Maintenance updates for WebSphere MQ Everyplace are always shipped as a complete new release. There are two options when upgrading from one release to another:

Completely uninstall the current level, and install the new level in same directory

When doing this it is recommended you keep the install package for the current level to allow it to be restored later if necessary.

Keep the existing level and install the new level into a new directory

After installation, check your classpath to ensure that the latest level of WebSphere MQ Everyplace is being invoked. If installing on Windows, make sure that you give the shortcuts folder for the new install a different name to the existing one.

For more general information on maintenance updates and their availability see the WebSphere MQ family Web page at *http://www.software.ibm.com/mqseries/*.

Appendix C. Differences between trace in WebSphere MQ Everyplace version 1.2.6 or lower and version 2.0

The tracing mechanism for WebSphere MQ Everyplace version 2.0 differs from the mechanism provided by version 1 of the product. The main objectives of these changes are to:

- Allow static methods to use the WebSphere MQ Everyplace trace mechanism
- Allow dynamic filtering of trace before the expense of gathering all trace data has been spent. Version 1 collected all information to be traced, then if a trace handler wished to discard that information, the cost of its' collection was wasted effort. Setting a filter in the com.ibm.mqe.MQeTrace class allows user code to direct WebSphere MQ Everyplace product code not to collect the trace information in the first place, reducing wasted memory and CPU cycles spent.
- Separate tracing functionality out from the WebSphere MQ Everyplace base class, allowing tracing to occur without instantiating a WebSphere MQ Everyplace object.
- Remove the dependency of tracing code on a resource bundle in the examples.trace package
- Remove the ability for users to modify the meaning of WebSphere MQ Everyplace product trace points. Such user changes confused IBM service staff.
- Remove the ability for users to generate WebSphere MQ Everyplace product trace in non-English language. Support of the product might require English trace information to be collected and analysed.
- Provide a selection of optional, fully-supported trace information collectors
- Provide a trace collection mechanism which does not need to render binary trace information into string information before storing data to persistent storage. The necessity to retain many trace strings in the JVM creating the trace information reduces the footprint of the WebSphere MQ Everyplace solution, while retaining the ability to collect trace information in a more compact form on persistent media.
- Retain a pluggable trace interface for extension by user code

How to migrate from WebSphere MQ Everyplace version 1 to the WebSphere MQ Everyplace version 2.0 trace mechanism

To migrate from version 1 of the product to version 2.0 of the product:

1. Review the documentation presented in the Java programming reference material, particularly the following classes
   - `com.ibm.mqe.MQeTrace`
   - `com.ibm.mqe.MQeTraceHandler`
   - All classes in the `com.ibm.mqe.trace` package
2. Stop using the `com.ibm.mqe.MQeTraceInterface` class. This class is deprecated in version 2.0. Change your code to implement the `com.ibm.mqe.MQeTraceHandler` interface instead.
3. Stop using the `com.ibm.mqe.MQe.setTraceHandler()` method. Use the `com.ibm.mqe.MQeTrace.setHandler()` method instead.
4. Stop using the `com.ibm.mqe.MQe.getTraceHandler()` method. Use the `com.ibm.mqe.MQeTrace.getHandler()` method instead.

5. Stop using the `com.ibm.mqe.MQe.trace(...)` methods. Use the `com.ibm.mqe.MQeTrace.trace(...)` methods instead.
6. Remove any dependencies your code has on `examples.trace.MQeTraceResource` string resource bundle classes. This class has been removed from version 2.0 of WebSphere MQ Everyplace. The version 2.0 trace mechanism does not provide a simple resource bundle in the examples.trace package to use when decoding trace information. Access to WebSphere MQ Everyplace product trace data is provided through the `com.ibm.mqe.trace.MQeTraceRenderer`, `com.ibm.mqe.trace.MQeTracePoint` and `com.ibm.mqe.trace.MQeTracePointGroup` classes.
7. Consider whether trace information can be left in binary format, using the `com.ibm.mqe.trace.TraceToBinaryFile` or similar classes provided by the WebSphere MQ Everyplace product.
8. Consider instantiating a trace collection handler, and setting it into the MQeTrace class, but setting the filter in the `MQeTrace` class to discard or collect information as desired.
9. Consider that IBM service staff may ask for capture of trace information to diagnose the cause of problems reported. Allowing the setting of the MQeTrace filter may be easier than allowing the configuration of a collection trace handler.

Appendix D. Notices

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

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

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

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

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

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

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

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

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,

Winchester,
Hampshire
England
SO21 2JN

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

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

COPYRIGHT LICENSE:

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

Trademarks

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

AIX DB2
Everyplace IBM iSeries MQSeries
SupportPac UNIX
WebSphere z/OS zSeries

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

UNIX is a registered trademark of X/Open in the United States and other countries.

Windows and Windows NT are registered trademark of Microsoft Corporation in the United States and other countries.

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

Glossary

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York:. McGraw-Hill, 1994.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

asynchronous messaging. A method of communicating between programs in which the programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

authenticator. A program that checks that verifies the senders and receivers of messages.

bridge. An WebSphere MQ Everyplace object that allows messages to flow between WebSphere MQ Everyplace and other messaging systems, including WebSphere MQ.

channel. See *dynamic channel*, client/server channel, peer channel, and *MQI channel*.

channel manager. An WebSphere MQ Everyplace object that supports logical multiple concurrent communication pipes between end points.

class. A class is an encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client. (1)In WebSphere MQ Everyplace, a client is WebSphere MQ Everyplace code running without a channel manager or channel listener. Contrast with *server (1)*. (2)In WebSphere MQ, a client is a run-time component that provides access to queuing services on a server for local user applications.

client/server channel. An WebSphere MQ Everyplace a unidirectional channel between a client and a server that can only be established from the client side. Contrast with *peer channel*.

compressor. A program that compacts a message to reduce the volume of data to be transmitted.

cryptor. A program that encrypts a message to provide security during transmission.

device. A small portable machine running WebSphere MQ Everyplace as a client. Contrast with *server(1)*.

dynamic channel. This is a name given to WebSphere MQ Everyplace channels that connect clients and servers to enable the transfer of messages. They are called *dynamic* because they are created on demand. See *client/server* and *peer* channels. Contrast with*MQI channel*.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

gateway. An WebSphere MQ Everyplace gateway is a computer running the WebSphere MQ Everyplace WebSphere MQ bridge code.

Hypertext Markup Language (HTML). A language used to define information that is to be displayed on the World Wide Web.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is a class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

Internet. The Internet is a cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Naming and Directory Service (JNDI). An API specified in the Java programming language. It provides naming and directory functions to applications written in the Java programming language.

Lightweight Directory Access Protocol (LDAP). LDAP is a client-server protocol for accessing a directory service.

Local area network (LAN). A computer network located on a user's premises within a limited geographical area.

message. In message queuing applications, a message is a communication sent between programs.

message queue. See queue

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method. Method is the object-oriented programming term for a function or procedure.

MQI channel. An MQI channel connects a WebSphere MQ client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner. MQI channels must be explicitly created. Contrast with *dynamic channels*.

WebSphere MQ. WebSphere MQ is a family of IBM licensed programs that provide message queuing services.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In WebSphere MQ, an object is a queue manager, a queue, or a channel.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private methods and fields in the classes.

peer channel. A bidirectional WebSphere MQ Everyplace channel, normally used between clients. The connection can be established from either end.

personal digital addistant (PDA). A pocket sized personal computer.

private. A private field is not visible outside its own class.

protected. A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible

queue. A queue is a WebSphere MQ object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. A queue manager is a system program the provides message queuing services to applications.

server. (1) An WebSphere MQ Everyplace server is WebSphere MQ Everyplace code with an WebSphere MQ Everyplace channel manager, and WebSphere MQ Everyplace channel listener, configured. This provides the ability to receive from multiple devices and servers concurrently. Contrast with *client (1)*. (2)A computer running WebSphere MQ Everyplace server code. Contrast with *device*. (3) A WebSphere MQ server is a queue manager that provides message queuing services to client applications running on a remote workstation. (4) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server, or the computer on which a server program runs.

servlet. A Java program which is designed to run only on a web server.

subclass. A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

synchronous messaging. A method of communicating between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing . Contrast with *asynchronous messaging*.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Web. See World Wide Web.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Bibliography

Related publications:

- *WebSphere MQ Everyplace Read Me First*, SC34-6276-01
- *WebSphere MQ Everyplace Introduction*, SC34-6277-01
- *WebSphere MQ Everyplace Java Programming Reference*, SC34-6279-01
- *WebSphere MQ Everyplace System Programming Guide*, SC34-6274-01
- *WebSphere MQ Everyplace C Bindings Programming Guide*, SC34-6280-01
- *WebSphere MQ Everyplace C Programming Reference*
- *WebSphere MQ Everyplace C Programming Guide for Palm OS*, SC34-6281-01
- *WebSphere MQ Everyplace Configuration Guide*, SC34-6283-01
- *WebSphere MQ An Introduction to Messaging and Queuing*, GC33-0805-01

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–842327
  - From within the U.K., use 01962–842327
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

SC34-6278-01