WebSphere MQ Everyplace

Systems Programming Guide

Version 2.0

SC34-6274-00

Take Note!

Before using this information and the product it supports, read the general information under “Notices” on page 75

First edition (November 2002)

This edition applies to WebSphere® MQ Everyplace™ Version 2.0 (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/mqseries/library/*.

Contents

About this book

This book is a programming guide for the WebSphere MQ Everyplace product . This book is intended to be used in conjunction with the WebSphere MQ Everyplace Application Programming Guide and existing books or manuals on the programming languages that are used to write WebSphere MQ Everyplace programs, that is the WebSphere MQ Everyplace C Programming Reference, the WebSphere MQ Everyplace Java Programming Reference, and the WebSphere MQ Everyplace C Bindings Programming Guide.

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 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, please make sure that you have the necessary licenses.

1. The pricing of licenses for use of the Program on servers is based on 'Processor License Units'. Use of each copy of the Program on a server requires one Processor License Unit to be acquired for each processor or symmetric multiprocessor contained in the server on which the copy of the Program is to run. Different types of Processor License Units and 'Device Use Authorisations' are required, depending on whether the Program is running on point-of-sale, that is retail, equipment or on another type of computer. Use of the Program on retail equipment requires a 'Retail' server license, whereas use on other (non-retail) equipment requires a 'Network' server license.
2. Additional 'Device Use Authorisation' is required for any use of the Program on a separate client device, except those included in the Network Server license, as described at 3) below.
3. Each 'Network' server license includes authorisation for the restricted use of the Program with no more than one hundred (100) client devices, on condition that all such copies are used in the same economic enterprise or organisation as the server copy.

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

Who should read this book

This book is intended for developers who want to write WebSphere MQ Everyplace programs for use in a pervasive computing environment. It contains information and examples for Java™ and C codebases.

Prerequisite knowledge

It is assumed that the reader has a working knowledge of the basic programming techniques for the language in which the WebSphere MQ Everyplace programs are to be written.

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 Everyplace books:

- WebSphere MQ Everyplace Read Me First, SC34-6277-00
- WebSphere MQ Everyplace Introduction
- WebSphere MQ Everyplace Application Programming Guide, SC34-6278-00

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

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.

Migrating from version 1.2.6 or lower to version 1.2.7 or higher

This section provides important information if you are migrating from version 1.2.6 or lower to a higher version or release of WebSphere MQ Everyplace

The `MQeMQMsgObject` class has been updated to put Ascii values to the `Msg_ReplyToQ` and `Msg_ReplyToQMgr` fields. This class still understands Unicode values already set to messages in Version 1.2.6 or lower, but the data is treated as being Ascii data. No difference occurs when sending an `MQeMQMsgObject` from a Version 1.2.6 or lower queue manager to a version 1.2.7 queue manager.However, an `MQeMQMsgObject` fails to send from a version 1.2.7 queue manager to a Version 1.2.6 or lower queue manager. To avoid this problem, do **one** of the following:

- Upgrade both sending and receiving installations to WebSphere MQ Everyplace version 1.2.7
- Place the WebSphere MQ Everyplace version 1.2.7 `MQeMQMsgObject` class on the classpath in front of the version 1.2.6 installation and upgrade the version 1.2.6 WebSphere MQ Everyplace classes at a later stage.
- Place the version 1.2.6 `MQeMQMsgObject` class on the classpath in front of the WebSphere MQ Everyplace version 1.2.7 installation and upgrade both installations at a later stage.

The following methods in the `com.ibm.mqe.MQe` class were deprecated in version 1.2.7:

- `MQe.utfToUnicode`
- `MQe.UnicodeToutf`

The following publically accessible variable in the com.ibm.mqe.MQe class has been deprecated in version 1.2.7:

- `MQe.Loader`

Migrating to version 2.0

This section contains information on updating your application to version 2.0.

Trace

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.

Chapter 1. Security

This chapter discusses the following aspects of security in WebSphere MQ Everyplace:

- Authenticators
- Certificate management

Authenticators

Authenticators are invoked by security attributes. Therefore, how and when they are used is determined by the specific implementation of an attribute. One main usage of authenticators is for controlling access to queues in queue-based security. Authenticators can be used in queue-based security to control access to queues. WebSphere MQ Everyplace provides a certificate authenticator as part of its base code, `com.ibm.mqe.attributes.MQeWTLSCertAuthenticator`. There are some Java example authenticators, in the examples.attributes directory, which are based on user names and passwords. There is also a C example, WinCEAuthenticator, in the `examples\src\WinCEAuthenticator` directory. In addition to these, WebSphere MQ Everyplace allows you to write your own authenticator.

How to write an authenticator

In queue-based security, authenticators are activated when a queue is first accessed and they can grant or deny access to the queue. When a queue is accessed from its local queue manager, the authenticator is activated when the first operation, for example put, get , or browse is performed on the queue. When a queue is accessed from a remote queue manager, WebSphere MQ Everyplace establishes a channel between the two queue managers and the authenticator is activated as part of establishing the channel.

Java codebase

All authenticators must extend the base authenticator class:

```
class MyAuthenticator extends com.ibm.mqe.MQeAuthenticator
```

The following methods in the base class can be overridden:

`activateMaster()`

The signature for this method is:

```
public byte[] activateMaster( boolean local ) throws Exception
```

It is invoked on the queue manager that initiates access to a queue. The parameter local indicates whether this is a local access, that is the queue is on the same queue manager, local == true, or a remote access, local == false. The method should collect data to authenticate the queue manager or user and return the data in a byte array. The data is passed to the `activateSlave()` method. The `activateMaster()` method in the base class, MQeAuthenticator, simply returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by WebSphere MQ Everyplace itself, but are passed back to the user's code and terminate the attempt to access the queue.

`activateSlave()`

The signature for this method is:

```
public byte[] activateSlave( boolean local,  byte data[] ) throws Exception
```

This is invoked on the queue manager that owns the queue. The parameter local indicates whether this is a local access, i.e. initiated on the same queue manager, local == true, or a remote access, local == false. The parameter datacontains the data returned by the `activateMaster()` method. The `activateSlave()` method should validate this data. If it is satisfied with the data it should call the `setAuthenticatedID()` method to set the name of the authenticated entity, this indicates that the first stage of the authentication was successful. It can then collect data to authenticate the local queue manager and return it in a byte array. The data is passed to the `slaveResponse()` method. If it is not satisfied with the data, it throws an exception indicating the reason. The `activateSlave()` method in the base class, MQeAuthenticator, checks whether the name of the authenticated entity has been set and if it has, it logs the name; it then returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by WebSphere MQ Everyplace itself, but are passed back to the initiating queue manager where they are re-thrown. WebSphere MQ Everyplace does not catch these exceptions on the initiating queue manager and they are passed back to the user's code and will terminate the attempt to access the queue.

`slaveResponse()`

The signature for this method is:

```
public void slaveResponse( boolean local, byte data[] ) throws Exception
```

It is invoked on the queue manager that initiates access to a queue. The local parameter indicates whether this is a local access, local == true, or a remote access, local == false. The parameter data contains the data returned by the activateSlave() method. If it is satisfied with the data it should call the `setAuthenticatedID()` method to set the name of the authenticated entity, this indicates that the second stage of the authentication was successful. If the `activateSlave()` method did not return any data, and the slaveResponse() method is satisfied with this, it still calls `setAuthenticatedID()` to indicate success. If it is not satisfied with the data, it throws an exception indicating the reason. The `slaveResponse()` method in the base class, MQeAuthenticator, simply returns null. It does not throw any exceptions. Any exceptions thrown by this method, in a subclass, are not caught by WebSphere MQ Everyplace itself, but are passed back to the user's code and terminate the attempt to access the queue.

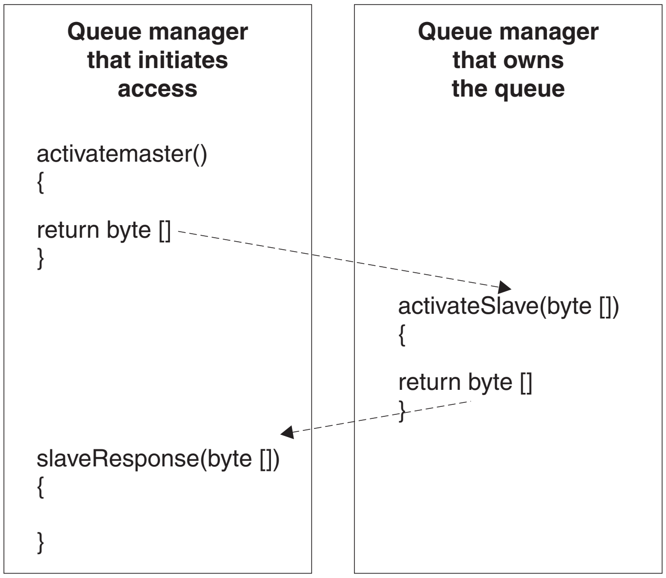


Figure 1. The slaveResponse() method in MQeAuthenticator

When a queue is accessed locally, the three methods are invoked in sequence on the local queue manager.

The example logon authenticator

The example logon authenticator shows how to implment these. It has a base class, `examples.attributes.LogonAuthenticator`, and three subclasses, one for the NTAuthenticator, one for the UnixAuthenticator, and one for the UseridAuthenticator. The base class provides common functionality and the subclasses provide functionality that is specific to the type of authenticator, that is NT, Unix, or Userid. The `activateMaster()` method in the LogonAuthenticator class creates an empty MQeFields object and passes it into a method called `prompt()`. This is overridden in each of the subclasses, and in each case it displays a Java dialog box, collects data from it, masks the data with a simple exclusive OR operation, and adds the data to the MQeFields object. The exclusive OR is used in the example authenticators but in practice it does not provide much protection. The MQeFields object is dumped to provide a byte array which is returned by `activateMaster()`. The `activateMaster()` method is invoked on the queue manager that initiates access to the queue, so the dialog box is displayed by this queue manager.

```
public byte[] activateMaster(boolean local) throws Exception {
  MQeFields fields = new MQeFields();
/* for request fields       */
  this.prompt(fields);
/* put up the dialog prompt  */
  return (fields.dump());
/* return ID                */
}
```

The `activateSlave()` method receives the data returned by `activateMaster()`, restores it into a MQeFields object and passes the object into the `validate()` method. This is overridden in each of the subclasses, and in each case it validates the data in a way appropriate to the authenticator. For example, in the NTAuthenticator subclass, the `validate()` method unmasks the data and passes it to the `logonUser()` method. This method uses Java Native Interface (JNI) to access the Windows security mechanism and check whether the user name and password are valid. If they are valid, the `validate()` method returns the user name, otherwise it throws an exception.

```
public byte[] activateSlave(boolean local,
                            byte    data[]) throws Exception {
  MQeFields fields = new MQeFields(data); /* work object              */
  try {
    authID = this.validate(fields);
 /* get the auth ID value    */
    setAuthenticatedID(authID);
 /* is it allowed ?          */
    super.activateSlave(local, data);
 /* call ancestor            */
    trace("_:Logon " + authID);
 /* trace                    */
    MQeFields result = new MQeFields();
 /* reply object             */
    result.putAscii(Authentic_ID, authID);/* send id                  */
    return (result.dump());
 /* send back as response    */
  }
  catch (Exception e) {
 /* error occured            */
    authID = null;
 /* make sure authID is null */
    setAuthenticatedID(null);
 /* invalidate               */
    throw e;
 /* re-throw the exception   */
  }
}
```

If the user name is valid, the `activateSlave()` method calls `setAuthenticatedID()` to register the user name and the calls `super.activateSlave()` which puts out a log message. It issues a trace message, adds the user name to a MQeFields object, dumps this to a byte array and returns it. If the user name is not valid, `validate()` throws an exception. The `activateSlave()` method catches the exception, ensures the authenticated id is null and re-throws the exception. The `slaveResponse method()` receives the byte array returned by `activateSlave()` and restores it into a MQeFields object. The user name that was validated by `activateSlave()` is extracted from this and passed to `setAuthenticatedID()`.

```
public void slaveResponse(boolean local, byte data[])

throws Exception {   super.slaveResponse(local, data);       /* call ancestor*/
MQeFields fields = new MQeFields(data);                      /* work object*/
setAuthenticatedID(fields.getAscii(Authentic_ID));           /* id to check   */
}
```

These authenticators behave the same for both local and remote accesses, so they ignore the local parameter to these methods.

C codebase

In the C codebase, you need to provide at least four functions to implement an authenticator needs. These functions are:

1. `new()`
2. `activateMaster()`
3. `ctivateSlave()`
4. `slaveResponse()`

In terms of functionality, functions 2 to 4 behave exactly the same as their Java counterpart implementation. If your `new()` function allocates any private memory, you then have to provide a `free()` function, which frees the private memory you have allocated.

new() The `new()` function is executed when the authenticator is loaded by WebSphere MQ Everyplace. It serves as an initialisation function for the authenticator. Its main functionality includes:

- Alocating private memory, if required
- Notifying theWebSphere MQ Everyplace system of the implementations for the `activateMaster()`, `activateSlave()`, `slaveResponse()`, and `free()` functions
- Providing initial values for private variables

To notify the WebSphere MQ Everyplace of the existence of your implementation, call the `mqeClassAlias_add()` function, which has the following signature:

```
MQERETURN mqeClassAlias_add(MQERETURN * pExceptBlock,
       MQeStringHndl hWinCEAuthName,
                          MQeStringHndl hModuleName,
                           MQeStringHndl hInitFuncName);
```

In the previous example, the `hWinCEAuthName` is a string name for the authenticator. The `hModuleName` is the dynamically loadable library file name in which your authenticator has been compiled into, and the `hInitFuncName` is the name of your new function, which can be an arbitrary name. The `new()`function has the following signature:

```
MQERETURN new(MQeAttrPlugin_SubclassInitInput * pInput,
              MQeAuthenticator_SubclassInitOutput * pOutput
           );
```

The `pOutput` points to an `MQeAuthenticator_SubclassInitOutput` structure, which needs to be filled in. The `MQeAuthenticator_SubclassInitOutput` contains the following fields:

MQEVERSION version;
Assign `MQE_CURRENT_VERSION` to this variable.

MQeStringHndl hClassName;
Assign the Java class name of the authenticator, MQeString, to this variable.

MQEBOOL regRequired;
Assign `MQE_FALSE` to this variable.

MQEKEYTYPE keyType;
Assign `MQE_KEY_NULL` to this variable.

MQeAuthenticator_FreeFunc fFree;
Assign the address of the `free()` function to this variable.

MQeAuthenticator_ActivateMasterPrepFunc fActivateMasterPrep;
Assign the address of the `activateMaster()` function to this variable.

MQeAuthenticator_ActivateSlavePrepFunc fActivateSlavePrep;
Assign the address of the `activateSlave()` function to this variable.

MQeAuthenticator_ProcessSlaveResponseFunc fProcessSlaveResponse;
Assign the address of the `activateSlave()` function to this variable.

MQeAuthenticator_CloseFunc fClose;
Assign `NULL` to this variable.

MQEVOID * pSubclassPrivateData;
Assign the address of authenticator's private data memory to this variable.

Any pointers or handles that are not used in the implementation must be initialised to `NULL`.

free() The signature of `free()` is:

```
MQERETURN free(MQeAuthenticatorHndl  hThis,
                 MQeAttrPlugin_FreeInput * pInput,
                 MQeAttrPlugin_FreeOutput * pOutput
                );
```

If the `new()` function allocates private memory, the pointer to the allocated memory can be retrieved into a pointer p using:

```
mqeAuthenticator_getPrivateData(hThis, pExceptBlock, (MQEVOID **) &p);
```

The pointer can then be used to free the memory. The MQeString assigned to the hClassName in the `new()` function, if any, are automatically freed by the system when `mqeAttrBase_free` is called.

activateMaster()
The signature of `activateMaster()` is:

```
MQERETURN activateMaster(MQeAuthenticatorHndl   hAuthenticator,
                           MQeAttrPlugin_ActivateMasterPrepInput *pInput,
                           MQeAttrPlugin_ActivateMasterPrepOutput * pOutput
                          );
```

Refer to description in the corresponding Java section for the required functionality for this function. The `pOutput` points to an `MQeAttrPlugin_ActivateMasterPrepOutput` structure which needs to be filled in. The `MQeAttrPlugin_ActivateMasterPrepOutput` contains the following fields:

MQEINT32 * pOutputDataLen;
Assign the length of the output data for `activateSlave()` to this variable.

MQEBYTE * pOutputData;
Assign the address of the output data buffer for `activateSlave()` to this variable.

activateSlave()
The signature of `activateSlave()` is:

```
  MQERETURN activateSlave(MQeAuthenticatorHndl hAuthenticator,
                           MQeAttrPlugin_ActivateSlavePrepInput *pInput,
                           MQeAttrPlugin_ActivateSlavePrepOutput *pOutput
                         );
```

Refer to description in the corresponding Java section for the required functionality for this function. The pInput points to an `MQeAttrPlugin_ActivateSlavePrepInput` structure which contains the input from the `activateMaster()` and the `pOutput` points to an

MQeAttrPlugin_ActivateSlavePrepOutput structure which needs to be filled in. The MQeAttrPlugin_ActivateSlavePrepInput contains the following fields:

MQEINT32 * pInputDataLen;
Get the length of the input data from activateMaster() from this variable.

MQEBYTE * pInputData;
Get the address of the input data buffer from activateMaster() from this variable.

The MQeAttrPlugin_ActivateSlavePrepOutput contains the following fields:

MQEINT32 * pOutputDataLen;
Assign the length of the output data for slaveResponse() to this variable.

MQEBYTE * pOutputData;
Assign the address of the output data buffer for slaveResponse() to this variable.

slaveResponse()
The signature of slaveResponse() is:

```
MQERETURN slaveResponse(MQeAuthenticatorHndl hAuthenticator,
                        MQeAttrPlugin_ProcessSlaveResponseInput *pInput,
                        MQeAttrPlugin_ProcessSlaveResponseOutput *pOutput
                         );
```

Refer to description in the corresponding Java section for the required functionality for this function. The pInput points to an MQeAttrPlugin_ProcessSlaveResponseInput structure which contains the input from the activateSlave(). The MQeAttrPlugin_ProcessSlaveResponseInput contains the following fields:

MQEINT32 * pInputDataLen;
Get the length of the input data from activateSlave() from this variable.

MQEBYTE * pInputData;
Get the address of the input data buffer from activateSlave() from this variable.

The example WinCEAuthenticator

The example WinCEAuthenticator shows how the methods listed in the previous section can be implemented. It is functionally very similar to the example NTAuthenticator in the Java code base.

Calling winCEAuthenticator_new() function implements the new() function. This allocates a private memory block to register the type of the authenticator, private to this implementation, filling-in private variables and the function pointers mentioned above so they point to the right function implementations, and set pOutput->hClassName to ″WinCEAuthenticator″. Notice that no ″WinCEAuthenticator″ is provided in the Java package. This is because the WinCEAuthenticator is designed to be executed only on a C client. The ″WinCEAuthenticator″ string is created for demonstration purposes only. The pOutput->hClassName must point to an existing Java class if the authenticator is to be used in a dialogue between a C client and a Java server.

```
 MQERETURN winCEAuthenticator_new(
        MQeAttrPlugin_SubclassInitInput * pInput,
        MQeAuthenticator_SubclassInitOutput * pOutput
       ) {

    MQeStringHndl hClassName;
    MQeExceptBlock * pExceptBlock =
        (MQeExceptBlock*) pOutput->pExceptBlock;

    (void)mqeString_newChar8(pExceptBlock,
                             &hClassName,
                             "WinCEAuthenticator");
    if (MQERETURN_OK == pExceptBlock->ec) {
      pOutput->pSubclassPrivateData = malloc(sizeof(MQEINT32));
      if (NULL != pOutput->pSubclassPrivateData) {
         *((MQEINT32 *)pOutput->pSubclassPrivateData) = AUTHENTICATOR;
         pOutput->hClassName = hClassName;
         pOutput->regRequired = MQE_FALSE;
         /* key type unknown */
         pOutput->keyType = MQE_KEY_NULL;

         /* pointers to subclass implementations of support methods */
         pOutput->fFree = winCEAuthenticator_free;
         pOutput->fActivateMasterPrep = winCEAuthenticator_activateMasterPrep;
         pOutput->fActivateSlavePrep = winCEAuthenticator_activateSlavePrep;
         pOutput->fProcessSlaveResponse = winCEAuthenticator_processSlaveResponse;
         pOutput->fClose = NULL;
      } else {
         pExceptBlock->ec  = MQERETURN_ALLOCATION_FAIL;
         pExceptBlock->erc = MQEREASON_NA;
      }
   }

   return pExceptBlock->ec;
}
```

Calling `winCEAuthenticator_free()` implements the `free()` function. It retrieves the private memory block allocated by `winCEAuthenticator_new()`, making sure the authenticator has got the right private signature, and then frees the memory block.

```
 MQERETURN winCEAuthenticator_free(MQeAuthenticatorHndl  hThis,
                                  MQeAttrPlugin_FreeInput * pInput,
                                  MQeAttrPlugin_FreeOutput * pOutput
                                    ) {

    MQeExceptBlock * pExceptBlock = (MQeExceptBlock*)
            pOutput->pExceptBlock;
    MQEINT32 * pType;

    pExceptBlock->ec  = MQERETURN_INVALID_ARGUMENT;
    pExceptBlock->erc = MQEREASON_INVALID_SIGNATURE;

    if ((NULL != hThis) &&
        (MQERETURN_OK == mqeAuthenticator_getPrivateData(hThis,
                                                 pExceptBlock,
                                                 (MQEVOID**) &pType))
                                                    ) {
       /* make sure it is an authenticator created here */
       if (AUTHENTICATOR == *pType) {
          pExceptBlock->ec  = MQERETURN_OK;
          pExceptBlock->erc = MQEREASON_NA;
          free(pType);
       }
    }

    return pExceptBlock->ec;
 }
```

Calling `winCEAuthenticator_activateMasterPrep()` implements the `activateMaster()` function. It creates an empty MQeFields structure and passes it into a function called `prompt()`. The `prompt()` function:

- Displays a dialogue box
- Collects data from the dialogue box
- Masks the data with a simple exclusive OR operation
- Adds the data to the MQeFields object

The exclusive OR is used in the example authenticators, but in practice it does not provide much protection. The MQeFields structure is then dumped to provide a byte array, which is returned by `winCEAuthenticator_activateMasterPrep()`.

```
MQERETURN winCEAuthenticator_activateMasterPrep(
     MQeAuthenticatorHndl  hAuthenticator,
     MQeAttrPlugin_ActivateMasterPrepInput *  pInput,
     MQeAttrPlugin_ActivateMasterPrepOutput * pOutput) {

   static MQeFieldsHndl hActivateMasterFields = NULL;
   MQEINT32 * pOutputDataLen = pOutput->pOutputDataLen;
   MQEBYTE * pOutputData = pOutput->pOutputData;
   MQeExceptBlock * pExceptBlock =
      (MQeExceptBlock*) pOutput->pExceptBlock;


   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   if (NULL == hActivateMasterFields) {
      /* get data for authentication */
      (void)mqeFields_new(pExceptBlock,
        &hActivateMasterFields);
      if (MQERETURN_OK == pExceptBlock->ec) {
         /**
          * Write your code here which puts the input data,
          * for example., userid, password into hActivateMasterFields.
          * The format is not important as long as it can be
          * understood by your corresponding code in
          * winCEAuthenticator_activateSlavePrep, which digests
          * these data.
          */
    prompt(hActivateMasterFields, pExceptBlock);}
   }

   if (MQERETURN_OK == pExceptBlock->ec) {
      /* dump the fields */
      (void)mqeFields_dump(hActivateMasterFields,
                           pExceptBlock,
                           pOutputData,
                           pOutputDataLen);
   }

   if ((NULL != hActivateMasterFields) &&
       ((NULL != pOutputData) ||
      (MQERETURN_OK != pExceptBlock->ec))) {
      /**
       * Caller has supplied a buffer or operation failed.
       * No need to keep the Fields any more.
       */
      (void)mqeFields_free(hActivateMasterFields, NULL);
      hActivateMasterFields = NULL;
   }

   return pExceptBlock->ec;
}
```

security

The `winCEAuthenticator_activateSlavePrep()` implements the `activateSlave()` function. The `winCEAuthenticator_activateSlavePrep()` method receives the data returned by `winCEAuthenticator_activateMasterPrep()`, restores it into an MQeFields structure and passes it into a `validate()` function. The `validate()` function unmasks the data and passes it to the system `LogonUser()` function. This function checks if the user name and password are valid.

On a WinCE system, the `LogonUser()` never returns if the user name and password are not valid. The following `winCEAuthenticator_activateSlavePrep()` and `winCEAuthenticator_processSlaveResponse()` implementations, however, assume that LogonUser() will always return with a value indicating whether or not the input is valid, in order to demonstrate what you need to do. If the user name and password are valid, the `winCEAuthenticator_activateSlavePrep()` function calls `mqeAuthenticator_setAuthenticatedID()` to register the user name as if the code is running on a server. It may be that this code is running on a client just as the `winCEAuthenticator_activateMasterPrep`. It then adds the user name to an MQeFields, dumps this to a byte array, and returns it. If the user name is not valid, the `winCEAuthenticator_activateSlavePrep()` function returns an error.

```
MQERETURN winCEAuthenticator_activateSlavePrep(
               MQeAuthenticatorHndl  hAuthenticator,
               MQeAttrPlugin_ActivateSlavePrepInput *  pInput,
               MQeAttrPlugin_ActivateSlavePrepOutput * pOutput) {

   static MQeFieldsHndl hActivateSlaveFields = NULL;
   MQeFieldsHndl hTempFields = NULL;
   MQEINT32 inputDataLen = pInput->inputDataLen;
   MQEBYTE * pInputData = pInput->pInputData;
   MQEINT32 * pOutputDataLen = pOutput->pOutputDataLen;
   MQEBYTE * pOutputData = pOutput->pOutputData;
   MQeExceptBlock * pExceptBlock =
       (MQeExceptBlock*) pOutput->pExceptBlock;

   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   if (NULL == hActivateSlaveFields) {
/* restore input */
      (void)mqeFields_new(pExceptBlock, &hTempFields);
      if (MQERETURN_OK == pExceptBlock->ec) {
          /* restore it into a MQeFields */
          (void)mqeFields_restore(hTempFields,
                                  pExceptBlock,
                                  pInputData,
                                  inputDataLen);
         if (MQERETURN_OK == pExceptBlock->ec) {
            MQeStringHndl hAuthenticID = NULL;

            /**
             * put your code, which digests(authenticates)
             * the input data your gathered in the
             * winCEAuthenticator_activateMasterPrep().
             * If successful, create an AuthenicateID string
             * in hAuthenticID.
             */
            (void)validate(hTempFields, pExceptBlock,
          &hAuthenticID);

             if (MQERETURN_OK == pExceptBlock->ec) {
                /**
                * If successfully authenticated,
                * set local id variable (recored a success)
                */
               (void)mqeAuthenticator_setAuthenticatedID(hAuthenticator,
                                                    pExceptBlock,
                                                          hAuthenticID);
               /* preparation for sending the id to the master */
               if (MQERETURN_OK == pExceptBlock->ec) {
                  /**
                   * Send the hAuthenticID to the Master,
                   * indicating a a success.
                   */
                  (void)mqeFields_new(pExceptBlock,
             &hActivateSlaceFields);
                 if (MQERETURN_OK == pExceptBlock->ec) {
                    MQeStringHndl hAuthenticIDField;
      (void)mqeString_newChar8(pExceptBlock,
                                             &hAuthenticIDField,
                                             AUTHENTIC_ID);
                     if (MQERETURN_OK == pExceptBlock->ec) {
                        (void)mqeFields_putAscii(hActivateSlaveFields,
                                                 pExceptBlock,
                                                 hAuthenticIDField,
                                                 hAuthenticID);
                        (void)mqeString_free(hAuthenticIDField, NULL);
                     }
                  }
               }
            }
         }
         (void)mqeFields_free(hTempFields, NULL);
      }
   }

   if (MQERETURN_OK == pExceptBlock->ec) {
      /* dump the fields */
      (void)mqeFields_dump(hActivateSlaveFields,
                            pExceptBlock,
                            pOutputData,
                         pOutputDataLen);
   }

   if ((NULL != hActivateSlaveFields) &&
        ((NULL != pOutputData) || (MQERETURN_OK != pExceptBlock->ec))) {
       /**
        * Caller has supplied a buffer or operation failed.
        * No need to keep the Fields any more.
        */
       (void)mqeFields_free(hActivateSlaveFields, NULL);
       hActivateSlaveFields = NULL;
   }

   return pExceptBlock->ec;
}
```

Calling `winCEAuthenticator_processSlaveResponse()` implements the `slaveResponse()` function. The `winCEAuthenticator_processSlaveRespons()` function receives the byte array returned by `winCEAuthenticator_activateSlavePrep()` and restores it into an MQeFields structure. The user name, validated by `activateSlave()`, is extracted from this and passed to `mqeAuthenticator_setAuthenticatedID()`.

```
MQERETURN winCEAuthenticator_processSlaveResponse(
         MQeAuthenticatorHndl     hAuthenticator,
         MQeAttrPlugin_ProcessSlaveResponseInput *   pInput,
         MQeAttrPlugin_ProcessSlaveResponseOutput *   pOutput
        ) {

   MQEINT32 inputDataLen = pInput->inputDataLen;
   MQEBYTE * pInputData = pInput->pInputData;
   MQeFieldsHndl hFields;
   MQeExceptBlock * pExceptBlock =
        (MQeExceptBlock *)pOutput->pExceptBlock;

   /* initialize exception block */
   pExceptBlock->ec  = MQERETURN_OK;
   pExceptBlock->erc = MQEREASON_NA;

   /* restore input */
   (void)mqeFields_new(pExceptBlock, &hFields);
   if (MQERETURN_OK == pExceptBlock->ec) {
      (void)mqeFields_restore(hFields,
                              pExceptBlock,
                              pInputData,
                              inputDataLen);
      /* get ID */
      if (MQERETURN_OK == pExceptBlock->ec) {
         MQeStringHndl hAuthenticIDField;

         (void)mqeString_newChar8(pExceptBlock,
                                  &hAuthenticIDField,
                                  AUTHENTIC_ID);
         if (MQERETURN_OK == pExceptBlock->ec) {
            MQeStringHndl hAuthenticID;
   (void)mqeFields_getAscii(hFields,
                                   pExceptBlock,
                                   &hAuthenticID,
                                   hAuthenticIDField);
            /** If the above call failed,
             * then the authentication by the slave was not successful.
             */
            if (MQERETURN_OK == pExceptBlock->ec) {
               /* set local ID */
               (void)mqeAuthenticator_setAuthenticatedID(hAuthenticator,
                                                         pExceptBlock,
                                                         hAuthenticID);
            }
            (void)mqeString_free(hAuthenticIDField, NULL);
         }
      }
      (void)mqeFields_free(hFields, NULL);
   }
   return pExceptBlock->ec;
}
```

Certificate management

WebSphere MQ Everyplace can use private or public key encryption for message level security using the MQeMTrustAttribute, and for queue based security using the MQeWTLSCertAuthenticator. Any entity, for example queue manager, queue, application, person, which needs private and public keys must have a private registry. When the registry is initialised it generates and store the keys, if the associated information is supplied. The private key is encrypted and stored directly in the registry. The public key is sent to the certificate server, this returns a public certificate containing the public key and the registry stores the certificate. For message level security, the certificates must also be copied to public registries so that they are available to other entities that need them. This is not required for queue based security.

The certificate server normally issues certificates, which are valid for 12 months. The certificates cannot be used once they have expired, so it is important to keep track of the expiry dates and to renew the certificates before they expire.

Examining certificates

Certificates can be examined using the `com.ibm.mqe.attributes.MQeListCertificates` class. This class opens a registry and allows you to list all the certificates in it, or to examine specific certificates by name. To use the class, you must supply the name of the registry and a MQeFields object that contains the information required to open it:

`MQeRegistry.LocalRegType (ascii)`
For a public registry, set this parameter to `com.ibm.mqe.registry.MQeFileSession`. For a private registry, set it to `com.ibm.mqe.registry.MQePrivateSession`.

`MQeRegistry.DirName (ascii)`
The name of the directory holding the registry files.

`MQeRegistry.PIN(ascii)`
The PIN protecting the registry. This is only required for private registries.

No other parameters are required to open the registry for this class. If the registry is a public registry with the name ″MQeNode_PublicRegistry″and the class is initialised in the directory that contains the registry, the MQeFields object can be null. If the registry belongs to the mini-certificate server, its name is ″MiniCertificateServer″. If the registry belongs to a queue, its name is

″MiniCertificateServer″. If the registry belongs to a queue, its name is

```
    MQeListCertificates list;
    String fileRegistry = "com.ibm.mqe.registry.MQeFileSession";
    String privateRegistry = "com.ibm.mqe.registry.MQePrivateSession";

    void open(String regName, String regDirectory,
     String regPIN) throws Exception
    {
        MQeFields regParams = new MQeFields();
        // if regPIN == null, assume file registry
        String regType = (regPIN == null) ?
         fileRegistry : privateRegistry;
        regParams.putAscii(MQeRegistry.RegType, regType);
        regParams.putAscii(MQeRegistry.DirName, regDirectory);
        if (regPIN != null)
            regParams.putAscii(MQeRegistry.PIN, regPIN);

        list = new MQeListCertificates(regName, regParams);
}
```

This constructor opens the registry. Once this has been done, the registry entries for the certificates can be retrieved. They can either be retrieved individually by name:

```
MQeFields entry = list.readEntry(certificateName);
```

or all the certificate entries in the registry can be retrieved together:

```
MQeFields entries = list.readAllEntries();
```

The value returned from `readAllEntries()` is a MQeFields object that contains a field for each certificate in the registry, the name of the field is the name of the certificate and the contents of the field is a MQeFields object containing the registry entry. You can process each registry entry using an enumeration:

```
     Enumeration enum = entries.fields();

            if (!enum.hasMoreElements())
                System.out.println("no certificates found");
```

```
else
{
    while (enum.hasMoreElements())
    {
        // get the name of the certificate
        String entity = (String) enum.nextElement();
        // get the certificate's registry entry
        MQeFields entry = entries.getFields(entity);

        // do something with it
        ...
    }
}
```

The certificate can be obtained from the registry entry using the `getWTLSCertificate()` method:

```
Object certificate = list.getWTLSCertificate(entry);
```

Information can now be obtained from the certificate:

```
  String subject   = list.getSubject(certificate);
String issuer    = list.getIssuer(certificate);
long   notBefore = list.getNotBefore(certificate);
long   notAfter  = list.getNotAfter(certificate);
```

The `notBefore` and `notAfter` times are the number of seconds since the midnight starting 1st January 1970, that is the standard UNIX format for dates and times.

Finally, the list object should be closed:

```
list.close();
```

The MQeListCertificates class is used in the example program, `examples.certificates.ListWTLSCertificates`, which is a command-line program that lists certificates.

The program has one compulsory and three optional parameters:

ListWTLSCertificates <regName>[<ini file>][<level>][<cert names>]

where:

regName
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
This 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 The level of detail for the listing. This can be:

- ″-b″ or ″-brief″, which prints the names of the certificate, one name per line.
- ″-f″ or ″-full″, which prints the names of the certificates and some of the contents.

This parameter is optional and if omitted the ″brief″ level of detail is used.

cert names
This 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.

The MQe_Explorer configuration tool can also be used to examine certificates which belong to queue managers or queues.

Renewing certificates

To ensure continuity of service, we recommend that you renew certificates before they expire. Certificates are renewed using the same mini-certificate issuance service that originally issued them. 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 a certificate is renewed, the new certificate contains the same public key as the old certificate. For additional security, you may wish to change credentials regularly. This involves generating a new private and public key, storing the new private key in the registry, and requesting a new certificate for the public key. If you use message level security with the MTrustAttribute, and change credentials, you will not be able to use the new credentials to read messages sent with the old credentials. The old credentials are not deleted, but are renamed within the registry so that they are still available.

The class `com.ibm.mqe.registry.MQePrivateRegistryConfigure` can be used both to renew certificates and to generate new credentials. To use the class, you must supply the name of the registry, an MQeFields object that contains the information required to open it, and optionally the registry’s PIN.

Chapter 2. Adapters

This chapter describes how to implement adapters in a WebSphere MQ Everyplace application. You can use WebSphere MQ Everyplace adapters to map WebSphere MQ Everyplace to storage or communications device interfaces. You can also write your own adapters.

This chapter contains the following sections:
- Storage adapters
- Communications adapters
- How to write adapters

Storage adapters

WebSphere MQ Everyplace provides the following storage adapters. You cannot alter the behavior of these adapters. For more information on the specific behaviour of each storage adapter, refer to the WebSphere MQ Everyplace Java Programming Reference and the WebSphere MQ Everyplace C Programming Reference.

Storage adapters	
MQeCaseInsensitiveDiskAdapter	Provides support for case insensitive matching when locating a specific file in permanent storage.
MQeDiskFieldsAdapter	Provides support for reading and writing to persistent storage.
MQeMappingAdapter	Provides support for mapping long file names to short file names.
MQeMemoryFieldsAdapter	Provides support for reading and writing to non-persistent storage.
MQeMidpFieldsAdapter	Provides support for reading and writing to permanent storage within a MIDP environment.
MQeReducedDiskFieldsAdapter	Provides support for high speed writing to permanent storage.

Communications adapters

WebSphere MQ Everyplace provides the following communications adapters. You can modify the behavior of these adapters using Java properties. For more information on how to use these properties and their effect on each communications adapter, refer to the WebSphere MQ Everyplace Java Programming Reference.

Communications adapters	
MQeMidpHttpAdapter	Provides support for reading and writing to the network using the HTTP 1.0 protocol in a MIDP environment.

MQeTcpipHistoryAdapter
Provides support for reading and writing to the network using the TCP protocol. This adapter provides the best TCP performance by chaching recently used data. Therefore, we recommend that you use this adapter.

MQeTcpipLengthAdapter
Provides support for reading and writing to the network using the TCP protocol.

MQeTcpipHttpAdapter
Provides support for reading and writing to the network using the HTTP 1.0 protocol. Also provides support for passing HTTP requests through proxy servers.

Note: If using the Microsoft JVM, the http:proxyHost and http:proxyPort properties are automatically set by the JVM using the settings in the Internet Explorer. If the use of proxies is not required for WebSphere MQ Everyplace, set the http.proxySet Java property to false.

MQeUdpipBasicAdapter
Provides support for reading and writing to the network using the UDP protocol. This adapter uses only one port on the server. The behaviour of this adapter is particularly sensitive to the various Java property settings, as detailed in the WebSphere MQ Everyplace Java Programming Reference.

MQeWESAuthenticationAdapter
Provides support for passing HTTP requests through WebSphere MQ Everyplace authentication proxy servers and transparent proxy servers.

You can also write your own adapters to tailor WebSphere MQ Everyplace for your own environment. The next section describes some adapter examples that are supplied to help you with this task.

How to write adapters

This example is not intended as a replacement for the adatpers that are supplied with WebSphere MQ Everyplace, but as a simple introduction on how to create a communications adapter.

To use your communications adapter, you must specify the correct class name when creating the listener on the server queue manager, and specify the connection definition on the client queue manager.

All communications adapters must inherit from MQeCommunicationsAdapter and must implement the required methods. In order to show how this might be done we shall use the example adapter, `examples.adapters.MQeTcpipLengthGUIAdapter`. This is a simple example that accepts data to be written. It also places the data length and the amount of data to be written to standard out, at the front of the data. When the adapter reads data, the data length is written to standard out. Proper error checking and recovery is not carried out. This must be added to any adapter written by a user.

WebSphere MQ Everyplace adapters use the default constructor. For this reason, an `activate()` method is used in order to set up the adapter with an `open()` method used to prepare the adapter for communication.

The `activate()` method is called only once in the life-cycle of an adapter and is, therefore, used to set up the information from MQePropertyProvider. The

MQePropertyProvider looks internally to verify that the specified property is available. If it is not available, it checks the Java properties. In this way, it is possible for a user to specify a property that may be set by the application or JVM command line. The MQeCommunicationsAdapter provides two variables that allow the adapter to identify its role within the communications conversation:

- If the adapter is being used by the MQeListener, the variable `listeningAdapter` is set to true.
- If the adapter has been created by the listening adapter in response to an incoming request, the `responderAdapter` variable is set to true.

The following code, taken from the `activate()` method, shows how to obtain the information from the MQePropertyProvider.

```
    if (!listeningAdapter) {
      // if we are not a listening adapter we need the
   address of the server
      address = info.getProperty
      (MQeCommunicationsAdapter.COMMS_ADAPTER_ADDRESS);
    }
```

The `open()` method is called before each conversation and must, therefore, be used to set information that needs to be reset for each request or response. For example, an adapter that is not persistent needs to create a socket each time it is opened. The following code shows the use of the variables that identify the role of the adatper role within the conversation:

```
      if (listeningAdapter && null == serverSocket) {
      serverSocket = new ServerSocket(port);
    } else if (!responderAdapter && null == mySocket) {
      mySocket = new Socket(InetAddress.getByName(address), port);
    }
```

Once the `activate()` and `open()` methods have been called, the listening adapter `waitForContact` method is called. This method must wait at named location. In an IP network, this will be a named port. When a request is received, a new adapter is created.

Note: This method must set the listeningAdapter to false and the responderAdapter to true.

Once the adapter has been set up correctly, you must must returned it to the caller. The following code shows how to do this:

```
 MQeTcpipLengthGUIAdapter clientAdapter =
      (MQeTcpipLengthGUIAdapter)
    MQeCommunicationsAdapter.createNewAdapter(info);

     // set the boolean variables so the adapter
  // knows it is a responder. the listening
     // variable will have been set to true as
  // the MQePropertyProvider has the relevant
     // information to create
  // this listening adapter.  We must therefore reset the
     // listeningAdapter variable to false and the
  //responderAdapter variable to true.
    clientAdapter.responderAdapter = true;
    clientAdapter.listeningAdapter = false;

    // Assign the new socket to this new adapter
    clientAdapter.setSocket(clientSocket);
    return clientAdapter;
```

The initiator adapter and responder adapter are responsible for the main part of the conversation. The initiator starts the conversation. The responder is created by the listening adapter, reads the request that is passed back to WebSphere MQ Everyplace, which then writes a response. The adapter determines how the read and the write are undertaken. The example uses a BufferedInputStream and a BufferedOutputStream.

Note: Use a a non-blocking mode of reading and writing. This enables the adapter to respond to requests to shutdown.

The following code, taken from the `waitForContact()` method, shows how the non-blocking read can be written. As WebSphere MQ Everyplace supports all Java runtime environments we are unable to use Java version 1.4 specific classes for our examples, although this version does contain new non-blocking classes

```
do {
      try {
        clientSocket = serverSocket.accept();
      } catch (InterruptedIOException iioe) {
         if (MQeThread.getDemandStop()) {
            throw iioe;
         }
      }
    } while (null == clientSocket);
```

An example of a simple communications adapter

This example uses the standard Java classes to manipulate TCPIP and adds a protocol of its own on top. This protocol has a header consisting of a four byte length of the data in the data packet followed by the actual data. This is so that the receiving end knows how much data to expect.

This example is not meant as a replacement for the adapters that are supplied with WebSphere MQ Everyplace but rather as a simple introduction into how to create communications adapters. In reality, much more care should be taken with error handling, recovery, and parameter checking. Depending on the WebSphere MQ Everyplace configuration used, the supplied adapters may be sufficient.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, that is the name of the host, port number and the output stream objects.

Note: With communications, ensure that the connection information is correct. For example, the http connection in J2ME has no timeout implementation. In J2SE, the client times out with an IO Exception. In Midp the server times out. If the default read-timeout has been increased for the J2SE client, the same exception is thrown, that is `com.ibm.mqe.MQeException: Data: (code=7)`. This is because the server writes back the exception to the client and the client cannot restore this data.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyTcpipAdapter extends MQeAdapter
  {
  protected     String               host         = "";
  protected     int                  port         = 80;
  protected     Object               readLock     = new Object( );
  protected     ServerSocket         serversocket = null;
  protected     Socket               socket       = null;
  protected     BufferedInputStream  stream_in    = null;
  protected     BufferedOutputStream stream_out   = null;
  protected     Object               writeLock    = new Object( );
```

Next the `activate` method is coded. This is the method that extracts from the file descriptor the name of the target network address if a connector, or the listening port if a listener. The fileDesc parameter contains the adapter class name or alias name, and any network address data for the adapter for example `MyTcpipAdapter:127.0.0.1:80`. The thisParam parameter contains any parameter data that was set when the connection was defined by administration, the normal value would be "?Channel". The thisOpt parameter contains the adapter setup options that were set by administration, for example `MQe_Adapter_LISTEN` if this adapter is to listen for incoming connections.

```
public void   activate( String   fileDesc,
                        Object   thisParam,
                        Object   thisOpt,
                        int      thisValue1,
                        int      thisValue2 ) throws Exception
  {
  super.activate( fileDesc,
                  thisParam,
                  thisOpt,
                  thisValue1,
                  thisValue2 );
  /* isolate the TCP/IP address -
       "MyTcpipAdapter:127.0.0.1:80"     */
  host = fileId.substring( fileId.indexOf( ':' ) + 1 );
  i    = host.indexOf( ':' );
/* find delimiter     */
  if ( i > -1 )
/* find it ?          */
    {
    port = (new Integer( host.substring( i + 1 ) )).intValue( );
    host = host.substring( 0, i );
    }
  }
```

The `close` method needs to be defined to close the output streams and flush any remaining data from the stream buffers. `Close` is called many time during a session between a client and a server, however, when the channel has completely finished with the adapter it calls WebSphere MQ Everyplace with the option `MQe_Adapter_FINAL`. If the adapter is to have one socket connection for the life of the channel then the call with `MQe_Adapter_FINAL` set, is the one to use to actually close the socket, other calls should just flush the buffers. If however a new socket is to be used on each request, then each call to WebSphere MQ Everyplace should close the socket, subsequent `open` calls should allocate a new socket:

```
 public void    close(  Object  opt ) throws Exception
   {
   if ( stream_out   != null )
/* output stream ?     */
     {
     stream_out.flush();
/* empty the  buffers  */
     stream_out.close();
/* close it            */
     stream_out = null;
/* clear               */
     }
   if ( stream_in    != null )
/* input stream ?      */
     {
     stream_in.close();
/* close it            */
     stream_in = null;
/* clear               */
     }
   if ( socket       != null )
/* socket ?           */
     {
     socket.close();
/* close it           */
     socket = null;
/* clear              */
     }
   if ( serversocket != null )
/* serversocket ?     */
     {
     serversocket.close();
/* close it           */
     serversocket = null;
/* clear              */
     }
   host = "";
   port = 80;
   }
```

The `control` method needs to be coded to handle an `MQe_Adapter_ACCEPT` request, to accept an incoming connect request. This is only allowed if the socket is a listener (a server socket). Any options that were specified for the listen socket (excluding `MQe_Adapter_LISTEN`) are copied to the socket created as a result of the accept. This is accomplished by the use of another control option `MQe_Adapter_SETSOCKET` this allows a socket object to be passed to the adapter that was just instantiated.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LISTEN      ) &&
         checkOption( opt, MQe.MQe_Adapter_ACCEPT ) )
      {
      /* CtrlObj - is a string representing the
       file descriptor of the */
      /*           MQeAdapter object to be returned e.g. "MyTcpip:"   */
      Socket  ClientSocket = serversocket.accept();
  /* wait connect  */
      String  Destination  = (String) ctrlObj;
  /* re-type object*/
      int i  = Destination.indexOf( ':' );
      if ( i < 0 )
        throw new MQeException( MQe.Except_Syntax,
                               "Syntax:" + Destination );
      /* remove the Listen option   */
      String NewOpt = (String) options;
  /* re-type to string   */
      int j  = NewOpt.indexOf( MQe.MQe_Adapter_LISTEN );
      NewOpt = NewOpt.substring( 0, j ) +
           NewOpt.substring
      ( j + MQe.MQe_Adapter_LISTEN.length( ) );
           MQeAdapter Adapter = MQe.newAdapter
              ( Destination.substring( 0,i+1 ),
                                          parameter,
                                          NewOpt + MQe_Adapter_ACCEPT,
                                          -1,
                                          -1 );
      /* assign the new socket to this new adapater  */
      Adapter.control( MQe.MQe_Adapter_SETSOCKET, ClientSocket);
      return( Adapter );
      }
    else
    if ( checkOption( opt, MQe.MQe_Adapter_SETSOCKET ) )
      {
      if ( stream_out != null )  stream_out.close();
      if ( stream_in  != null )  stream_in .close();
      if ( ctrlObj    != null )
  /* socket supplied ?*/
        {
        socket     = (Socket) ctrlObj;
  /* save the socket     */
        stream_in  = new BufferedInputStream (socket.getInputStream ());
        stream_out = new BufferedOutputStream(socket.getOutputStream());
        }
    else
      return( super.control( opt, ctrlObj ) );
  }
```

The open method needs to check for a listening socket or a connector socket and create the appropriate socket object. Reinitialization of the input and output streams is achieved by using the `control` method, passing it a new socket object. The opt parameter may be set to `MQe_Adapter_RESET`, this means that any previous operations are now complete any new reads or writes constitute a new request.

```
  public void  open( Object opt ) throws Exception
    {
    if ( checkOption( MQe.MQe_Adapter_LISTEN ) )
      serversocket = new ServerSocket( port, 32 );
    else
      control( MQe.MQe_Adapter_SETSOCKET,
       new Socket( host, port ) );
    }
```

The `read` method can take a parameter specifying the maximum record size to be read.

This examples calls internal routines to read the data bytes and do error recovery (if appropriate) then return the correct length byte array for the number of bytes read. Care needs to be taken to ensure that only one read at a time occurs on this socket. The opt parameter may be set to:

MQe_Adapter_CONTENT
read any message content

MQe_Adapter_HEADER
read any header information

```
{ public byte[] read( Object opt, int recordSize ) throws Exception

    int Count = 0;
 /* number bytes read   */
    synchronized ( readLock )
 /* only one at a time  */
      {
      if ( checkOption(opt, MQe.MQe_Adapter_HEADER )  )
        {
        byte lreclBytes[] = new byte[4];
 /* for the data length */
        readBytes( lreclBytes, 0, 4 );
 /* read the length     */
        int  recordSize = byteToInt( lreclBytes, 0, 4 );
        }
      if ( checkOption( opt, MQe.MQe_Adapter_CONTENT   ) )
        {
        byte Temp[] = new byte[recordSize];
 /* allocate work array */
        Count = readBytes( Temp, 0, recordSize);/* read data   */
        }
      }
    if ( Count < Temp.length )
 /* read all length ?   */
```

```
     Temp    = MQe.sliceByteArray( Temp, 0, Count );
   return ( Temp );
/* Return the data     */
   }
```

The `readByte` method is an internal routine designed to read a single byte of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
 protected int readByte( ) throws Exception
   {
   int intChar    = -1;
/* input characater    */
   int RetryValue =  3;
/* error retry count   */
   int Retry = RetryValue + 1;
/* reset retry count   */
   do{
/* possible retry      */
     try
/* catch io errors     */
       {
       intChar = stream_in.read();
/* read a character    */
       Retry   = 0;
 /* dont retry          */
       }
     catch ( IOException e )
/* IO error occured    */
       {
       Retry   = Retry - 1;
/* decrement           */
       if ( Retry == 0 )  throw e;
/* more attempts ?     */
       }
     } while ( Retry != 0 );
/* more attempts ?     */
   if ( intChar == -1 )
/* end of file ?       */
     throw new EOFException();
/* ... yes, EOF        */
   return( intChar );
/* return the byte     */
   }
```

The `readBytes` method is an internal routine designed to read a number of bytes of data from the socket and to attempt to retry any errors a specific number of times, or throw an end of file exception if there is no more data to be read.

```
 protected int readBytes( byte buffer[],
         int offset, int recordSize )
   throws Exception
   {
   int RetryValue = 3;
   int i = 0;
/* start index         */
   while ( i < recordSize )
/* got it all in yet ? */
     {
/* ... no              */
     int NumBytes = 0;
/* read count          */
     /* retry any errors based on the QoS Retry value */
     int Retry = RetryValue + 1;
/* error retry count   */
     do{
/* possible retry      */
```

```
        try
 /* catch io errors     */
          {
          NumBytes = stream_in.read( buffer,
       offset + i, recordSize - i );
          Retry    = 0;
 /* no retry           */
          }
        catch ( IOException e )
 /* IO error occured    */
          {
          Retry    = Retry - 1;
 /* decrement          */
          if ( Retry == 0 )  throw e;
 /* more attempts ?     */
          }
        } while ( Retry != 0 );
 /* more attempts ?     */
      /* check for possible end of file  */
      if ( NumBytes < 0 )
 /* errors ?           */
        throw new EOFException( );
 /* ... yes            */
      i = i + NumBytes;
 /* accumulate         */
      }    return ( i );
 /* Return the count    */
    }
```

The `readln` method reads a string of bytes terminated by a 0x0A character it will ignore 0x0D characters.

```
    {
    synchronized ( readLock )
 /* only one at a time  */
      {
      /* ignore the 4 byte length   */
      byte lreclBytes[] = new byte[4];          /* for the data length */
      readBytes( lreclBytes, 0, 4 );
/* read the length     */

      int intChar    = -1;
 /* input characater    */
      StringBuffer Result = new StringBuffer( 256 );
      /* read Header from input stream          */
      while ( true )
 /* until "newline"     */
        {
        intChar = readByte( );
/* read a single byte  */
        switch ( intChar )
/* what character      */
          {
          case -1:
/* ... no character    */
            throw new EOFException();
/* ... yes, EOF        */
          case 10:
/* eod of line         */
            return( Result.toString() );
/* all done            */
          case 13:
/* ignore              */
            break;
          default:
/* real data           */
            Result.append( (char) intChar );
/* append to string    */
          }
/* end of line ?       */
        }
      }
    }
```

The `status` method returns status information about the adapter. In this example it returns for the option `MQe_Adapter_NETWORK` the network type (TCPIP), for the option `MQe_Adapter_LOCALHOST` it returns the tcpip local host address.

```
public String status( Object  opt ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_NETWORK ) )
      return( "TCPIP" );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_LOCALHOST ) )
      return( InetAddress.getLocalHost( ).toString() );
    else
    return( super.status( opt ) );
    }
```

The `write` method writes a block of data to the socket. It needs to ensure that only one write at a time can be issued to the socket. In this example it calls an internal routine `writeBytes` to write the actual data and perform any appropriate error recovery.

The opt parameter may be set to:

MQe_Adapter_FLUSH
flush any data in the buffers

MQe_Adapter_HEADER
write any header records

MQe_Adapter_HEADERRSP
write any header response records

```
  public void write( Object opt, int recordSize, byte data[] )
    throws Exception
    {
    synchronized ( writeLock )
 /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) ||
           checkOption( opt, MQe.MQe_Adapter_HEADERRSP )  )
        writeBytes( intToByte( recordSize ), 0, 4 );
 /* write length*/
      writeBytes( data, 0, recordSize );
 /* write the data      */
      if ( checkOption( opt, MQe.MQe_Adapter_FLUSH  ) )
        stream_out.flush( );
 /* make sure it is sent */
      }

    }
```

The `writeBytes` is an internal method that writes an array (or partial array) of bytes to a socket, and attempt a simple error recovery if errors occur.

```
protected void writeBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
    if ( buffer != null )
 /* any data ?          */
      {
```

```
     /* break the data up into manageable chuncks */
     int i = 0;
/* Data index          */
     int j = recordSize;
/* Data length         */
     int MaxSize    = 4096;
/* small buffer        */
     int RetryValue = 3;
/* error retry count   */
     do{
/* as long as data     */
       if ( j < MaxSize )
/* smallbuffer ?       */
         MaxSize = j;
       int Retry = RetryValue + 1;
/* error retry count   */
       do{
/* possible retry      */
          try
/* catch io errors     */
           {
           stream_out.write( buffer, offset + i, MaxSize );
           Retry = 0;
/* don't retry         */
           }
         catch ( IOException e )
/* IO error occured    */
           {
           Retry = Retry - 1;
/* decrement           */
           if ( Retry == 0 )  throw e;
/* more attempts ?     */
           }
         } while ( Retry != 0 );
/* more attempts ?     */

       i = i + MaxSize;
/* update index        */
       j = j - MaxSize;
/* data left           */
       } while ( j > 0 );
/* till all data sent  */
     }
   }
```

The `writeLn` method writes a string of characters to the socket, terminating with 0x0A and 0x0D characters.

The opt parameter may be set to:

MQe_Adapter_FLUSH
flush any data in the buffers

MQe_Adapter_HEADER
write any header records

MQe_Adapter_HEADERRSP
write any header response records

```
 public void writeln( Object opt, String data ) throws Exception
   {
   if ( data == null )
/* any data ?          */
     data = "";
   write( opt, -1, MQe.asciiToByte( data + "\r\n" ) );
/* write data  */
   }
```

This is now a complete (though very simple) tcpip adapter that will communicate to another copy of itself one of which was started as a listener and the other started as a connector.

An example of a simple message store adapter

This example creates an adapter for use as an interface to a message store. It uses the standard Java i/o classes to manipulate files in the store.

This example is not meant as a replacement for the adapters that are supplied with WebSphere MQ Everyplace but rather as a simple introduction into how to create a message store adapter.

A new class file is constructed, inheriting from MQeAdapter. Some variables are defined to hold this adapter's instance information, such as the name of the file/message and the location of the message store.

The MQeAdapter constructor is used for the object, so no additional code needs to be added for the constructor.

```
public class MyMsgStoreAdapter extends    MQeAdapter
                               implements FilenameFilter
  {
  protected String  filter   = "";
 /* file type filter    */
  protected String  fileName = "";
 /* disk file name      */
  protected String  filePath = "";
 /* drive and directory */
  protected boolean reading  = false;
/* opened for reading  */
  protected boolean writing  = false;
```

Because this adapter implements FilenameFilter the following method must be coded. This is the filtering mechanism that is used to select files of a certain type within the message store.

```
  public boolean accept( File dir, String name )
    {
    return( name.endsWith( filter ));
    }
```

Next the `activate` method is coded. This is the method that extracts, from the file descriptor, the name of the directory to be used to hold all the messages.

The Object parameter on the method call may be an attribute object. If it is, this is the attribute that is used to encode and/or decode the messages in the message store.

The Object options for this adapter are:

- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

Any other options should be ignored.

```
public void   activate( String  fileDesc,
                        Object  param,
                        Object  options,
                        int     value1,
                        int     value2 ) throws Exception
```

```
    {
    super.activate( fileDesc, param, options, lrecl, noRec );
    filePath    = fileId.substring( fileId.indexOf( ':' ) + 1 );
    String Temp = filePath;
 /* copy the path data  */
    if ( filePath.endsWith( File.separator ) )
 /* ending separator ?  */
      Temp      = Temp.substring( 0, Temp.length( ) -
                                  File.separator.length( ) );
    else
      filePath  = filePath + File.separator;
 /* add separator       */
    File diskFile = new File( Temp );
    if ( ! diskFile.isDirectory( ) )
 /* directory ?         */
      if ( ! diskFile.mkdirs( ) )
 /* does mkDirs work ?  */
        throw new MQeException( MQe.Except_NotAllowed,
                          "mkdirs '" + filePath + "' failed" );
    filePath = diskFile.getAbsolutePath( ) + File.separator;
    this.open( null );
    }
```

The `close` method disallows reading or writing.

```
public void close( Object opt ) throws Exception
    {
    reading  = false;
/* not open for reading*/
    writing  = false;
/* not open for writing*/
    }
```

The `control` method needs to be coded to handle an `MQe_Adapter_LIST` that is, a request to list all the files in the directory that satisfy the filter. Also to handle an `MQe_Adapter_FILTER` that is a request to set a filter to control how the files are listed.

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LIST   ) )
      return( new File( filePath ).list( this ) );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER ) )
      {
      filter = (String) ctrlObj;
 /* set the filter      */
      return( null );
 /* nothing to return   */
      }
    else
    return( super.control( opt, ctrlObj ) );
 /* try ancestor        */
    }
```

The erase method is used to remove a message from the message store.

```
  public void erase( Object opt ) throws Exception
    {
    if ( opt instanceof String )
 /* select file ?       */
      {
      String FN = (String) opt;
 /* re-type the option  */
      if ( FN.indexOf( File.separator ) > -1 )
 /* directory ?         */
        throw new MQeException( MQe.Except_Syntax,
          "Not allowed" );
      if ( ! new File( filePath + FN ).delete( ) )
        throw new MQeException( MQe.Except_NotAllowed,
          "Erase failed" );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
          "Not supported" );
    }
```

The open method sets the Boolean values that permit either reading of messages or writing of messages.

```
public void open( Object opt ) throws Exception
    {
    this.close( null );
 /* close any open file */
    fileName = null;
 /* clear the filename  */
    if ( opt instanceof String )
 /* select new file ?   */
      fileName = (String) opt;
 /* retype the name     */
    reading  = checkOption( opt, MQe.MQe_Adapter_READ   ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    writing  = checkOption( opt, MQe.MQe_Adapter_WRITE  ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    }
```

The readObject method reads a message from the message store and recreates an object of the correct type. It also decrypts and decompresses the data if an attribute is supplied on the activate call. This is a special function in that a request to read a file that satisfies the matching criteria specified in the parameter of the read, returns the first message it encounters that satisfies the match.

```
public Object readObject( Object opt ) throws Exception
    {
    if ( reading )
      {
      if ( opt instanceof MQeFields )
        {
        /* 1. list all files in the directory */
        /* 2. read each file in turn and restore as a Fields object */
        /* 3. try an equality check - if equal then return that object */
        String List[] = new File( filePath ).list( this );
        MQeFields Fields = null;
        for ( int i = 0; i < List.length; i = i + 1 )
          try
            {
            fileName = List[i];
   /* remember the name   */
            open( fileName );
   /* try this file       */
            Fields = (MQeFields) readObject( null );
            if ( Fields.equals( (MQeFields) opt ) )
   /* match ?     */
              return( Fields );
            }
          catch ( Exception e )
   /* error occured       */
            {
            }
   /* ignore error        */
        throw new MQeException( Except_NotFound, "No match" );
        }
      /* read the bytes from disk   */
      File diskFile = new File( filePath + fileName );
      byte data[] = new byte[(int) diskFile.length()];
      FileInputStream InputFile = new FileInputStream( diskFile );
      InputFile.read( data );             /* read the file data  */
      InputFile.close( );                 /* finish with file    */
      /* possible Attribute decode of the data       */
      if ( parameter instanceof MQeAttribute )
  /* Attribute encoding ?*/
        data = ((MQeAttribute) parameter).decodeData( null,
                                                      data,
                                                      0,
                                                      data.length );
      MQeFields FieldsObject = MQeFields.reMake( data, null );
      return( FieldsObject );
      }
    else
      throw new MQeException( MQe.Except_NotSupported,
           "Not supported" );
    }
```

The `status` method returns status information about the adapter. In this examples it can return the filter type or the file name.

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER   ) )
      return( filter   );
    if ( checkOption( opt, MQe.MQe_Adapter_FILENAME ) )
      return( fileName );
    return( super.status( opt ) );
    }
```

The `writeObject` method writes a message to the message store. It compresses and encrypts the message object if an attribute is supplied on the `activate` method call.

```
public void writeObject( Object  opt,
                           Object  data ) throws Exception
    {
    if ( writing && (data instanceof MQeFields) )
      {
      byte dump[] = ((MQeFields) data).dump( );
 /* dump object */
      /* possible Attribute encode of the data                        */
      if ( parameter instanceof MQeAttribute )
        dump = ((MQeAttribute) parameter).encodeData( null,
                                                      dump,
                                                      0,
                                                      dump.length );
      /* write out the object bytes                                   */
      File diskFile = new File( filePath + fileName );
      FileOutputStream OutputFile = new FileOutputStream( diskFile );
      OutputFile.write( dump );                /* write the data      */
      OutputFile.getFD().sync( );              /* synchronize disk    */
      OutputFile.close();                      /* finish with file    */
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

This is now a complete (though very simple) message store adapter that reads and writes message objects to a message store.

Variations of this adapter could be coded for example to store messages in a database or in nonvolatile memory.

The WebSphere Everyplace Suite (WES) communications adapter

WebSphere MQ Everyplace provides sophisticated security that allows applications to run over HTTP, through the protection of an Internet firewall. The purpose of the WebSphere Everyplace communications adapter is to allow WebSphere MQ Everyplace applications to authenticate themselves with the WebSphere Everyplace authentication proxy and thus allow messages to flow through it. Figure 2 shows a basic scenario with two applications communicating over the Internet through the WebSphere Everyplace authentication proxy.

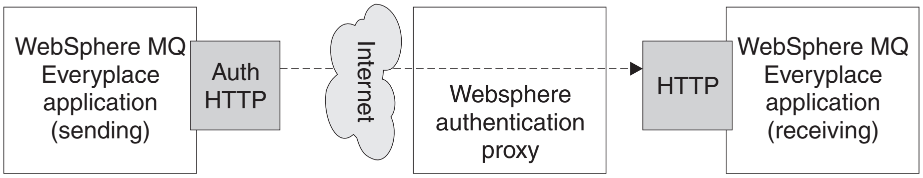

WebSphere MQ Everyplace application (sending)
Auth HTTP
Internet
Websphere authentication proxy
HTTP
WebSphere MQ Everyplace application (receiving)

Figure 2. Applications communicating through the WebSphere authentication proxy

The WebSphere MQ Everyplace adapter acts as the Auth HTTP adapter on the sending application. The receiving application could use either the same adapter or the standard HTTP adapter provided with WebSphere MQ Everyplace.

However, the real value of WebSphere MQ Everyplace is that it allows asynchronous messaging to occur in a typically synchronous environment. It is possible to gather enqueued requests from the receiving application and deal with them time-independently. Figure 3 shows how incoming requests could be made to reach WebSphere MQ servers asynchronously.

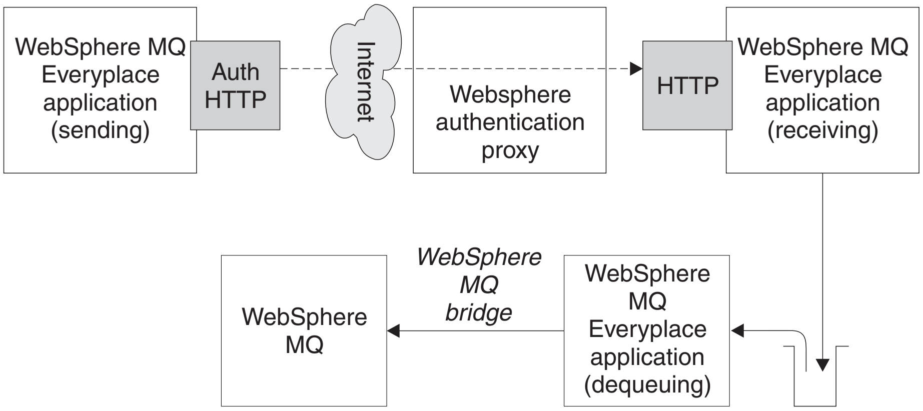

WebSphere MQ Everyplace application (sending)
Auth HTTP
Internet
Websphere authentication proxy
HTTP
WebSphere MQ Everyplace application (receiving)
WebSphere MQ
WebSphere MQ bridge
WebSphere MQ Everyplace application (dequeuing)

Figure 3. Applications communicating asynchronously through the WebSphere Authentication Proxy

In each of these environments the WebSphere authentication proxy is adding the ability to control access to the receiving applications. The adapter code supports this by adding (application-supplied) user ID and password information to each outgoing HTTP request. The WebSphere authentication proxy accepts these requests and verifies that the supplied credentials are valid for the current environment. If the credentials are valid the proxy forwards the request to the receiving application.

The WebSphere Everyplace adapter files

In a standard WebSphere MQ Everyplace installation the WebSphere Everyplace adapter consists of, and is supported by the following files:

...\Java\com\ibm\mqe\adapters\MQeWESAuthenticationAdapter.class
- The WebSphere Everyplace adapter class.

...\Java\examples\application\Example7.class
- Compiled example application that uses the adapter

...\Java\examples\application\Example7.java
- Source for the example application

...\Java\examples\adapters\WESAuthenticationGUIAdapter.class
- Compiled example adapter that adds a user interface to the WebSphere Everyplace adapter. As with other example classes, this class is not meant as a replacement for the base WES adapter class, but rather as a demonstration of how to tailor the WES adapter to suit your requirements.

...\Java\examples\adapters\WESAuthenticationGUIAdapter.java
- Source for the example adapter

If your environment `CLASSPATH` variable is set to find all classes within the WebSphere MQ Everyplace Java folder, the WebSphere Everyplace adapter class files should be accessible from within the Java environment. If the files are not accessible, issue a command such as:

```
set CLASSPATH=%CLASSPATH%;c:\mqe\java
```

This will make the new classes visible to Java. (The exact format of this command may vary from system to system.) Once this is complete you should be able to use the WebSphere Everyplace adapter classes in the same way as any other WebSphere MQ Everyplace classes.

Using the WebSphere Everyplace adapter

This section provides information on how to use the WebSphere Everyplace adapter. The information is divided into three parts:

General operation
This describes in detail, how to use the adapter in your applications

Using the Authentication Dialog Example
This describes how to use an example class, examples.adapters.WESAuthenticationGUIAdapter. This class is derived from the base WES adapter class and provides a small user interface to collect the ID and password of the user.

Using the Application Example
This describes how to use the supplied example file examples.application.Example7 which is configured to use the base WES adapter.

The information in this section assumes that both the WebSphere Everyplace authentication proxy and WebSphere MQ Everyplace have been installed and configured correctly. It is also assumed that an WebSphere MQ Everyplace server queue manager and an WebSphere MQ Everyplace client queue manager have been configured.

WebSphere adapter

General Operation

1. Configure the client queue manager to send messages using the new adapter by modifying the client queue manager’s configuration .ini file so that the `Network` alias points to com.ibm.mqe.adapters.MQeWESAuthenticationAdapter. Use the following command:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```
2. Configure the server queue manager to decode the stream of data that the Client Adapter supplies using either the new adapter or the standard HTTP adapter. Do this by changing the line in the server queue manager’s configuration .ini file so that the `Network` alias points to either com.ibm.mqe.adapters.MQeWESAuthenticationAdapter or com.ibm.mqe.adapters.MQeTcpipHttpAdapter. Use one of the following commands:

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
   ```
3. Modify the client queue manager code so that the required user ID and password are set before the first network operation is started. For example, insert the following line near the top of your code:

   ```
   com.ibm.mqe.adapters.MQeWESAuthenticationAdapter.
   setBasicAuthorization("myUserId@myRealm", "myPassword");
   ```

   Replace the parameters with a valid WES Server user ID and password.

   You also need to add code to catch the new MQeException `Except_Authenticate` after each network operation, in case the supplied credentials were invalid.
4. Check that the client queue manager can still send messages to the server queue manager without going through the proxy.
5. Configure the client machine to send HTTP requests through the proxy. Depending on how WES has been configured, the adapter will need to work with either a *transparent proxy* or an *authentication proxy*.

   As a *transparent proxy*

   In this mode, the WES server acts as a simple HTTP proxy. In this case, you need to set the following Java application system properties that relate to proxy information:

   http.proxyHost
   Must be set to the host name of the WES proxy

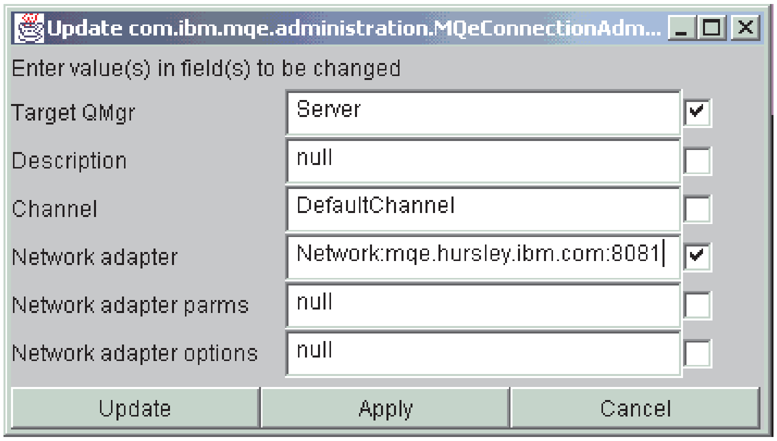
   http.proxyPort
   Must be set to the name of the port that the proxy is listening on

   http.proxySet
   Must be set to `true`, which tells the adapter to use transparent proxy mode

   The above parameters can be set by adding the following to your Java application:

   ```
     System.getProperties( ).put( "http.proxySet",  "true" );
    System.getProperties( ).put( "http.proxyHost", "wes.hursley.ibm.com" );
    System.getProperties( ).put( "http.proxyPort", "8082" );
   ```

The client queue manager's connection to the target WebSphere MQ Everyplace server is similar to a connection that doesn't use the WES proxy.



Enter value(s) in field(s) to be changed

Target QMgr: Server ☑
Description: null ☐
Channel: DefaultChannel ☐
Network adapter: Network:mqe.hursley.ibm.com:8081 ☑
Network adapter parms: null ☐
Network adapter options: null ☐

Figure 4. Administration interface panel

You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

As an *Authentication Proxy*

In this mode, the WES server forwards requests to services, based on the URL that you supply. For example, you may want requests for `http://wes.hursley.ibm.com/mqe` to be forwarded to an WebSphere MQ Everyplace queue manager running on `mqe.hursley.ibm.com:8082`.

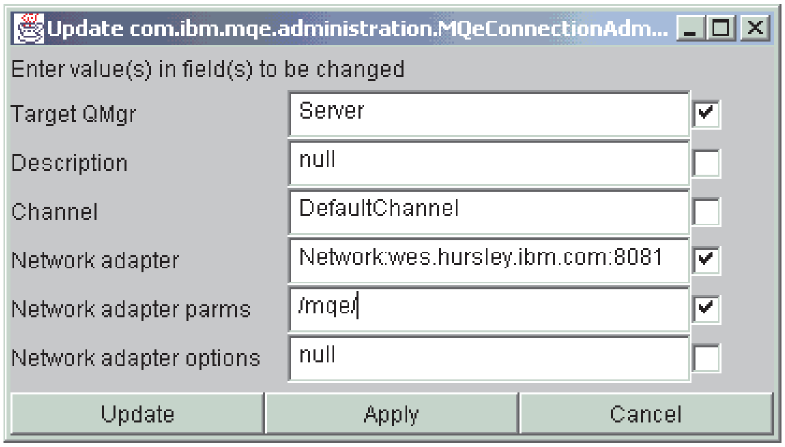
To set this up from WebSphere MQ Everyplace you need to update the client's *`connection`* reference to the server.

Target network adapter
Should point to the Authentication Proxy machine and port

Network adapter parameters
Should contain the pathname to the required service

If you are using the WebSphere MQ Everyplace Example Administration tool, select **Connection** and then **Update** to configure this.

WebSphere adapter



Enter value(s) in field(s) to be changed

Target QMgr	Server	☑
Description	null	☐
Channel	DefaultChannel	☐
Network adapter	Network:wes.hursley.ibm.com:8081	☑
Network adapter parms	/mqe/	☑
Network adapter options	null	☐

Figure 5. Administration interface panel

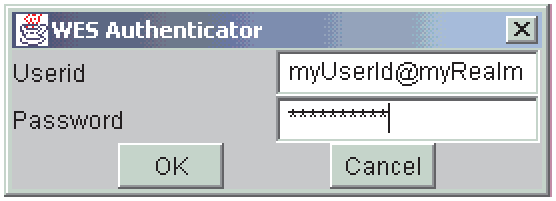
Note: The reference to the WES Server is entered in the **Network adapter** field, and the pathname is entered in the **Network adapter parms** field.

You need to restart the server and client queue managers for the new settings to take effect. The client should then be able to send messages to the server through the proxy.

Using the Authentication Dialog Example

The following information describes the use of the example class file, examples.adapters.WESAuthenticationGUIAdapter. This class adds a small user interface to the base WES adapter function.

1. Follow steps (1) and (2) of the 'General operation' procedures, but substitute 'WESAuthenticationGUIAdapter' for 'WESAuthenticationAdapter' in step (1).
2. Configure the client's TCP/IP settings as in step (5) of 'General operation'.



Userid	myUserId@myRealm
Password	**********

Figure 6. WebSphere Everyplace Suite adapter user dialog

The client should now able to send messages to the server using the WESAuthenticationGUIAdapter. This adapter intercepts write calls to the WES adapter, and on the first request it pops up a dialog box that prompts for user ID and password information.

When the user clicks on **OK** or presses the **Enter** key, the **setBasicAuthorization()** method is called with the values from the **userid** and **password** fields. The **write()** is then forwarded on to the underlying WES adapter. The dialog box also has a **Cancel** button which, when selected, cancels the current write operation by not forwarding the request to the WES adapter. This causes an `MQeException (Except_Stopped)` to be thrown.

If authentication fails, the dialog box is redisplayed on the next **write()** along with any information provided by the server. In order to learn of an authentication failure, the example adapter intercepts **read()** calls and catches any `Except_Authenticate` MQeExceptions coming from the adapter.

Note: Web browsers do not generally send authentication information on the first flow. This typically results in a `401` or `407` response that contains the realm information. Only then does the browser send the authenticated request. User clients may wish to follow this convention.

Using the Application Example

The following information describes the use of the example application file, examples.application.Example7. This example behaves in a similar way to the MQSeries Everyplace programming example examples.application.Example1 and uses the basic WES adapter for communications.

1. Follow steps (1) and (2) of the 'General operation' procedures.
2. Configure the client's TCP/IP settings as in step (5) of 'General operation'.
3. Edit the example file ...\Java\examples\application\Example7.java inserting a valid user ID and password, and then recompile the application.
4. Restart the server.
5. Run the Example7 program using the following command:

   ```
   java examples.application.Example7 Server client.ini
   ```

   where

   Server
   is the name of the remote queue manager (that the client already knows how to reach)

   client.ini
   points to the client's .ini configuration file.

   The application starts the client queue manager, authenticates with the proxy, puts a message to server and then get a message from the server.

WebSphere adapter

Chapter 3. Rules

Websphere MQ Everyplace uses rules to allow applications to monitor and modify the behavior of some of its major components. Rules take the form of methods on Java classes or functions in C methods that are loaded when WebSphere MQ Everyplace components are initialized.

A component's rules are invoked at certain points during its execution cycle. Rules methods with particular signatures are expected to be available, so when providing implementations of rules, ensure that you use the correct signatures.

Default or example rules are provided for all relevant WebSphere MQ Everyplace components. You can customize these to satisfy particular user requirements. Within the Java codebase, the `MQeQueueProxy` interface provides the user with accessor methods for queues, allowing the user to interact with queues in certain rule methods.

Rules may be grouped into the following categories:
- Queue manager rules.
- Queue rules.
- Attribute rules. For information on attribute rules, refer to the WebSphere MQ Everyplace Application Programming Guide and the WebSphere MQ Everyplace Configuration Guide.
- Bridge rules. For information on bridge rules, refer to the WebSphere MQ Everyplace Application Programming Guide.

Rules may also be categorized into two groups depending upon whether they can affect application behavior, 'modification' rules, or are intended for notification purposes only, notification' rules.

Queue manager rules

Queue manager rules are invoked when:
- The queue manager is activated
- The queue manager is closed
- A queue is added to the queue manager (Java codebase only)
- A queue is removed from the queue manager (Java codebase only)
- A put message operation occurs
- A get message operation occurs
- A delete message operation occurs
- An undo message operation occurs
- The queue manager is triggered to transmit any pending messages, as described in Transmission rules

Loading and activating queue manager rules

Java codebase

Queue manager rules are loaded, or changed whenever a queue manager administration message containing a request to update the queue manager rule class is received.

If a queue manager rule has already been applied to the queue manager, the existing rule is asked whether it may be replaced with a different rule. If the answer is yes, the new rule is loaded and activated. A restart of the queue manager is not required.

The `QueueManagerUpdater` command-line tool in the package `examples.administration.commandline` shows how to create such an administration message.

C codebase

The user's rules module is loaded and initialized when the queue manager is loaded into memory. This occurs as a result of calls either to `mqeAdministrator_QueueManager_create()` or to `mqeQueueManager_new()`. The set-up steps are as follows:

- The application must register a rules alias, linking the rules alias to the rules module name and entry point, by using mqeClassAlias_add(), for example:

```
#define RULES_ALIAS "myAlias"
    #define MODULE_NAME "myRulesModule.dll"
    #define ENTRY_POINT "myRules_new"
    ...

    mqeString_newUtf8(pExceptBlock,
        &rulesAlias, RULES_ALIAS);
    mqeString_newUtf8(pExceptBlock,
        &moduleName, MODULE_NAME);
    mqeString_newUtf8(pExceptBlock,
        &entryPoint, ENTRY_POINT);
    mqeClassAlias_add(pExceptBlock,
        rulesAlias, moduleName, entryPoint);
```

- The rules alias must be included in the queue manager start-up parameters passed to either `mqeAdministrator_QueueManager_create()` or `mqeQueueManager_new()`, for example.:

```
MQeQueueManagerParms      qmParams;
    qmParams.hQueueStore = msgStore; /* String parameters for the*/
             /*location of the msg store */
    qmParams.hQueueManagerRules = rulesAlias; /* add in rules alias */


    /* Indicate what parts of the structure have been set */
    qmParams.opFlags = QMGR_Q_STORE_OP | QMGR_RULES_OP;

    ...

    rc = mqeAdministrator_QueueManager_create(hAdmin,pExceptBlock,
                                &hQM,qmName, &qmParms, &regParms);
```

- An initialization function or entry point must be supplied by the user. The following is an example of an initialization function for a rules implementation. The members of the parameter structures are documented in the WebSphere MQ Everyplace C Programming Reference.

```
MQERETURN myRules_new( MQeRulesNew_in_ * pInput,MQeRulesNew_out_ * pOutput) {

    MQERETURN rc = MQERETURN_OK;
    /* declare an instance of the private data */
  /*structure passed around between rules invocations. */
  /*This holds user data which is 'global' between rules. */
    myRules * myData = NULL;

    /* allocate the memory for the structure */
    myData = malloc(sizeof(myRules));
    if(myData != NULL)    {
  /* map user rules implementations to
   function pointers in output parameter structure */
        pOutput->fPtrActivateQMgr   = myRules_ActivateQMgr;
        pOutput->fPtrCloseQMgr      = myRules_CloseQMgr;
        pOutput->fPtrDeleteMessage  = unitTestRules_DeleteMessage;
        pOutput->fPtrGetMessage     = myRules_getMessage;
        pOutput->fPtrPutMessage     = myRules_putMessage;
        pOutput->fPtrTransmitQueue  = myRules_TransmitQueue;
        pOutput->fPtrTransmitQMgr   = myRules_TransmitQMgr;
        pOutput->fPtrActivateQueue  = myRules_activateQueue;
        pOutput->fPtrCloseQueue     = myRules_CloseQueue;
        pOutput->fPtrMessageExpired = myRules_messageExpired;

        /* initialize data in the private data structure */
        mydata->carryOn = MQE_TRUE;
        mydata->hAdmin = NULL;
        mydata->hThread = NULL;
        mydata->ifp = NULL;
        mydata->triggerInterval = 15000;

        /* now assign the private data structure to */
   /*the output parameter structure variable */
        pOutput->pPrivateData = (MQEVOID *)mydata;
    }
    else {
        /* We had a problem so clear up any strings in the structure -
            none in this case */
    }

    return rc;
}
```

The rules module is unloaded when the queue manager is freed. Note that, unlike the java codebase, the rules implementation is linked to the execution lifecycle of a single queue manager and may not be replaced during the course of this lifecycle.

Using queue manager rules

This section describes some examples of the use of queue manager rules.

In the Java codebase, a user provides an implementation of a rule method by subclassing the `MQeQueueManagerRule` class.

In the C codebase, a user maps rules functions to relevant rules function pointers. These pointers are passed into the rules initialization function, which is also the entry point to the user’s rules module.

For a description of all parameters passed to rules functions in the C codebase, see the WebSphere MQ Everyplace C Programming Reference.

Example put message rule

This first example shows a put message rule that insists that any message being put to a queue using this queue manager must contain a WebSphere MQ Everyplace message ID field:

Java codebase

```
/* Only allow msgs containing an ID field to be placed on the Queue */

public void putMessage( String destQMgr, String destQ, MQeMsgObject msg,
                       MQeAttribute attribute, long confirmId )         {
    if ( !(msg.Contains( MQe.Msg_MsgId )) )    {
         throw new MQeException( Except_Rule, "Msg must contain an ID" );
    }
}
```

C codebase

```
 MQERETURN myRules_putMessage( MQeRulesPutMessage_in_ * pInput,
                              MQeRulesPutMessage_out_ * pOutput)    {
    // Only allow msgs containing an ID field to be placed on the Queue
    MQERETURN rc = MQERETURN_OK;
    MQEBOOL contains = MQE_FALSE;

    MQeExceptBlock * pExceptBlock=(MQeExceptBlock*)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    rc = mqeFields_contains(pInput->hMsg,pExceptBlock,
         &contains, MQE_MSG_MSGID);
    if(MQERETURN_OK == rc && !contains)     {
        SET_EXCEPT_BLOCK( pExceptBlock,
                          MQERETURN_RULES_DISALLOWED_BY_RULE,
                          MQEREASON_NA);
    }
}
```

Notice the manner in which the exception block instance is retrieved from the output parameter structure and then set with the appropriate return and reason codes. This is the way in which the rule function communicates with the application, thus modifying application behaviour.

Example get message rule

The next example rule is a get message rule that insists that a password must be supplied before allowing a get message request to be processed on the queue called OutboundQueue. The password is included as a field in the message filter passed into the `getMessage()` method.

Java codebase

```
/* This rule only allows GETs from 'OutboundQueue',
  if a password is  */
/* supplied as part of the filter */

public void getMessage( String destQMgr,
        String destQ, MQeFields filter,
                       MQeAttribute attr, long confirmId )         {
   super.getMessage( destQMgr, destQ, filter, attr, confirmId );
   if (destQMgr.equals(Owner.GetName()
        && destQ.equals("OutboundQueue"))  {
       if ( !(filter.Contains( "Password" ) )      {
           throw new MQeException( Except_Rule,
            "Password not supplied" );
       }
       else    {
           String pwd = filter.getAscii( "Password" );
           if ( !(pwd.equals( "1234" )) )   {
               throw new MQeException( Except_Rule,
              "Incorrect password" );
           }
       }
   }
}
```

C codebase

```
MQERETURN myRules_getMessage( MQeRulesGetMessage_in_ * pInput,
                              MQeRulesGetMessage_out_ * pOutput)    {
    MQeStringHndl hQueueManagerName, hCompareString, hCompareString2,
                  hFieldName, hFieldValue;
    MQEBOOL isEqual = MQE_FALSE;
    MQEBOOL contains = MQE_FALSE;
    MQeQueueManagerHndl hQueueManager;

    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
                           (MQeExceptBlock *)
          (pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* get the current queue manager */
     rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                                  &hQueueManager);
    if(MQERETURN_OK == rc) {
        // if the destination queue manager is the local queue manager
              rc = mqeQueueManager_getName( hQueueManager,
                                        pExceptBlock,
                                        &hQueueManagerName );
        if(MQERETURN_OK == rc)     {
            rc = mqeString_equalTo(pInput->hQueue_QueueManagerName,
                                   pExceptBlock,
                                   &isEqual,
                                   hQueueManagerName);
            if(MQERETURN_OK == rc && isEqual)    {
                // if the destination queue name is "OutboundQueue"
                rc = mqeString_newUtf8(pExceptBlock,
                                       &hCompareString,
                                       "OutboundQueue");
                rc = mqeString_equalTo(pInput->hQueueName,
                                       pExceptBlock,
                                       &isEqual,
                                       hCompareString);
                if(MQERETURN_OK == rc && isEqual)    {
                    // password required for this queue
                    MQEBOOL contains = MQE_FALSE;
                    rc = mqeString_newUtf8(pExceptBlock,
                                           &hFieldName,
                                           "Password");
                    rc = mqeFields_contains(pInput->hFilter,
                                            pExceptBlock,
                                            &contains,
                                            hFieldName);
                    if(MQERETURN_OK == rc && contains == MQE_FALSE)     {
                        SET_EXCEPT_BLOCK(pExceptBlock,
                                         MQERETURN_RULES_DISALLOWED_BY_RULE,
                                         MQEREASON_NA);
                    }
                    else   {
                            // parse password, etc.
                    }
                }
            }
        }
    }
}
```

This previous rule is a simple example of protecting a queue. However, for more comprehensive security, you are recommended to use an authenticator. An authenticator allows an application to create access control lists, and to determine who is able to get messages from queues.

Example remove queue rule

The next example rule is called when a queue manager administration request tries to remove a queue. The rule is passed an object reference to the proxy for the queue in question. In this example, the rule checks the name of the queue that is passed, and if the queue is named PayrollQueue, the request to remove the queue is refused.

Java codebase

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueueProxy queue )
throws Exception    {
    if ( queue.getQueueName().equals( "PayrollQueue" ) )    {
        throw new MQeException( Except_Rule,
          "Can't delete this queue" );
    }
}
```

C codebase
This rule is not implemented in the C codebase.

Transmission rules

A message that is put to a remote queue that is defined as synchronous is transmitted immediately. Messages put to remote queues defined as asynchronous are stored within the local queue manager until the queue manager is triggered into transmitting them. The queue manager can be triggered directly by an application. The process can be modified or monitored using the queue manager's transmission rules.

The transmission rules are a subset of the queue manager rules. The two rules that allow control over message transmission are:

`triggerTransmission()`
This rule determines whether to allow message transmission at the time when the rule is called. This can be used to veto or allow the transmission of all messages, that is, either all or none are allowed to be transmitted.

`transmit()`
This rule makes a decision to allow transmission on a per queue basis for asynchronous remote queues. For example, this makes it possible only to transmit the messages from queues deemed to be high priority. The `transmit()` rule is only called if the `triggerTransmission()` rule returns successfully.

Trigger transmission rule

WebSphere MQ Everyplace calls the `triggerTransmission` rule when transmission is triggered. This occurs when the queue manager `triggerTransmission` method or function is explicitly called from an application or a rule. Additionally, in the Java codebase, the rule may be invoked when a message is put onto a remote asynchronous queue. The default rule behavior in both Java and C allows the attempt to transmit pending messages to proceed. For example, this is the default Java rule in com.ibm.mqe.MQeQueueManagerRule:

```
/* default trigger transmission rule -
  always allow transmission */
public boolean triggerTransmission(int noOfMsgs,
             MQeFields msgFields ){
    return true;
}
```

The return code from this rule tells the queue manager whether or not to transmit any pending messages. A return code of true means "transmit", while a return code of false means "do not transmit at this time".

The user may over-ride the default behavior by implementing their own triggerTransmission() rule. A more complex rule can decide whether or not to transmit immediately based on the number of messages awaiting transmission on asynchronous remote queues. The following example shows a rule that only allows transmission to continue if there are more than 10 messages pending transmission.

Java codebase

```
/* Decide to transmit based on number of pending messages */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields ) {
    if(noOfMsgs > 10)    {
        return true; /* then transmit */
    }
    else {
        return false; /* else do not transmit */
    }
}
```

C codebase

```
/* The following function is mapped to the
  fPtrTransmitQMgr function pointer  */
/* in the user's initialization function output parameter structure. */

MQERETURN myRules_TransmitQMgr( MQeRulesTransmitQMgr_in_  * pInput,
                        MQeRulesTransmitQMgr_out_ * pOutput)    {
    MQeExceptBlock * pExceptBlock =
                        (MQeExceptBlock*)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

   /* allow transmission to be triggered only
   if the number of pending messages > 10  */
    if(pInput->msgsPendingTransmission <= 10) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                         MQERETURN_RULES_DISALLOWED_BY_RULE,
         MQEREASON_NA);
    }
}
```

Transmit rule

The `transmit()` rule is only called if the `triggerTransmission()` rule allows transmission. It returns a value of true or `MQERETURN_OK`. The `transmit()` rule is called for every remote queue definition that holds messages awaiting transmission. This means that the rule can decide which messages should be transmitted on a queue by queue basis. The example rule below only allows message transmission from a queue if the queue has a default priority greater than 5. If a message has not been assigned a priority before being placed on a queue, it is given the queue's default priority.

Java codebase

```
public boolean transmit( MQeQueueProxy queue )    {
    if ( queue.getDefaultPriority() > 5 )    {
        return (true);
    }
    else    {
        return (false);
    }
}
```

C codebase

```
/* The following function is mapped to the fPtrTransmitQueue function*/
/* pointer in the user's initialization
/* function output parameter structure. */

MQERETURN myRules_TransmitQueue( MQeRulesTransmitQueue_in_ * pInput,
                        MQeRulesTransmitQueue_out_ * pOutput) {
    MQERETURN rc = MQERETURN_OK;
    MQEBYTE queuePriority;

    MQeRemoteAsyncQParms queueParms = REMOTE_ASYNC_Q_INIT_VAL;
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
                   (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* inquire upon the default priority of the queue*/
    /* specify the subject of the inquire
   in the queue parameter structure*/
  queueParms.baseParms.opFlags =  QUEUE_PRIORITY_OP ;

    rc =  mqeAdministrator_AsyncRemoteQueue_inquire(myData->hAdmin,
                                 pExceptBlock,
                                 pInput->hQueueName,
                                 pInput->hQueue_QueueManagerName,
                                 &queueParms);
    // if the default priority is less than 6, disallow the operation
    if(MQERETURN_OK == rc
   && queueParms.baseParms.queuePriority < 6) {
        SET_EXCEPT_BLOCK(pExceptBlock,
                         MQERETURN_RULES_DISALLOWED_BY_RULE,
         MQEREASON_NA);
    }
}
```

A sensible extension to this rule can allow all messages to be transmitted at 'off-peak' time. This allows only messages from high-priority queues to be transmitted during peak periods.

A more complex example

The following example assumes that the transmission of the messages takes place over a communications network that charges for the time taken for transmission. It also assumes that there is a cheap-rate period when the unit-time cost is lower. The rules block any transmission of messages until the cheap-rate period. During the cheap-rate period, the queue manager is triggered at regular intervals.

Java codebase

```
import com.ibm.mqe.*;
import java.util.*;
/**
* Example set of queue manager
  rules which trigger the transmission
* of any messages waiting to be sent.
*
```

```
* These rules only trigger the
  transmission of messages if the current
* time is between the values defined
  in the variables cheapRatePeriodStart
* and cheapRatePeriodEnd
* (This example assumes that transmission
   will take place over a
* communication network which charges
   for the time taken to transmit)
*/
public class ExampleQueueManagerRules extends MQeQueueManagerRule
implements Runnable
{
    // default interval between triggers is 15 seconds
    private static final long
     MILLISECS_BETWEEN_TRIGGER_TRANSMITS = 15000;

    // interval between which we c
   heck whether the queue manager is closing down.
      private static final long
     MILLISECS_BETWEEN_CLOSE_CHECKS = 1000 ;

    // Max wait of ten seconds to kill off
    the background thread when
    // the queue manager is closing down.
    private static final long
    MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS = 10000;

    // Reference to the control block used to
    communicate with the background thread
    // which does a sleep-trigger-sleep-trigger loop.
    // Note that freeing such blocks for garbage
    collection will not stop the thread
    // to which it refers.
    private Thread th = null;

    // Flag which is set when shutdown of
    the background thread is required.
    // Volatile because the thread using the
    flag and the thread setting it to true
    // are different threads, and it is
    important that the flag is not held in
    // CPU registers, or one thread will
    see a different value to the other.
    private volatile boolean toldToStop = false;
  //cheap rate transmission period start and end times
    protected int cheapRatePeriodStart = 18;  /*18:00 hrs */
    protected int cheapRatePeriodEnd = 9;     /*09:00 hrs */
}
```

The `cheapRatePeriodStart` and `cheapRatePeriodEnd` functions define the extent of this cheap rate period. In this example, the cheap-rate period is defined as being between 18:00 hours in the evening until 09:00 hours the following morning.

The constant `MILLISECS_BETWEEN_TRIGGER_TRANSMITS` defines the period of time, in milliseconds, between each triggering of the queue manager. In this example, the trigger interval is defined to be 15 seconds.

The triggering of the queue manager is handled by a background thread that wakes up at the end of the triggerInterval period. If the current time is inside the cheap rate period, it calls the `MQeQueueManager.triggerTransmission()` method to initiate an attempt to transmit all messages awaiting transmission. The background thread is created in the `queueManagerActivate()` rule and stopped in the `queueManagerClose()` rule. The queue manager calls these rules when it is activated and closed respectively.

```
/**
* Overrides MQeQueueManagerRule.queueManagerActivate()
* Starts a timer thread
*/
public void queueManagerActivate()throws Exception {
    super.queueManagerActivate();
    // background thread which triggers transmission
    th = new Thread(this, "TriggerThread");
    toldToStop = false;
    th.start();    // start timer thread
}


/**
* Overrides MQeQueueManagerRule.queueManagerClose()
* Stops the timer thread
*/
 public void queueManagerClose()throws Exception {
    super.queueManagerClose();

    // Tell the background thread to stop,
   as the queue manager is closing now.
    toldToStop = true ;

    // Now wait for the background thread,
   if it's not already stopped.
    if ( th != null)  {
        try {
        // Only wait for a certain time before
    giving up and timing out.
         th.join( MAX_WAIT_FOR_BACKGROUND_THREAD_MILLISECONDS );

       // Free up the thread control block for garbage collection.
            th = null ;
        } catch (InterruptedException e) {
            // Don't propogate the exception.
            // Assume that the thread will stop shortly anyway.
        }
    }
}
```

The code to handle the background thread looks like this:

```
/**
* Timer thread
* Triggers queue manager every interval until thread is stopped
*/
public void run()    {
    /* Do a sleep-trigger-sleep-trigger loop until the */
  /*  queue manager closes or we get an exception.*/
    while ( !toldToStop) {
        try {

            // Count down until we've waited enough
         // We do a tight loop with a smaller granularity because
         // otherwise we would stop a queue manager from closing quickly
         long timeToWait = MILLISECS_BETWEEN_TRIGGER_TRANSMITS ;
         while( timeToWait > 0 && !toldToStop ) {

           // sleep for specified interval
                Thread.sleep( MILLISECS_BETWEEN_CLOSE_CHECKS );

                // We've waited for some time.
        Account for this in the overall wait.
                 timeToWait -= MILLISECS_BETWEEN_CLOSE_CHECKS ;
           }
   if( !toldToStop && timeToTransmit()) {
             // trigger transmission on QMgr (which is rule owner)
             ((MQeQueueManager)owner).triggerTransmission();
                       }
           } catch ( Exception e ) {
               e.printStackTrace();
           }
        }
     }
}
```

The variable owner is defined by the class `MQeRule`, which is the ancestor of `MQeQueueManagerRule`. As part of its startup process, the queue manager activates the queue manager rules and passes a reference to itself to the rules object. This reference is stored in the variable owner.

The thread loops indefinitely, as it is stopped by the `queueManagerClose()` rule, and it sleeps until the end of the `MILLISECS_BETWEEN_TRIGGER_TRANSMITS` interval period. At the end of this interval, if it has not been told to stop, it calls the `timeToTransmit()` method to check if the current time is in the cheap-rate transmission period. If this method succeeds, the queue manager's `triggerTransmission()` rule is called.The `timeToTransmit` method is shown in the following code:

```
protected boolean timeToTransmit()    {
    /* get current time */
    Calendar calendar = Calendar.getInstance();
    calendar.setTime( new Date() );
    /* get hour */
    int hour = calendar.get( Calendar.HOUR_OF_DAY );
    if ( hour >= cheapRatePeriodStart || hour
      < cheapRatePeriodEnd ) {
        return true; /* cheap rate */
    }
    else    {
        return false; /* not cheap rate */
    }
}
```

C codebase

The C example emulates the java codebase example. While the native C codebase is entirely single-threaded, it is possible for the user to write platform-specific code in which threads are created. In this example of a user-written queue manager activate rule, a thread is spawned which loops, sleeping for a period of time defined in a `triggerInterval` variable and then, providing it has not been asked to stop, checking that we are in a cheap rate period prior to attempting to trigger transmission. Data, which is required between rules invocations, is stored in the rule's private data structure. Refer to the WebSphere MQ Everyplace C Programming Reference on how rules private data is carried around between rules invocations. The queue manager's close rule function is used to provide the thread's terminating condition, setting a boolean switch, `carryOn` to `MQE_FALSE`. This switch can be initialized to `MQE_TRUE` in the rules initialization function. This function waits until the thread is suspended before passing control back to the application.

The private data structure passed between rule invocations is as follows:

```
struct myRules_st_ {
// rules instance structure
   MQeAdministratorHndl hAdmin;
// administrator handle to carry around between

// rules functions
    MQEBOOL carryOn;
// used for trigger transmission thread
    MQEINT32 triggerInterval;
// used for trigger transmission thread
    HANDLE hThread;
// handle for the trigger transmission thread
};

typedef struct myRules_st_ myRules;

The queue manager activate rule:

MQEVOID myRules_activateQueueManager( MQeRulesActivateQMgr_in_ * pInput,
                                    MQeRulesActivateQMgr_out_ * pOutput) {
    // retrieve exception block - passed from application
    MQeExceptBlock * pExceptBlock = (MQeExceptBlock *)
             (pOutput->pExceptBlock);

    // retrieve private data structure passed
   between user's rules invocations
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeQueueManagerHndl hQueueManager;
    MQERETURN rc = MQERETURN_OK;

    rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                &queueManager);
    if(MQERETURN_OK == rc) {
        // set up the private data administrator
     handle using the retrieved
        // application queue manager handle.
     This is done here rather than in
        // the rules initialization function as the
     queue manager has not yet been
   // activated fully when the rules
   //initialization function is invoked.
        rc = mqeAdministrator_new(pExceptBlock,
          &myData>hAdmin,hQueueManager);
    }
    if(MQERETURN_OK == rc) {
        DWORD tid;
        // Launch thread to govern calls to trigger transmission
        myData->hThread = (HANDLE) CreateThread(NULL,
                                            0,
                                            timeToTrigger,
                                            (MQEVOID *)myData,
                                            0,
                                            &tId);
                if(myData>hThread == NULL) {
            // thread creation failed
            SET_EXCEPT_BLOCK(pExceptBlock,
           MQERETURN_RULES_ERROR,
           MQEREASON_NA);
        }
    }
}
```

The `timeToTrigger` function provides the equivalent functionality of the `run()` method in the java example above. Notice the use of the private data variable `carryOn`, type `MQEBOOL`, as one of the conditions for the while loop to continue. Once this variable has a value of `MQE_FALSE`, the while loop will terminate, causing the thread to terminate when the function is exited.

```
DWORD _stdcall timeToTrigger(myRules * rulesStruct) {

    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    MQeExceptBlock exceptBlock;
    myRules * myData = (myRules *)rulesStruct;
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);

    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* so long as there is not a grave
    internal error and the termination
           condition has not been set */
        while(!(EC(&exceptBlock) ==
        MQERETURN_QUEUE_MANAGER_ERROR &&
                ERC(&exceptBlock) ==
        MQEREASON_INTERNAL_ERROR) &&
                myData->carryOn == MQE_TRUE) {
           /* Are we in a cheap rate transmission period? */
            if(timeToTransmit())    {
                /* if so,  attempt to trigger transmission */
                rc = mqeQueueManager_triggerTransmission(hQueueManager,
                                        &exceptBlock);

                /* wait for the duration of the trigger interval */
                Sleep(myData->triggerInterval);
            }
        }
    }
    return 0;
}
```

The `timeToTransmit()` function returns a boolean to indicate whether or not we are in a cheap transmission period:

```
MQEBOOL timeToTransmit() {

    SYSTEMTIME timeInfo;
    GetLocalTime(&timeInfo);

    if (timeInfo.wHour >= 18 || timeInfo.wHour < 9) {
        return MQE_TRUE;
    } else {
        return MQE_FALSE;
    }
}
```

It would probably be a better idea to define constants for the cheap rate interval boundary times and carry these around in the rules private data structure also but that has been not been done here for reasons of clarity.

The function returns `MQE_TRUE` to suggest that we are in a cheap rate period, that is between the hours of 18:00 and 09:00. A return value of `MQE_TRUE` is one of the prerequisites for transmission to be triggered in `timeToTrigger()`. Finally, the queue manager close rule is used to terminate the thread.Notice that one of the conditions for termination of the `timeToTrigger()` function is for the boolean variable carryOn to have a value of `MQE_FALSE`. In the close function, the value of carryOn is set to false. But, there may still be a considerable lapse of time between when this value is set to `MQE_FALSE` and when the `timeToTrigger()` function is exited. The value of triggerInterval + the time taken to perform a `triggerTransmission` operation. Also, we wait for the thread to terminate in this function. We also call `triggerTransmission()` one more time in case there are still some pending messages.

```
 MQEVOID myRules_CloseQMgr( MQeRulesCloseQMgr_in_ * pInput,
                            MQeRulesCloseQMgr_out_ * pOutput)     {
    MQERETURN rc = MQERETURN_OK;
    MQeQueueManagerHndl hQueueManager;
    myRules * myData = (myRules *)pInput->pPrivateData;
    DWORD result;
    MQeExceptBlock exceptBlock =
       *((MQeExceptBlock *)pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(&exceptBlock);
    // Effect the ending of the thread by
    setting the MQEBOOL continue to MQE_FALSE
    // This leads to a return from timeToTrigger()
     and hence the implicit call
    // to _endthread
    myData->carryOn = MQE_FALSE;

    /* wait for the thread in any case */
    result = WaitForSingleObject(myData->hThread, INFINITE);

    /* retrieve the current queue manager */
    rc = mqeQueueManager_getCurrentQueueManager(&exceptBlock,
                 &hQueueManager);
    if(MQERETURN_OK == rc) {
        /* attempt to trigger transmission one
   /* last time to clean up queue */
        rc = mqeQueueManager_triggerTransmission(hQueueManager,
                 &exceptBlock);
    }
}
```

Activating asynchronous remote queue definitions

The queue manager can activate its asynchronous remote queue definitions and home server queues at startup time. In the Java codebase, activating asynchronous remote queue definitions results in an attempt to transmit any messages they contain, while activating home server queues results in an attempt to get any messages that are waiting on their assigned store-and-forward queue. The activateQueues() rule allows this behavior to be configured.

The default rule just returns true.

```
public boolean activateQueues()    {
    return true; /* activate queues on queue manager start-up */
}

/*As with other rules examples above,
  a check can be made to see if the current */
/* time is inside the cheap-rate transmission period.
  This information can then */
/* be used to determine whether queues should be activated or not.

public boolean activateQueues()    {
    if ( timeToTransmit() )    {
        return true;
    }
    else    {
        return false;
    }
}
```

If `activateQueues()` returns false, the remote queue definitions are only activated when a message is put onto them. Home server queues can be activated by calling the queue manager's `triggerTransmission()` method.

In the C codebase, activation of home server queues and asynchronous queues does not result in any attempts to transmit or pull down pending messages. Only explicit calls to the queue manager's `triggerTransmission()` function have this result. There is no implementation of an activateQueues rule in the C codebase. Activation of queues occurs at queue manager start-up.

Queue rules

In the Java codebase, each queue has its own set of rules. A solution can extend the behavior of these rules. All queue rules should descend from class com.ibm.mqe.MQeQueueRule.

In the C codebase, only a single set of rules is loaded. A user can implement different rules for different queues by loading other rules modules from the 'master' module. The master rules functions can then invoke the corresponding functions in any other modules as required.

Queue rules are called when:
- The queue is activated.
- The queue is closed.
- A message is placed on the queue using a put operation (Java codebase only).
- A message is removed from the queue using a get operation.
- A message is deleted from the queue using a delete operation (Java codebase only).
- The queue is browsed.
- An undo operation is performed on a message on the queue.
- A message listener is added to the queue (Java codebase only).
- A message listener is removed from the queue (Java codebase only).
- A message expires.
- An attempt is made to change a queue's attributes, that is authenticator, cryptor, compressor (Java codebase only).
- A duplicate message is put onto a queue.
- A message is being transmitted from a remote asynchronouse queue.

Using queue rules

This section describes some examples of the use of queue rules.

The first example shows a possible use of the message expired rule. Both queues and messages can have an expiry interval set. If this interval is exceeded, the message is flagged as being expired. At this point the messageExpired() rule is called. On return from this rule, the expired message is deleted.

In the following example, a copy of the message is put onto a Dead Letter Queue.

Java codebase

```
/* This rule puts a copy of any expired messages to a Dead Letter Queue */

public boolean messageExpired( MQeFields entry, MQeMsgObject msg )
         throws Exception   {
```

```
    /* Get the reference to the Queue Manager */
    MQeQueueManager qmgr = MQeQueueManager.getReference(
                           ((MQeQueueProxy)owner).getQueueManagerName());
    /* need to set re-send flag so that put of message
   to new queue isn't rejected */
    msg.putBoolean( MQe.Msg_Resend, true );
    /* if the message contains an expiry
    interval field - remove it */
    if ( msg.contains( MQe.Msg_ExpireTime )    {
        msg.delete( MQe.Msg_ExpireTime );
    }
    /* put message onto dead letter queue */
    qmgr.putMessage( null, MQe.DeadLetter_Queue_Name,
        msg, null, 0 );
    /* Return true. Note that no use is made
    of this return value - the message is
   always deleted but the return value is kept
   for backward compatibility */
    return (true);
}
```

C codebase

```
MQEVOID myRules_messageExpired( MQeRulesMessageExpired_in_ * pInput,
                          MQeRulesMessageExpired_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQeExceptBlock * pExceptBlock =
       (MQeExceptBlock *)(pOutput->pExceptBlock);

    MQEBOOL contains = MQE_FALSE;
    MQeFieldsHndl hMsg;
    MQeQueueManagerHndl hQueueManager;
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    /* Set re-send flag so that attempt to put
   message to new queue isn't rejected */
    // First, clone the message as the
  //input parameter is read-only
    rc = mqeFields_clone(pInput->hMsg, pExceptBlock,
          &hMsg);
    if(MQERETURN_OK == rc) {
        rc = mqeFields_putBoolean(hMsg, pExceptBlock,
                                  MQE_MSG_RESEND, MQE_TRUE);
        if(MQERETURN_OK == rc) {
            // if the message contains an expiry
      interval field - remove it
            rc = mqeFields_contains(hMsg, pExceptBlock,
             &contains,
                                    MQE_MSG_EXPIRETIME);
            if(MQERETURN_OK == rc && contains) {
                rc = mqeFields_delete(hMsg, pExceptBlock,
                                      MQE_MSG_EXPIRETIME);
            }
if(MQERETURN_OK == rc) {
                // put message onto dead letter queue
                MQeStringHndl hQueueManagerName;
                rc = mqeQueueManager_getCurrentQueueManager(pExceptBlock,
                                                    &hQueueManager);
                if(MQERETURN_OK == rc) {
          rc = mqeQueueManager_getName(hQueueManager,
                                                pExceptBlock,
                                               &hQueueManagerName);
                    if(MQERETURN_OK == rc) {
                        // use a temporary exception block as don't care
                        // if dead letter queue does not exist
                        MQeExceptBlock tempExceptBlock;
                        SET_EXCEPT_BLOCK_TO_DEFAULT(&tempExceptBlock);
                        rc = mqeQueueManager_putMessage( hQueueManager,
                                                  &tempExceptBlock,
                                                  hQueueManagerName,
                                      MQE_DEADLETTER_QUEUE_NAME,
                                                   hMsg, NULL, 0 );
                       (MQEVOID)mqeString_free(hQueueManagerName,
                 &tempExceptBlock);
                    }
                }
            }
        }
    }
}
```

The previous example sends any expired messages to the queue manager's dead-letter queue, the name of which is defined by the constant `MQe.DeadLetter_Queue_Name` in the Java codebase and `MQE_DEADLETTER_QUEUE_NAME` in the C codebase. It is worth noting that the queue manager rejects a put of a message that has previously been put onto another queue. This protects against a duplicate message being introduced into the Websphere MQ Everyplace network. So, before moving the message to the dead-letter queue, the rule must set the resend flag. This is done by adding the Java `MQe.Msg_Resend` or C `MQE_MSG_RESEND` field to the message.

The message expiry time field must be deleted before moving the message to the dead-letter queue. The following example shows how to log an event that occurs on the queue.

In the Java example, the event that occurs is the creation of a message listener, in the C example, it is a put message request. However, the principle is the same in both cases:

Java codebase

In the example, the queue has its own log file, but it is equally as valid to have a central log file that is used by all queues. The queue needs to open the log file when it is activated, and close the log file when the queue is closed. The queue rules, `queueActivate` and `queueClose` can be used to do this. The variable `logFile` needs to be a class variable so that both rules can access the log file.

```
/* This rule logs the activation of the queue */
public void queueActivate()    {
    try    {
     logFile = new LogToDiskFile( \\log.txt );
     log( MQe_Log_Information, Event_Activate, "Queue " +
     ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
     ((MQeQueueProxy)owner).getQueueName() + " active" );
    }
    catch( Exception e )    {
        e.printStackTrace( System.err );
    }
}

/* This rule logs the closure of the queue */
public void queueClose()    {
    try    {
      log( MQe_Log_Information, Event_Closed, "Queue " +
          ((MQeQueueProxy)owner).getQueueManagerName() + " + " +
          ((MQeQueueProxy)owner).getQueueName() + " closed" );
        /* close log file */
        logFile.close();
    }
    catch ( Exception e )    {
        e.printStackTrace( System.err );
    }
}
```

The addListener rule is shown in the following code. It uses the MQe.log method to add an Event_Queue_AddMsgListener event.

```
/* This rule logs the addition of a message listener */
public void addListener( MQeMessageListenerInterface listener,
         MQeFields filter ) throws Exception
   {
    log( MQe_Log_Information, Event_Queue_AddMsgListener,
        "Added listener on queue "
         + ((MQeQueueProxy)owner).getQueueManagerName() + "+"
         + ((MQeQueueProxy)owner).getQueueName() );
}
```

C codebase

In the C codebase example, a central log is set up for all queues using the queue activate and close rules. This log is then used to keep track of all putMessage operations. Because the log is shared between rules invocations, the information needed to access the log is stored in the rules private data structure - see the Websphere MQ Everyplace for Multiplatforms C Programming Reference for information on how the rules private data structure is passed around between rules invocations. In this case, the private data structure contains a file handle for passing between rules invocations:

```
struct myRulesData_  {
// rules instance structure
      MQeAdministratorHndl hAdmin;  /
 administrator handle to carry around between
// rules functions
      FILE * ifp;
// file handle for logging rules
};
typedef struct myRulesData_ myRules;
```

In the rules queue activate function, the file is opened and the activation of the queue logged:

```
MQEVOID myRules_activateQueue(MQeRulesActivateQueue_in_ * pInput,
                              MQeRulesActivateQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the input
    structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
        (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp == NULL)   {
  // initialized to NULL in the rules initialization function
        myData->ifp = fopen("traceFile.txt","w");
        rc =  mqeString_getUtf8(pInput->hQueueName,
            pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
              pExceptBlock, qName, &size);
             if(MQERETURN_OK ==
        rc && myData->ifp != NULL) {
                  fprintf(myData->ifp,
      "Activating queue %s \n", qName);
              }
         }
    }
}
```

In the rules queue close function, the file is closed after the closure of the queue is logged:

```
MQEVOID myRules_closeQueue(MQeRulesCloseQueue_in_ * pInput,
                           MQeRulesCloseQueue_out_ * pOutput)  {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName;
    MQEINT32 size;

    // recover the private data from the
   input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
     (MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
        pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
         pExceptBlock, qName, &size);
             if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp,
       "Closing queue %s \n", qName);
             }
        }
        fclose(myData->ifp);
        MyData->ifp = NULL;
    }
}
```

The rules put message function ensures that each put message operation is logged:

```
MQERETURN myRules_putMessage(MQeRulesPutMessage_in_ * pInput,
                             MQeRulesPutMessage_out_ * pOutput)    {
    MQERETURN rc = MQERETURN_OK;
    MQECHAR * qName, * qMgrName;
    MQEINT32 size;

    // recover the private data from the input structure parameter pInput
    myRules * myData = (myRules *)(pInput->pPrivateData);

    MQeExceptBlock * pExceptBlock =
(MQeExceptBlock *)(pOutput->pExceptBlock);
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);

    if(myData->ifp != NULL)   {
        rc =  mqeString_getUtf8(pInput->hQueueName,
            pExceptBlock, NULL, &size);
        if(MQERETURN_OK == rc) {
             qName = malloc(size);
             rc = mqeString_getUtf8(pInput->hQueueName,
             pExceptBlock, qName,&size);
        }
        if(MQERETURN_OK == rc)   {
            rc =  mqeString_getUtf8(pInput->hQueue_QueueManagerName,
             pExceptBlock,
                                 NULL, &size);
            if(MQERETURN_OK == rc) {
                qMgrName = malloc(size);
                rc = mqeString_getUtf8(pInput->hQueue_QueueManagerName,
                                    pExceptBlock,
              qMgrName, &size);
            }
        }
if(MQERETURN_OK == rc)  {
                 fprintf(myData->ifp, "Putting a message
         onto queue %s on queue
                         manager %s\n",qName, qMgrName);
        }
    }
    /* allow the operation to proceed regardless of what
   went wrong in this rule */
    SET_EXCEPT_BLOCK_TO_DEFAULT(pExceptBlock);
    return EC(pExceptBlock);
}
```

Chapter 4. Aliases

Aliases can be assigned for WebSphere MQ Everyplace queues to provide a level of indirection between the application and the real queues. For example, a queue can be given a number of aliases and messages sent to any of these names will be accepted by the queue.

Using queue aliases

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 cllient 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:

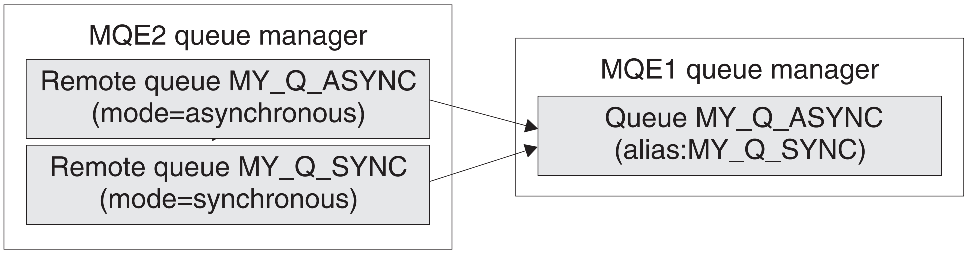
- A queue `Q1`
- An application that gets messages from `Q1`
- An application that puts messages to `Q1`

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 7. Two modes of transfer to a single queue

Using queue manager aliases

Aliases can be used for WebSphere MQ Everyplace queue managers, and can be used by application programs, to provide a level of indirection between the application and the real object.

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

Examples of queue manager aliasing

Addressing a queue manager with several different names

Suppose you have a queue manager `SERVER23QM` on the server `SAMPLEHOST`, listening on port `8082`. You have an application `SERVICEX` that accesses this queue manager, and wants to refer to the queue manager as `SERVICEXQM`. This can be achieved using an alias for the queue manager as follows:

- **Configure a connection on the SERVER23QM :**

  Connection Name/Target queue manager: `SERVICEXQM`

  Description: Alias definition to enable `SERVER23QM` to receive messages sent to `SERVICEXQM`

  Channel: "null"

  Network Adapter: "null"

  Network adapter options: "null"

- **Create a local queue on the `SERVER23QM` queue manager:**

  Queue Name: `SERVICEXQ`

  Queue Manager: `SERVER23QM`

  The server-side application takes messages from this queue, and process them, sending messages back to the client.

A WebSphere MQ Everyplace application can now put messages to the `SERVICEXQ` on either the `SERVER23QM` queue manager, or the `SERVICEXQM` queue manager. In either case, the message will arrive on the `SERVICEXQ`.

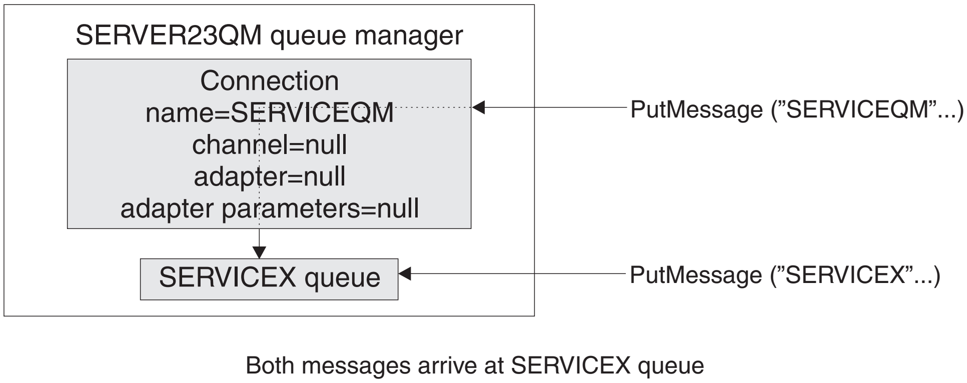


Figure 8. Addressing a queue manager with two different names

If the `SERVICEXQ` queue is moved to another queue manager, the connection alias can be set up on the new queue manager, and the applications do not need to be changed.

Different routings from one queue manager to another

Using the scenario just described, an WebSphere MQ Everyplace queue manager on a mobile device (`MOBILE0058QM`) can now access the `SERVICEXQ` queue in a number of different ways. Two examples are described here:

- **Aliasing on the sending side**

  Using this method of routing, the receiving queue manager does not know that the sending queue manager has given him an alias name. The aliasing is confined to the sending queue manager only.

  On the mobile device:

  - Create a connection from `MOBILE0058QM` to the `SERVER23QM` queue manager:

    `Connection name`
    `SERVER23QM`

    `Network Adapter parameter`
    `Network:SAMPLEHOST:8082`

  - Create an alias called `SERVICEXQM` for queue manager `SERVER23QM`

  When a message is sent from the mobile device application to the `SERVICEXQM` queue manager, WebSphere MQ Everyplace maps the `SERVICEXQM` name to `SERVER23QM` in the connection , and sends the message to the `SERVER23QM` queue manager.

  If the `Mobile58QM` then wished to send its messages to a different server queue manager, `Server24QM`, it would remove the alias `SERVICEXQM` from the `Server23QM` connection, and add it to a `Server24QM` connection. This has no impact on the receiving queue managers, or the sending applications.

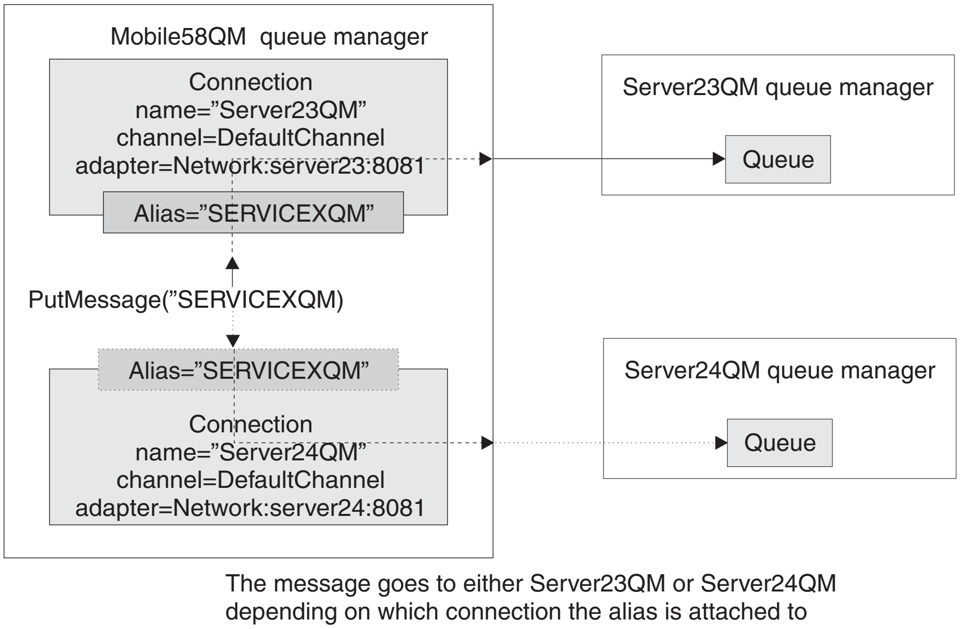


Figure 9. Addressing a queue manager with two different names

- **Virtual queue manager on the receiving side**

  Using this method, the sending queue managers think that its messages are routed through an intermediate queue manager before reaching the target queue manager. The target queue manager doesn't actually exist. The 'intermediate' queue manager captures all the message traffic for this virtual target queue manager.

  On the mobile device:

  - Create a connection from `MOBILE0058QM` to the `SERVER23QM` queue manager:

`Connection name`	`SERVER23QM`
`Network Adapter parameter`	`Network:SAMPLEHOST:8082`

  - Create a second connection to the `SERVICEXQM` that routes messages through the first connection:

`Connection name`	`SERVICEXQM`
`Network Adapter parameter`	`SERVER23QM`

    Note: This is not an alias. It is a *via routing*, indicating that messages headed for `SERVICEXQM` are to be routed via the `SERVER23QM` queue manager on the receiving side.

  The via routing on the mobile device causes any messages that are put to `SERVICEXQM` to be directed to `Server23QM`. `Server23QM` gets the messages and notes that they are destined for the `SERVICEXQM` queue manager. It resolves the `SERVICEXQM` name and finds that it is an alias which represents the `Server23QM` queue manager (itself). The `Server23QM` queue manager then accepts the messages and puts them onto the queue.

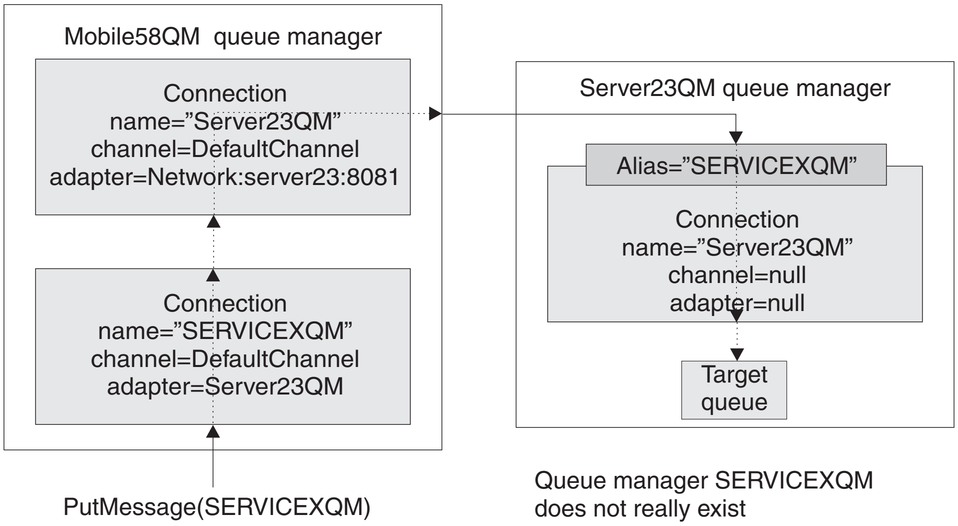


Figure 10. Addressing a queue manager with two different names

As an alternative to the above, you can keep the `SERVICEXQM` in existence, but move it from its original machine to the same machine (but a different JVM) as the `Server23QM` queue manager. `SERVICEXQM` needs to listen on a different port, so the connection from `Server23QM` to `SERVICEXQM` needs to be changed as well.

Chapter 5. Applying maintenance

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:

1. Completely uninstall the current level, and install the new level in the same directory. We recommend that you keep the install package for the current level to allow it to be restored later if necessary.
2. 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/.

Chapter 6. Trace and logging

This chapter provides information on trace and logging for WebSphere MQ Everyplace applications under the following headings.

- Trace information in the Java codebase
- Trace information in the C codebase

Trace information in the Java codebase

The trace mechanism provided and used by WebSphere MQ Everyplace has the following features:

- A pluggable interface to allow user-written trace handlers to be implemented if required.
- A veriety of implementations of the trace handler interface to cater for a veriety of uses. One such implementation supports a crude form of circular logging, so older trace information is discared when newer trace information becomes available. See the `com.ibm.mqe.trace.MQeTraceToBinaryFile` for more details.
- A separation between the trace point number, and the meaning, or textual representation of that trace point. This separation of the number from lengthy meanful string information allows for collection of the trace point numbers to be performed at run-time, and the rendering of that information to a readable format to be done off-line at a later time. This can mean trace information files are smaller and generated more quickly at the point of capture, but much larger and more accessable at the time they are read.
- Dynamic, run-time filtering of trace information.

Generating trace information

Tracing in the Java codebase is performed using the `com.ibm.mqe.MQeTrace class`. All calls to `com.ibm.mqe.MQeTrace.trace()` methods pass the following information:

- A number, by which the trace point can be identified.
- A group bit-mask, which identifies this trace point as being classified as part of one or more groups of trace points. This information is used in conjunction with the `MQeTrace.setFilter()` method, to allow unwanted trace information to be filtered-out at run-time. Many of the bits in the group bit-mask have a defined meaning. For example, if the `MQeTrace.GROUP_ERROR` bit is set, then the trace point indicates that an error is being reported. Multiple group bits can be set for the same trace point.
- A number of parameters. A `tostring()` method is invoked for each parameter, so that a string is extracted at run-time, and added to the trace point.

Classes shipped in WebSphere MQ Everyplace generate lots of trace information using these methods, such that the trace point numbers are all negative. We recommend that programs using this trace mechanism use positive numbers, or zero.

Several bit-fields are reserved for user applications, for example, the `MQeTrace.GROUP_MASK_USER_DEFINED` bits-fields. For convenience, `MQeTrace.GROUP_USER_DEFINED_1` maps to one such bit, for example:

```
MQeTrace.trace(this, (short) 1, MQeTrace.GROUP_ERROR |
        MQeTrace.GROUP_USER_DEFINED_1, thingToLog );
```

This statement implements a logical `AND` operation on the `GROUP_ERROR` and `GROUP_USER_DEFINED_1`, maintaining the run-time filter with the `MQeTrace` class. If the result is non-zero, then the corresponding method on the `MQeTraceHandler` interface class is called, if a handler has been set.

There are several variants of the `MQeTrace.trace()` method, including methods that trace different numbers of parameters with the trace point.

Capturing trace information

WebSphere MQ Everyplace does not automatically capture the trace information provided by the `MQeTrace.trace()` methods. The solution programmer must capture the trace messages. We strongly recommend that your solution includes a mechanism to allow the capture of WebSphere MQ Everyplace trace events, as this output may be requested by the IBM service teams when investigating any problems reported.

To capture WebSphere MQ Everyplace trace information, you need to ensure that

- A trace handler has been provided, and set using the `MQeTrace.setHandler()` method.
- The run-time filter maintained by the `MQeTrace.setFilter()` method is not excluding the information you want to capture.

The required trace handler must implement the MQeTraceHandler interface. WebSphere MQ Everyplace ships with several trace handlers, used for different purposes:

MQeTraceToReadable
This renders trace information to a `printstream` in a readable format.

MQeTraceToBinaryFile
This collects trace information into a file, or sequence of files.

MQeTraceFromBinaryFile
You can use this to render this binary information file format into readable text.

MQeTraceToBinaryMidp
Collects binary trace information when running inside a MIDP java environment.

```
  // Allocate a trace instance, so our handler
 //isn't garbage collected when its' on.
        myMQeTrace = new MQeTrace();

        // Allocate a trace handler
        // This one puts trace output to stdout by default.
        MQeTraceHandler handler = (MQeTraceHandler)
          new com.ibm.mqe.trace.MQeTraceToReadable();

        // Set this handler as the one MQe uses.
        MQeTrace.setHandler(handler);

        // Set the filter so we collect those
   //pieces of trace we are interrested in.
        // In this case, collect all the default trace information.
       MQeTrace.setFilter(MQeTrace.GROUP_MASK_DEFAULT);

    ...
```

```
// To end trace set the filter to zero and the handler to null
MQeTrace.setFilter(0);
     MQeTrace.setHandler(null);
```

This example shows the creation of a trace handler, `MQeTraceToReadable` in this case, and setting of the filter to capture the default trace informaton. This can result in lots of information being captured. You can use a more restrictive filter to capture only a subset of the data. For example, collecting errors, warnings, and user-coded trace points might be more appropriate:

```
MQeTrace.GROUP_ERROR | MQeTrace.GROUP_WARNING | MQeTrace.GROUP_MASK_USER_DEFINED
```

Notes:

1. The IBM Service team may ask you to use the `MQeTrace.GROUP_MASK_ALL` value when diagnosing a problem.
2. When using the MQeTraceToBinaryMidp tracehandler, you require an additional step to recover the trace. The MIDP tracehandler either stores the trace in a record store or in memory. Once trace has finished, call `sendDataToUrl()` to recover this binary data. By default, this sends the data to a servlet. For more information, refer to the `examples.trace.MQeTraceServlet` section of the WebSphere MQ Everyplace Java Programming Reference.

Writing your own trace handler

Solution providers may wish to write their own trace handlers, to

- Do more complex filtering.
- Store trace information in a different place or form to that used by the supplied trace handlers
- Re-route trace information generated through this mechanism to another trace capture mechanism. For example, the trace handlers supplied with WebSphere MQ Everyplace rely on function supplied by underlying classes:

  com.ibm.mqe.trace.MQeTracePointGroup
  This class holds information about a logical grouping of trace points.

  com.ibm.mqe.trace.MQeTraceRenderer
  Provides a programmatic way of managing a collection of tracePointGroups and tracePoints information. It provides methods to add or remove tracePointGroups, individual tracePoints to and from the collection of tracePointGroups, and collection of tracePoints.

  com.ibm.mqe.trace.MQeTracePoint
  A collection of information that describes a particular trace point.

The trace handlers in the product populate a series of MQeTracePointGroup objects with a collection of MQeTracePoint objects. The groups are added to the MQeTraceRenderer, and the MQeTraceRenderer is used to map from the trace point number passed on the `MQeTrace.trace()` methods, to a readable string.

The separation of the readable string from the trace point number allows the code to collect just the number, and separate the information collection stage from the stage that renders to readable strings.

Where possible, the trace handlers supplied also gather stack trace information when a java.lang.throwable is passed as a parameter to the `MQeTrace.trace()` method.

You can implement the trace handler interface, and intercept trace information from your application and the WebSphere MQ Everyplace system classes. For examples of this, refer to the following classes in the WebSphere MQ Everyplace Java Programming Reference:

- `examples.trace.MQeTrace`
- `examples.trace.MQeTraceToFile`

You can add trace points to existing trace point groups, or to your own trace point groups. You can add these to the base MQeTraceRenderer, and use them in conjunction with the existing trace handlers. For an example of this, please refer to the MQeTrace class section of the WebSphere MQ Everyplace Java Programming Reference.

Trace points generated from WebSphere MQ Everyplace

All of the WebSphere MQ Everyplace trace points use negative trace point numbers. They are provided to facilitate problem diagnosis for the IBM Service team, when investigating a reported problem on the Websphere MQ Everyplace product.

Each trace point may change its meaning, value, and order in respect of other trace points between versions of the WebSphere MQ Everyplace classes. A trace point used in one version of WebSphere MQ Everyplace might never be issued in another. For this reason, we strongly recommend that solutions do not use a trace point as a trigger for application logic.

When rendering trace point information to a readable format, maintain a consistent version between all of the WebSphere MQ Everyplace classes. Failure to do so might result in misleading information being written to the trace output.

Trace information in the C codebase

This section describes trace and logging in the C codebase. It shows you how to enable WebSphere MQ Everyplace trace on PocketPC and PocketPC2002 devices and emulators.

Information on trace and logging is organised under the following headings:

- Trace architecture
- Configuring trace

Trace architecture

In WebSphere MQ Everyplace, trace is configured globally. This means that trace is enabled for the device, and when enabled, all C WebSphere MQ Everyplace applications generate trace.

You can configure the location where you want trace files to be written. Each application generates a unique file in the form MQEnnnnn.trc, where ″nnnnn″ is the process identifier of the application. WebSphere MQ Everyplace trace files are written in a binary format to minimise their size. You can either send these binary trace files directly to an IBM Service Representative to decode them or, alternatively, use the MQenativeTraceFormatter.exe utility provided with WebSphere MQ Everyplace. This utility runs on Windows, but does not run on PocketPC. It takes the trace file as an argument and prints the decoded output to standard out. The output can be captured in a file by running a command, such as:

```
MQenativeTraceFormatter.exe AMQ12345.trc > AMQ12345.txt
```

This will decode the file AMQ12345.trc and place the output in a file called AMQ12345.txt.

Configuring trace in the C codebase

Trace is controlled on the PocketPC via entries in the ″Windows Registry″. These trace values are under the HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQe\CurrentVersion\Trace key. You can set the values in a number of ways:

- Manually, using a registry editor, such as the Remote Registry Editor provided with eMbedded Visual Tools V3.0
- With a .Reg file, which you can download to the device and then execute
- Programatically, using the supplied mqeTrace_setOptions function

For information on the mqeTrace_setOptions function, see the C Programming Reference on the product CD. If you set the value manually or use a .Reg file, all values should be of type REG_SZ. WebSphere MQ Everyplace supports the following values:

Table 1. Trace values supported in WebSphere MQ Everyplace

Value Name	**Supported values**	**Description**
Enable	Yes or no	Turns trace on and off.
Location	Full path	Directory where trace files are written to. The location string must be a valid file path, for example mqetrace.
Timestamp	Yes or no	Determine if timestamp information is added to each tracepoint. Set to ″no″ to reduce file size and increase speed.
Parameters	Yes or no	Determine if parameter information is added to each tracepoint. Set to ″no″ to reduce file size and increase speed.
WrapLength	Value	Advanced value, described in the following list under Wraplength.
SubtractMethodFilter	Value	Advanced value, described in the following list under AddMethodFilter and SubtractMethodFilter.
AddMethodFilter	Value	Advanced value, describted in the following list under AddMethodFilter and SubtractMethodFilter.

In Table 1, the following conditions apply:

WrapLength

This is the maximum size, in bytes, that an individual trace file will reach. Once this value is reached, the trace file begins to wrap using a ″circular buffer″ algorithm. However, this takes a considerable amount of time, and may significantly slow down execution speed once the file starts wrapping. Therefore, leave this value at -1, except in circumstances where disk space is at a premium.

Note: This is the maximum length of a single trace file. If an application is run multiple times, or multiple applications are run, then each generated trace file reaches this size.

AddMethodFilter and SubtractMethodFilter

These values allow sophisticated control over exactly what trace points are produced. Incorrect use can seriously limit the effectiveness and understandability of the trace files. You should leave these fields blank, unless an IBM service representative instructs you otherwise. If you do send trace files to IBM, you must include details of what both of these fields are set to.

Chapter 7. WebSphere MQ Everyplace Diagnostic tool

WebSphere MQ Everyplace includes a small diagnostic tool that can be used to gather the information required by technical support personnel to assist with problem determination. The tool collects information about the local WebSphere MQ Everyplace environment. In particular:

- CLASSPATH and PATH information
- Java and C system variables
- Version information of the WebSphere MQ Everyplace classes

No personal information or WebSphere MQ Everyplace message data is collected by this program, and it should normally only be used at the request of IBM technical support personnel.

This tool should not be confused with the trace facility, which is used to gather debugging information on a running WebSphere MQ Everyplace system.

Invoking the MQeDiagnostics Tool

If you need to use this tool it can be invoked as follows.

On Windows NT and Windows 2000

1. From a command prompt change to the `...\mqe\Java\demo\Windows\ folder`.
2. Edit the `MQeDiagnostics.bat` file to suit your environment. The file makes use of the `JavaEnv.bat` script, so either ensure that `JavaEnv.bat` correctly sets up your *`CLASSPATH`* and *`PATH`* environment variables, or configure them directly from within the `MQeDiagnostics.bat` script.
3. Run the `MQeDiagnostics.bat` file and follow the on screen prompts.
4. Once the tool has completed, look through the `MQeDiagnostics.out` file for any errors. Common errors include:

   ″.\MQeDiagnostics.properties could not be found″
   The tool requires the `MQeDiagnostics.properties` file to be supplied as input. Edit `MQeDiagnostics.bat` so that it points to the correct location for this file and rerun the tool.

   ″com.ibm.mqe.support.MQeDiagnostics is not recognized as an internal or external command...″
   JavaEnv.bat is not configured correctly. Edit `MQeDiagnostics.bat` and JavaEnv.bat if necessary and rerun the tool.

   ″java.lang.NoclassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″
   Edit `JavaEnv.bat` and `MQeDiagnostics.bat` if necessary so that the `...\MQe\Java\Jars\MQeDiagnostics.jar` can be found in the *`CLASSPATH`* environment variable.

   Note: Not all WebSphere MQ Everyplace classes can supply version information, so the MQeDiagnostics.out file may include some ″Unknown version!″ messages.

   5) Send MQeDiagnostics.out to the WebSphere MQ Everyplace support personnel.

On UNIX systems

1. From a command prompt change to the C folder name or the ...\mqe\Java\demo\UNIX\ folder.
2. Edit the MQeDiagnostics script to suit your environment. The file makes use of the JavaEnv script, so either ensure that JavaEnv correctly sets up your *`CLASSPATH`* and *`PATH`* environment variables, or configure them directly from within the MQeDiagnostics script.
3. Run the MQeDiagnostics script and follow the on screen prompts.
4. Once the tool has completed, look through the `MQeDiagnostics.out` file for any errors. Common errors include:

   ″.\MQeDiagnostics.properties could not be found″
   The tool requires the `MQeDiagnostics.properties` file to be supplied as input. Edit `MQeDiagnostics.bat` so that it points to the correct location of this file and rerun the tool.

   ″com.ibm.mqe.support.MQeDiagnostics : command not found″
   JavaEnv is not configured correctly. Edit MQeDiagnostics and JavaEnv if necessary and rerun the tool.

   ″java.lang.NoClassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″
   Edit JavaEnv and MQeDiagnostics if necessary so that the `...\MQe\Java\Jars\MQeDiagnostics.jar` file can be found in the *`CLASSPATH`* environment variable.

   Note: Not all WebSphere MQ Everyplace classes can supply version information, so the MQeDiagnostics.out file may include some ″Unknown version!″ messages.
5. Send `MQeDiagnostics.out` to the WebSphere MQ Everyplace support personnel.

Other systems

On other systems, the MQeDiagnostics tool should be invoked directly.

1. Add the `MQeDiagnostics.jar` file to your classpath.
2. Invoke the `com.ibm.mqe.support.MQeDiagnostics` class from the Java runtime environment. For example:

   ```
   C example
   ```

   ```
   java com.ibm.mqe.support.MQeDiagnostics
   MQeDiagnostics.properties > MQeDiagnostics.out
   ```

   The program takes the `MQeDiagnostics.properties` file as an argument.
3. Send the output from the tool to the WebSphere MQ Everyplace support personnel.

Appendix. 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.

IBM
MQSeries

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

Windows NT is a registered trademark of Microsoft Corporation in the United States andin the United States and/or other countries other countries.

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

Bibliography

Related publications:

- *WebSphere MQ Everyplace Read Me First*, GC34-6276-00
- *WebSphere MQ Everyplace Introduction*, SC34-6277-00
- *WebSphere MQ Everyplace Application Programming Guide*, SC34-6278-00
- *WebSphere MQ Everyplace C Bindings Programming Guide*, SC34-6280-00
- *WebSphere MQ Everyplace Java Programming Reference*
- *WebSphere MQ Everyplace C Programming Reference*
- *WebSphere MQ Everyplace C Programming Guide for PalmOS*, SC34-6281-00
- *WebSphere MQ Everyplace Configuration Guide*, SC34-6283-00
- *WebSphere MQ An Introduction to Messaging and Queuing*, GC33-0805-01
- *WebSphere MQ for Windows NT V5R1 Quick Beginnings*, GC34-5389-00

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.

Printed in U.S.A.

SC34-6274-00