WebSphere MQ Everyplace V2.0.2

Contents

Developing a basic application

This topic contains the information that you need for creating a simple MQe application. It introduces the MQe development toolkit and explains what you need to do to set up your development environment. The walkthrough then gives step-by-step instructions on how to create a simple MQe application, and verify that it is working.

A simple example application called HelloWorld is also described. This simple application demonstrates how to use some of the features of MQe. Finally, the topic introduces some of the tools that you can use to develop and administer MQe applications.

Introduction to the WebSphere MQe development kit

This topic introduces the WebSphere® MQe Development Kit, which is a development environment for writing messaging and queueing applications based on Java and C. For information on the availability of development kits for environments other than Java™ and C, see the WebSphere MQ web site at:

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

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

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

Examples
: Java source code and classes that demonstrate how to use many features of MQe.

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

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

Examples
: C source code that demonstrates how to use the many features of MQe.

Setting up your development environment

This topic provides information on setting up your development environment for Java and C.

Java development

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

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

  Windows®

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

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

  UNIX®

  In a UNIX environment you can use the following:

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

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

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

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

Note: Customized versions of JavaEnv.bat or JavaEnv may be overwritten if you reinstall MQe.

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

Possible values are:

Sun - Sun

JB - Borland JBuilder

MS - Microsoft®

IBM® - IBM

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

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

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

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

J2ME environment

There are two distinct J2ME environments:

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

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

Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP)
: Applications developed for MIDP can also be compiled using a normal J2SE JVM (again using the bootclasspath to point to the required Midp class library), but they normally have to be run within a Midp Emulator. Therefore, we recommend developing the application using one of the MIDP Toolkits available on the Web. MQe provides a MIDP jar that should be used within this environment. The `MQeMidp.jar` is in the <MQeInstallDir>\Java\Jars directory.

C development

To develop programs in C using the MQe Development Kit, you need the following tools:

Microsoft eMbedded Visual C++ (EVC) Version 3.0.
This is included in Microsoft eMbedded Visual Tools 3.0, which is available as a free download from the Microsoft web page:

`http://msdn.microsoft.com/mobile/`

You must use version 3.0 as version 4.0 does not support PocketPC.

An SDK for your chosen platform
Microsoft eMbedded Visual Tools 3.0 includes an SDK for PocketPC 2000. You can also download an SDK for PocketPC 2002 from Microsoft:

`http://msdn.microsoft.com/mobile/`

C Bindings

For the C Bindings code base, see the C Bindings Programming Reference.

Native C

For general information see C Programming Reference, in particular the page *Compilation Information*. However, that page is now slightly out of date and this topic provides an update.

For the native C code base, support is provided for four platforms:

- PocketPC2000
- PocketPC2002
- PocketPC2003
- Windows 32bit.

For PocketPC, binaries are provided for both the device and the emulator that is available in the Integrated Development Environment Microsoft eMbedded Visual C++. The binaries provided for the devices are compiled for ARM processors.

Binary files

The root of the binary files, as well as the documentation and examples, is the C directory below the directory where you choose to install MQe.

Then in the C directory, the files are located as follows:

PocketPC2000

ARM

DLLs C\PocketPc2000\arm\bin

LIBs C\PocketPc2000\arm\lib

Emulator

DLLs C\PocketPc2000\x86emulator\bin

LIBs C\PocketPc2000\x86emulator\lib

PocketPC2002

ARM

DLLs C\PocketPc2002\arm\bin

LIBs C\PocketPc2002\arm\lib

Emulator

DLLs C\PocketPc2002\x86emulator\bin

LIBs C\PocketPc2002\x86emulator\lib

PocketPC2003

ARM

DLLs C\PocketPc2003\arm\bin

LIBs C\PocketPc2003\arm\lib

Emulator

DLLs C\PocketPc2003\x86emulator\bin

LIBs C\PocketPc2003\x86emulator\lib

Windows 32bit

DLLs C\Win32\Native\bin

LIBs C\Win32\Native\lib

Header files

The header files are common to all the Native platforms, and can be found in the include directory below the installation directory.

MQe_API.h

This is the ″root″ header file. If this is included all relevant header files included for you.

In order to ensure the correct files and definitions are included you must indicate that you are running the Native code base as follows:

```
#define   NATIVE                  // or specify this as an option to the compiler
#include  <published/MQe_API.h>
```

Linking

You need to link against the following two libraries:

HMQ_nativeAPI.lib
// the API library

HMQ_nativeCnst.lib
// the static constant MQeString library

You need to include both these files. Then an optimizing linker removes links to any functions and constants that you have not used.

The other MQe libraries are statically and dynamically linked with the main API library and are included as required.

Using embedded Visual C++

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

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

- Win32 (WCE ARM) Debug
- Win32 (WCE ARM) Release

Note: Some of the Target Processor or **Target Operating System** options may not be available, depending on which SDKs you have installed.

- Include the header files for the native C code base. These are shared between the two versions of PocketPC and by the C Bindings. The header file location is in the installation directory under `include`. If you include the root header file, MQe_API.h, you include all the functions that you may require. As header files are shared, you need to define which version of the code base you are using, as shown in the following example:

```
#define NATIVE
#define MQE_PLATFORM  PLATFORM_WINCE

/*Alternatively, you should add this to the Preprocessor Definitions
in the Project Settings Dialog.  Add the following to the start
of the list*/
NATIVE,MQE_PLATFORM=PLATFORM_WINCE

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

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

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

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

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

Threading

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

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

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

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

Calling conventions

The calling convention for all of the APIs has been explicitly set at `_cdecl`. However, you can use a different default calling convention in your application.

Handles and items

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

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

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

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

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

MQe memory functions

MQe provides the following functions for memory management:

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

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

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

MQeString

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

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

Constant Strings

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

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

Constructor

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

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

Destructor

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

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

Getter

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

This function populates a character buffer with the contents of an MQeString performing conversion wherever necessary. Only simple conversions are carried out. No code page conversion is attempted. For example, if an SBCS string has been put into the string, then trying to get the data out as DBCS (Unicode) data works correctly. If the data was put in as DBCS however, and you try to get the data out as SBCS, this only works if the data does not have any

values that cannot be represented with a single byte. When get() is used for SBCS, DBCS, or QBCS, each character is represented by its Unicode code point value.

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

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

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

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

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

This function duplicates an MQeString.

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

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

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

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

Tester

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

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

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

This function determines whether two strings are equivalent.

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

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

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

Walkthrough: creating a basic application

This topic contains step-by-step instructions for creating a simple MQe application. It describes the steps you need to perform to create and configure your first queue manager, and then to verify that it can send and receive messages from another queue manager.

As well as describing what you need to do, it also tells you which MQe_Script commands you can use to perform each task simply. MQe_Script uses defaults for many attributes, which you would otherwise have to specify if you were writing equivalent code.

MQe_Script is available as part of the Server Support SupportPac™ from the IBM Web site. This SupportPac includes full documentation on the use of all MQe_Script commands, including details of the defaults and explanations of how to change them if necessary.

You can also perform many of the steps involved in this process using the MQe_Explorer, which is included in the same SupportPac.

Finally, the walkthrough provides links to pieces of example code that show you how to perform many of the steps programmatically.

Once a queue manager has been created and started, all of the configuration (including the creation of queues, connection definitions, remote queue definitions, and listeners) is performed using administration messages.

1. Create a queue manager (QM1)

When you create a queue manager, you need to define the following attributes:

- Queue manager name
- Public or private registry
- Registry location
- Message store adapter
- Default queues
  - AdminQ
  - AdminReplyQ
  - DeadLetterQ

– System.default.local.Q

You can also set other (optional) attributes at this time, including a description, channel timeout, channel attribute rule name, and queue manager rule, but these are not included in this walkthrough.

Creating QM1 using MQe_Script:

You can use the following MQe_Script command to create a queue manager called QM1:

```
mqe_script_qm -create -qmname QM1
```

This command creates a queue manager called QM1, with the following characteristics:

- Public registry
- A base location of C:\program files\mqe\java\mqe_script. The default registry and queue directories are in subdirectories in this path
- Uses the default message store and saves the information to disk
- Contains 4 default queues

An ini file is also created so that the queue manager information is saved and can be started again by passing the location of this file to an appropriate method.

2. Start the queue manager (QM1)

When you have created the queue manager called QM1, you need to start it.

Starting QM1 using MQe_Script:

You can use the following MQe_Script command to start the queue manager called QM1:

```
mqe_script_qm -load
```

When no name is supplied, this command starts the queue manager that has just been created. If you want to know how to load a queue manager and specify the INI file, see the documentation supplied with MQe_Script.

3. Create a local queue (Q1)

When you have started the QM1 queue manager, you can create a local queue called Q1:

Creating Q1 using MQe_Script:

You can use the following MQe_Script command to create a local queue called Q1:

```
mqe_script_appq -create -qname Q1
```

This command creates a basic local queue (also know as *application queues*) called Q1, on the QM1 queue manager.

4. Create a connection definition

When you have created your local queue (Q1), you need to create a connection definition, specifying the following:

- The name of the queue manager that you want to connect to (the remote queue manager)
- The port on which the remote queue manager will be listening
- The communications adapter.

Creating a connection definition using MQe_Script:

You can use the following MQe_Script command to create a connection definition:

```
mqe_script_condef -create -cdname QM2 -port 1881
```

This command creates a connection definition to a queue manager called QM2, which is listening on port 1881. It is not necessary for QM2 to exist when the connection is created, but it must exist when you try to send a message to a remote queue on that queue manager. As no adapter is specified, the Http adapter is used by default.

5. Create a remote queue definition

When you have created a connection definition, you need to create a remote definition of a local queue on queue manager QM2.

Creating a remote queue definition using MQe_Script:

You can use the following MQe_Script command to create a remote queue definition:

```
mqe_script_sproxyq -create -qname Q2 -destination QM2
```

This command creates a synchronous proxy queue, which is a remote definition of a local queue on QM2. It is not necessary for QM2 to exist when the remote queue definition is created. However, you must create a connection definition (see "4. Create a connection definition" on page 10) before you can create this remote queue definition.

6. Create a listener (L1)

When you have created a remote queue definition, you need to create a listener.

Creating a listener using MQe_Script:

You can use the following MQe_Script command to create a listener called L1 (on queue manager QM1):

```
mqe_script_listen -create -listenname L1 -port 1882
```

Creates a listener for queue manager QM1 and listens on port 1882. The default communications adapter is used, which is the Http adapter.

7. Start listener (L1)

When you have created a listener, you need to start it.

Starting a listener using MQe_Script:

You can use the following MQe_Script command to start the listener L1:

```
mqe_script_listen -start -listenname L1
```

8. Create a second queue manager (QM2)

When you have finished configuring QM1 (as shown in the previous steps in this walkthrough), you need to create a second queue manager called QM2:

Creating QM2 using MQe_Script:

You can use the following MQe_Script command to create a queue manager called QM2:

```
mqe_script_qm -create -qmname QM2
```

This command creates a queue manager called QM2, with the following characteristics:

- Public registry
- A base location of C:\program files\mqe\java\mqe_script. The default registry and queue directories are in subdirectories in this path
- Uses the default message store and saves the information to disk
- Contains 4 default queues

An ini file is also created so that the queue manager information is saved and can be started again by passing the location of this file to an appropriate method.

9. Start QM2

When you have created the queue manager called QM2, you need to start it.

Starting QM2 using MQe_Script:

You can use the following MQe_Script command to start the queue manager called QM2:

```
mqe_script_qm -load
```

When no name is supplied, this command starts the queue manager that has just been created. If you want to know how to load a queue manager and specify the INI file, see the documentation supplied with MQe_Script.

10. Create a local queue (on QM2) called Q2

When you have started the QM2 queue manager, you can create a local queue called Q2.

Creating Q2 using MQe_Script:

You can use the following MQe_Script command to create a local queue called Q2:

```
mqe_script_appq -create -qname Q2
```

This command creates a basic local queue called Q2, on the QM2 queue manager.

11. Create a connection definition (on QM2)

When you have created your local queue (Q2), you need to create a connection definition, specifying the following:

- The name of the queue manager that you want to connect to (the remote queue manager)
- The port on which the remote queue manager will be listening
- The communications adapter.

Creating a connection definition using MQe_Script:

You can use the following MQe_Script command to create a connection definition:

```
mqe_script_condef -create -cdname QM1 -port 1882
```

This command creates a connection definition to a queue manager called QM1, which is listening on port 1882. It is not necessary for QM1 to exist when the connection is created, but it must exist when you try to send a message to a remote queue on that queue manager. As no adapter is specified, the Http adapter is used by default.

12. Create a remote queue definition (on QM2)

When you have created a connection definition, you need to create a remote definition of a local queue on queue manager QM1.

Creating a remote queue definition using MQe_Script:

You can use the following MQe_Script command to create a remote queue definition:

```
mqe_script_sproxyq -create -qname Q1 -destination QM1
```

This command creates a synchronous proxy queue, which is a remote definition of a local queue on QM1. It is not necessary for QM1 to exist when the remote queue definition is created, but it must exist before a message is put to it.

13. Create a listener (on QM2) called L2

When you have created a remote queue definition, you need to create a listener.

Creating a listener using MQe_Script:

You can use the following MQe_Script command to create a listener called L2 (on queue manager QM2):

```
mqe_script_listen -create -listenname L2 -port 1881
```

Creates a listener for queue manager QM2 and listens on port 1881. The default communications adapter is used, which is the Http adapter.

14. Start the listener L2 (on QM2)

When you have created a listener, you need to start it.

Starting a listener using MQe_Script:

You can use the following MQe_Script command to start the listener L2:

```
mqe_script_listen -start -listenname L2
```

15. Send (PUT) a message from QM1 to QM2

Now that you have created and started the two queue managers, created your queues and connection definitions, and created and started your listeners, you are in a position to send messages between the two queue managers.

Sending a message using MQe_Script:

On QM1, you can use the following MQe_Script command to send a message from QM1 to QM2:

```
mqe_script_msg -put -qname Q2 -qmname QM2
```

This command puts a message to queue Q2 on queue manager QM2.

16. Receive (GET) the message on QM2

Now that a message has been put to the queue from QM1, you can get the message from QM2.

Receiving a message using MQe_Script:

You can use the following MQe_Script command to get the message from the queue:

```
mqe_script_msg -get -qname Q2 -qmname QM2
```

17. Displaying details of MQe objects

You can display details of the MQe objects that you have created by issuing the `inquireall` MQe_Script command. For example, to see information about the local queue manager, use the following command:

```
mqe_script_qm -inquireall
```

This displays all the information about the local queue manager, and shows you any defaults that MQe_Script has used.

You can also display information about other objects, by specifying the object name. For example:

```
mqe_script_condef -inquireall -cdname QM2
```

An example MQe application (HelloWorld)

This topic describes how to create a basic application (called HelloWorld) using the MQe Java and C APIs. It contains information on designing, developing, deploying, and running the application.

Java ″HelloWorld″

This section describes how to design, develop, deploy, and run a basic ″HelloWorld″ application in Java.

Designing the Java application

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

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

Typically, configuring new entities is a separate process from their actual use. Once configured, administering these entities also requires a different process than using them. This section concentrates on usage rather than administration.

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

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

Developing the Java application

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

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

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

Overview of examples.helloworld.run:

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

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

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

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

Start the queue manager:

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

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

```
public void start() throws Exception {

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

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

         // Create all the configuration
```

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

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

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

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

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

          config.putFields("Registry", registrySection);

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

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

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

Create a message and put to a local queue:

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

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

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

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

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

```
public void get() throws Exception {
        System.out.println("Getting the test message.");
        MQeMsgObject msg = myQueueManager.getMessage( queueManagerName,
                                                      MQe.System_Default_Queue_Name,
                                                      null, null, 0L );

        if (msg != null) {
            System.out.println("Got the test message.");

            if (msg.contains("myFieldName")) {
```

```
                String textGot = msg.getUnicode("myFieldName");

                System.out.println("Message contained the text '" + textGot + "'");
            }
        }
    }
```

Stopping and deleting the queue manager:

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

Stopping the queue manager

You can stop the queue manager using a controlled shutdown.

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

Deleting the definition of the queue manager from the disk

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

Running the Java application

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

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

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

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

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

C ″HelloWorld″

This section describes how to design, develop, deploy and run a ″HelloWorld″ application in C.

Designing the C application

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

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

Typically, configuring new entities is a separate process from their actual use. Once configured, administering these entities also requires a different process than using them. This section concentrates on usage rather than administration.

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

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

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

Developing the C application

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

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

Overview of HelloWorld_Runtime.c:

You need to include just one header file to access the APIs. You must include the NATIVE definition to indicate that this is not the CBindings. You must also define the MQE_PLATFORM upon which you intend to run the application.

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

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

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

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

```
MQeStringHndl     hLocalQMName;

       ...

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

The first API call made is session initialize:

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

Start the queue manager:

This process involves two steps:

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

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

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

```
     if (MQERETURN_OK == rc) {

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

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

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

```
     /* Start the queue manager  */

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

Create a message:

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

```
  MQeFieldsHndl hMsg;

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

Put message to a local queue:

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

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

```
                                        0);

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

Get message from a local queue:

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

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

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

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

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

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

    }
```

Shutdown:

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

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

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


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

Compiling:

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

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

Deploying the C application

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

The following instructions are applicable to both the emulator and an actual device:

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

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

Running the C application

This section describes how to run the ″HelloWorld″ application in Java and C, on the PocketPC or emulator.

This example involves two steps:

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

Using the MQe development and administration tools

The following are some of the tools that you can use to develop or administer MQe applications:

MQe_Explorer
The MQe_Explorer provides a graphical user interface for the management of an MQe network and its interconnection with MQ. It allows MQe queue managers and their associated objects, such as queues, connections, and bridges, to be locally or remotely configured. MQe_Explorer also provides a simple way of creating local queue managers, which can then be further configured to meet the needs of applications. It also offers a launch and debug environment for MQe applications. MQe_Explorer is available as part of the Server Support SupportPac.

MQe_Script
MQe_Script is a command-line based tool for MQe, and is platform independent. It allows MQe queue managers and their associated objects, such as queues, connections, listeners, and bridge objects to be locally or remotely configured. Test messages can also be sent to the queues to validate the operation of the network. Like the MQe_Explorer, MQe_Script provides a simple way of creating local queue managers, which you can then configure and extend for use by your application. MQe_Script is available as part of the Server Support SupportPac.

MQe_Service
MQe_Service is a wizard-based tool for MQe local queue manager creation and operation.

Additionally, it enables the automated set up of MQe gateway and MQ queue managers, where messages are required to pass between MQe and MQ networks.

Rational® Software Development Platform

Eclipse

Eclipse is an open industry-supported platform for software development tools. It provides a plug-in based framework that facilitates the creation, integration, and use of software tools. For more information on Eclipse, see:

`http://www.eclipse.org`

Index