MQSeries Everyplace for Multiplatforms

Native Client Information

Version 1.2.7

SC34-6187-00

MQSeries Everyplace for Multiplatforms

Native Client Information

Version 1.2.7

SC34-6187-00

Take Note!

Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices" on page 177

Licence warning

MQSeries Everyplace for Multiplatforms Version 1.2.7 is a toolkit that enables users to write MQSeries Everyplace applications and to create an environment in which to run them.

The licence conditions under which the toolkit is purchased determine the environment in which it can be used:

*If MQSeries Everyplace for Multiplatforms is purchased for use as a **device** (client) it may **not** be used to create an **MQSeries Everyplace channel manager**, or an **MQSeries Everyplace channel listener.**, or an **MQSeries Everyplace bridge***

*The presence of an **MQSeries Everyplace channel manager**, or an **MQSeries Everyplace channel listener**, or an **MQSeries Everyplace bridge** defines a **gateway** (server) environment, which requires a gateway licence.*

Fourth Edition (March 2002)

This edition applies to MQSeries Everyplace for Multiplatforms Version 1.2.7 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 MQSeries family library Web page at `http://www.ibm.com/software/mqseries/library/`.

Contents

About this book

This book is a programming guide for the MQSeries Everyplace for Multiplatforms product (generally referred to in this book as MQSeries Everyplace). It contains information on how to use the MQSeries Everyplace C APIs, and guidance is provided for writing C programs to perform common messaging tasks. In many cases example code is supplied. The C versions of the MQSeries Everyplace APIs are described in detail.

The book is divided into three parts:

- Part 1, "Device information" on page 1 – Getting started information for specific devices
- Part 2, "General programming guidance" on page 17 – General programming guidance for the native client
- Part 3, "Programming reference" on page 43 – Reference information for the native client

This book is intended to be used in conjunction with

- *MQSeries Everyplace for Multiplatforms, Introduction* – a detailed description of the capabilities of MQSeries Everyplace,
- *MQSeries Everyplace for Multiplatforms, Programming Guide* – guidance for programming with MQSeries Everyplace
- *MQSeries Everyplace for Multiplatforms, Programming Reference* – detailed descriptions of the Java® version of the MQSeries Everyplace API

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/`

This document is continually being updated with new and improved information. For the latest edition, please see the MQSeries family library Web page at the Web site indicated above.

Who should read this book

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

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

Prerequisite knowledge

THis documentation assumes that the reader has a working knowledge of C programming techniques, and an understanding of MQSeries Everyplace as described in *MQSeries Everyplace for Multiplatforms, Introduction*.

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 book:

- *WebSphere MQ An Introduction to Messaging and Queuing*

This book is 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/`

Summary of changes

This section describes changes in this edition of *MQSeries Everyplace for Multiplatforms, Native Client Information*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

Changes for this edition (SC34-6187–00)

This edition contains corrections and clarifications to the first edition, including:

- Updates for CodeWarrior, versions 7 and 8
- Instructions using the Palm 0S 3.5 and MQSeries Everyplace on Windows 2000

changes

Introduction

The MQSeries Everyplace C application programming interface (API) is an WebSphere MQ messaging product designed for use on pervasive computing devices. This API enables a device to interchange messages with the MQSeries Everyplace network and with other members of the WebSphere MQ family, extending the reach of WebSphere MQ network to pervasive devices.

MQSeries Everyplace is optimized for hand-held devices that are resource constrained, for example with small memory or low power. MQSeries Everyplace has a small footprint, (on the Palm it's less than 88 KB). The design of MQSeries Everyplace follows the principles of software programming for these devices, as recommended by the device operating system manufacturers. The C programming interface supports this programming model by offering APIs that can be called multiple times to move a block of data between the application and the MQSeries Everyplace system.

MQSeries Everyplace on pervasive devices interoperates with an MQSeries Everyplace Java server. It uses the Web standard HTTP 1.0 protocol to communicate with the server. The use of this protocol enables MQSeries Everyplace messages to pass through standard firewalls without any need to modify the firewalls.

This programming information includes:

- A brief description of the software components that make up the MQSeries Everyplace for individual devices and guidance on setting up the devices to use MQSeries Everyplace.
- Guidance on writing programs to perform common messaging tasks
- Detailed descriptions of the native client API and other reference material

Installation

The MQSeries Everyplace Native Client Version 1.0 files must be installed on a Microsoft® Windows NT® PC or laptop. This is the environment in which MQSeries Everyplace applications are written. To enable a specific device to run MQSeries Everyplace applications, some of the client files and the applications are downloaded to the device. Part 1, "Device information" on page 1 provides information on the download procedures for each supported device type.

Part 1. Device information

Only PalmOS pervasive devices are supported by the MQSeries Everyplace Version 1.2.7 native client.

device information

Chapter 1. Getting started with Palm

This section explains how to set up and run a basic MQSeries Everyplace program from a Palm OS device, such as Palm V or IBM Workpad C3 to an MQSeries Everyplace Java server.

Prerequisites

This information assumes the following environment:

- A Palm OS device with Palm OS Version 2.0 or later
- A cradle for the device including a serial connection to a PC or laptop
- A Microsoft Windows NT/2000 PC or laptop
- Palm Desktop (in particular the HotSync Manager and the Install Tool - used for HotSyncing) installed on the PC.
- Metrowerks CodeWarrior for Palm Computing Release 5, or later installed on the PC (check `http://www.palm.com/devzone/tools/cw/` for updates and patches)
- Access to either an existing MQSeries Everyplace server (queue manager and queue names as well as IP address, port, and channel commands) or the MQSeries Everyplace Java server code

You also need to install the following programs from the web:

- WebSphere Studio Device Developer at: *http://www.embedded.oti.com*
- Palm OS Emulator at: *http://www.palmos.com*
- Palm Desktop 4.0 at: *http://www.palm.com*
- PilRC (a compiler for the Palm)at: *http://www.handango.com*
- Cygwin (required by PilRC) at: *http://www.cygwin.com*
- JDK 1.4 at: *http://www.java.sun.com*
- J2ME Wireless Toolkit at: *http://www.java.sun.com*

Overview

The following sections of this documentation explain how to:

1. Create and compile a Palm program that utilizes MQSeries Everyplace using *Metrowerks CodeWarrior*.
2. Use the SupportPac EAP1 examples project file and code.
3. HotSync the various files needed to run the program onto the Palm OS device.
4. Install, configure, and start Windows® RAS (Remote Access Service) on your PC.
5. Configure networking and MQSeries Everyplace on the Palm OS device.
6. Create an MQSeries Everyplace queue manager and start an MQSeries Everyplace server on the PC.
7. Run the palm program on the PalmOS device to connect to the MQSeries Everyplace server.

Creating and compiling a basic Palm program that uses WebSphere MQ messaging

Note: SupportPac EAP1 includes an example application and its source code. You can use this program instead of a user written program. Please refer to Using the SupportPac EAP1 example project file and code.

To create your own program, use the following procedure:

1. Start *Metrowerks CodeWarrior*
2. Create a new project using the **File** menu.
3. Select one of the following, depending on your system:
   a. **CodeWarrior Version 5**
      1) From the dialog box prompting you to "Select Project Stationary", expand the **+** next to "Palm OS" and click **Palm OS C App**.
      2) Click **OK**.
      3) In the dialog box that is displayed, select an appropriate directory for the project folder and give the project a name, such as "BasicApp".
      4) Click **OK**.
   b. **CodeWarrior Version 6**
      1) From the tabbed dialog box, click **Palm OS 3.1 (English) Stationary**.
      2) In the text field on the right, give the project a name, such as "BasicApp" and set the location text field to the directory in which you want to store your project.
      3) Click **OK**.
      4) In the dialog box that is displayed, click **Palm OS C App**.
      5) Click **OK**.
   c. **CodeWarrior Version 7**
      1) From the tabbed dialog box, click **Palm OS 3.5 Stationary**.
      2) In the text field on the right, give the project a name, such as ″BasicApp″ and and set the location text field to the directory in which you want to store your project.
      3) Click **OK**.
      4) In the dialog box that is displayed, click **Palm OS C App**.
      5) Click **OK**.
   d. **CodeWarrior Version 8**

      Note: You cannot use the CodeWarrior Version 8 Demo to build an MQSeries Everyplace because you will have link errors.

      1) From the project tab in the **New**dialog box, click **Palm OS Application Stationary**.
      2) In the text field on the right, give the project a name, such as ″BasicApp″ and set the location text field to the directory in which you want to store your project.
      3) Click **OK**.
      4) In the dialog box that is displayed, click **Palm OS C App**.
      5) Click **OK**.

This creates a new folder and a set of source files within that folder. The folder has the same name as the project, and the project file within the folder has the extension name, .mcp, for example ″BasicApp.mcp″.

4. In CodeWarrior, a project window called ″ExtensionName.mcp (″BasicApp″ for example) opens. Expand the **Source** and **Resource** folders to display the **Starter.c** and **Starter.rsrc** files.

   Double-click the **Starter.c** file to open an edit window.

5. Edit the **Starter.c** file as shown in the following example (adding the text in italics).

```
#include <Pilot.h>
#include <SysEvtMgr.h>
#include "StarterRsc.h"
#include <hmq.h>          /* <- MQe header file */
static Err AppStart(void)
{
    StarterPreferenceType prefs;
    Word prefsSize;

    /******* MQe defines ************/
    MQEHSESS   hSess;
    MQEHFIELDS hMsg;
    MQEINT32   compcode;
    MQEINT32   reason;
    MQEPMO     pmo = MQEPMO_DEFAULT; /* Set default put message options */

    /******* End of MQe defines *****/

    // Read the saved preferences / saved-state information.
    prefsSize = sizeof(StarterPreferenceType);
    if (PrefGetAppPreferences(appFileCreator, appPrefID, , , true) != noPreferenceFound)
        {
        }

    /*************** MQe code added ***************/

    /* Initialize the session: connect to the local queue manager   */
    hSess = MQeInitialize("MyAppsName", &compcode, &reason);

    /* Allocate memory for the MQeMsgObject (an MQeFields object with two set fields) */
    hMsg = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_MSGOBJECT, &compcode, &reason);

    /* If the allocation was successful put some data into the MQeFields object */
    if ( compcode == MQECC_OK ) {
        /* Put ASCII text "Hello World" into the MQeMsgObject */
        /*in a field named "HelloAscii" */
        MQeFieldsPut(hSess, hMsg, "HelloAscii", MQE_TYPE_ASCII,
                    "Hello World", StrLen("Hello World"), &compcode, &reason );
    }

    /* Now put the message to a Queue Manager and Queue */
    MQeQMgrPutMsg( hSess, "aQMgrName", "aQueueName", &pmo, hMsg, &compcode,  &reason);
    /* If the  initial allocation was successful, */
    /* free the memory held by the MQeMsgObject */
    if (hMsg!=MQEHANDLE_NULL) {
        MQeFieldsFree(hSess, hMsg, &compcode, &reason );
    }
    /* Terminate the session */
    MQeTerminate(hSess, &compcode, &reason);

    /*************** End of MQe code **************/
    return 0;
}
```

This code:

a. Starts a session to the local queue manager
b. Creates an MQeFields message object
c. Puts some data into the message object
d. Puts the message to a queue manager and queue with the names `aQMgrName` and `aQueueName`. Change these names to those of the queue manager and queue that you want to use. If the queue manager and queue are on a separate MQSeries Everyplace server, the queue manager name is `ExampleQM` and the queue name is `SYSTEM.DEFAULT.LOCAL.QUEUE`.
e. Frees the message object and terminates the session when the message put is complete.

When you have finished editing the Starter.c file, save it and close the edit window.

6. Link the MQSeries Everyplace stub library as follows:
   a. Select the project window
   b. Click **Project — > Add Files**. This opens the **Select files to add...** dialog.
   c. Change the **Files of type** filter to **Library files** and navigate to the **hmq.lib** file. If you kept the directory structure of this SupportPac after installation, **hmq.lib** is in the **MQeNativeClient/Palm OS Support** directory.
   d. Select **hmq.lib** and click **Add— > OK**. CodeWarrior should then automatically add a path entry that points to where the hmq.lib has been installed, that is the ″User Paths″ list in the ″Target″ group of ″Starter Settings″.

   A project message appears confirming that an access path has been added.
7. Ensure that the compiler knows where to find the MQSeries Everyplace header file hmq.h as follows:
   a. Do **one** of the following to display the **Starter Settings** dialog:
      - From the **Edit** menu, select **Starter-Debug Settings**.
      - In the project window, select **Starter-Debug**from the drop-down menu and click the**Starter-Debug Settings** icon.
   b. In the **Target** group on the left of the dialog box, click **Access Paths**.
   c. Click the **System Paths** radio button and click **Add**. This opens the **Browse for Folder** dialog box.
   d. Navigate to the folder where hmq.h is stored and click **OK** to add the path to hmq.h to the `System Path`.
   e. In the **Starter-Debug Settings** dialog, click **OK**.
8. Do one of the following to compile the program:
   - Click the **Make** icon.
   - Click **Project— > Make**.

   The project builds a compiled program called Starter.prc and stores it in the project folder.

Using the SupportPac EAP1 example project file and code

Note: This section is independent of the previous section, Creating and Compiling a basic Palm program that uses WebSphere MQ messaging. You cannot use the CodeWarrior Version 8 Demo to build an MQSeries Everyplace application because you will have link errors.

To use the example that comes with the SupportPac EAP1:

1. Double-click **MQeNativeClient\Palm\samples\MQeExample\MQeExample.mcp**
2. Click **OK** if you are asked whether to convert the project file.
3. If CodeWarrior cannot find ″Palm OS 3.0 Support″:
   a. Click the**Project (″MQeExample″) Settings** icon.
   b. Expand the **Target** group and click **Access Paths —> System Paths —> {Compiler}Palm OS 3.0 Support**.
   c. Click **Change**.
   d. From the **Browse for Folder** window, select the Palm OS Support that you have on your system. For example, select the Palm OS 3.1 Support directory for CodeWarrior Version 6, the ″Palm OS 3.5 Support″ directory for CodeWarrior Version 7, and the ″Palm OS Support″ directory for CodeWarrior Version 8. These directories are just below where you have installed CodeWarrior.
4. For CodeWarrior Versions 7 and 8, modify the MQeExample.c file:
   a. Click the **Files** tab in your project window.
   b. Click **AppSource**.
   c. Double-click **MQeExample.c**.
   d. Change `#include<Pilot.h>` to

      ```
      #include<PalmOS.h>
      #include<PalmCompatibility.h>
      ```
5. In the project window, click the**Project (″MQeExample″) Settings** icon. In the **Target** group, click the **User Paths** radio button.
6. Click **Change** and select the path to where you installed hmq.lib.
7. Click the **System Paths** radio button (also in the **Target**group. Enter paths to where hmq.h and hmq.Helper.h have been installed.
8. In the project window, click the **Debug** icon to rebuild the project.

HotSyncing the program and MQSeries Everyplace files onto the Palm OS device

This section describes the objects that need on the Palm OS device and the process for downloading them.

MQSeries Everyplace system components for the Palm device

MQSeries Everyplace for the Palm device consists of the following components:

1. **shared libraries `hmqLib.prc` and `hmqFields.prc`**

   These libraries support MQSeries Everyplace applications on the Palm.

   You must install both of these files on the device.

2. **GUI program `hmqIni.prc`**

   This GUI program hooks into the Palm preference panel. It enables you to manually configure the system parameters needed to run the MQSeries Everyplace system. You must install this program on the device.

3. **Stub library `hmq.lib`**

   You must link the small stub library, `hmq.lib` with an application program to use the MQSeries Everyplace system. The stub library consists of two object files, a stub for the shared libraries and object code for the helper functions. The helper functions are provided for programmers who want to use an

object-oriented style. However, using any of these helper functions increases the application code size by approximately 6KB.

4. **Include files `hmq.h` and `hmqHelper.h`**

   You must include the hmq.hinclude file in all applications. You only needhmqHelper.h if you use the helper functions.

HotSync Setup for Palm OS 3.5 and Windows 2000

You need to change the HotSync setup to use HotSync and MQSeries Everyplace simultaneously. To set up a local HotSync on Palm OS 3.5:

1. From the applications menu, tap the **HotSync** icon.
2. From the HotSync menu, select the following:
   - **Modem Sync Preferences — > Network**
   - **LANSync Preferences — > LANSync**
   - **Primary PC Setup** and enter your **Primary PC Name**, that is the nameyour PC uses on the network
   - **Finding your PC Name** and enter your **Primary PC Address**, that is your ip address
   - **Modem Setup:**

     Modem:
     IBM WorkPad c3 Modem

     Speed 57,000

     Speaker
     low

     Flow Ctl
     Automatic

     Country
     United Kingdom

     String AT&FX4

     Dialin TouchTone

To install the library files, see Installing the files on Palm OS

Installing the files on Palm OS

1. Start the Palm HotSync manager and the Palm Install Tool.
2. Click **Add** and navigate to the project folder.
3. Select the Starter.prc file and click **Open**. This adds the file to the list of files to install.

   If you want to use the example application (palmos-example), add the MQeExample.prc file from the palmos-example folder instead of Starter.prc.
4. Repeat steps 2–3 to add the hmqLib.prc, hmqFields.prc, and hmqIni.prc files (found within the MQSeries Everyplace folder) to the list of files to install.
5. Ensure that the serial cradle for the Palm device is plugged into the correct serial port on the PC.
6. Place the device in the cradle.
7. Click the **HotSync** icon on the PC.

   This installs the files on the PalmOS device.

Installing, configuring and starting Windows RAS (Remote Access Service) on the PC

In order find out whether you need to install RAS or not you need to perform the following tasks. Choose the appropriate platform for your PC.

Windows NT and Palm OS 3.1

On the PC:

1. Click **Start— > Settings— > Control Panel — > Network settings**
2. Click the **Services** tab and check to see if Remote Access Service (RAS) is already installed in the **Network Services** list.

   If RAS is not installed, it may be necessary to install a new modem type and then install RAS.

   If RAS is installed, check that the correct modem is installed
3. Create a new user on the system.

The procedures for these tasks are described in the following sections.

Install the modem

To install the modem, close the **Network settings** dialog and open the **Modems** dialog.

If a modem called "Dial-Up Networking Serial Cable between 2 PCs" is installed, no modem installation is required and you can go directly to the “Install RAS” section.

If this modem is not installed,

1. Click **Add** to start the **Install New Modem Wizard**.
2. Select the **Don’t Detect My Modem** check box and click **Next**.
3. Click the **(Standard Modem Types)** Manufacturer and the **Dial-Up Networking Serial Cable between 2 PCs** modem.
4. Click **Next**. A Port selection dialog is displayed.
5. Click the port that the Palm cradle is plugged into and click **Next** to install the modem.
6. Click **Finish** to close the wizard . The new modem is added to the list.
7. Click **Properties**. Set the Maximum Speed to 19200
8. Click **Close** to close the **Modems** dialog.

Install RAS

To install RAS:

1. Click **Control Panel— > Network**.
2. Click the **Services** tab and click **Add**.
3. In the dialog that is displayed, click **Remote Access Service— > OK**. A Windows dialog asks for the location of some Windows NT files. These are either in the i386 directory of the boot partition (C:\i386\), or on the Windows NT CD-ROM (also in the i386 directory). When these files are installed, the **Remote Access Setup** dialog and the **Add RAS Device** dialogs are displayed.
4. Click the **Dial-Up Networking Serial Cable Between 2 PC** modem in this dialog and click **OK**.
5. In the **Remote Access Setup** dialog, click **Configure** to display a **Configure Port Usage** dialog

6. Click the **Dial out and Receive calls** radio button and click **OK**.
7. Click **Network** to display **Network Configuration** settings.
8. Ensure the following:
   - In the **Dial out Protocols** area, ensure that *only* **TCP/IP** is checked.
   - In the **Server Settings** area, ensure that *only* **TCP/IP** is checked.
   - Click the **Require Microsoft encrypted authentication** radio button.
   - Clear the **Require data encryption** check box.
9. Click the **Configure** button next to the **TCP/IP** checkbox.
10. Ensure the following:
    - In the **Allow remote TCP/IP clients to access:** area, click the **Entire Network** radio button.
    - If your network is DHCP enabled, click the **Use DHCP to assign temporary TCP/IP addresses** radio button. If your network is **not** DHCP enabled, click the **Use static address pool** radio button and specify a range of IP addresses to use.
    - Click **OK** and then click **OK** again to start the RAS setup.
11. Click **Close** to close the **Network** dialog.
12. Restart the PC when prompted to do so.

Create a new user

Use the following procedure to create a new user in order to access the PC via RAS. The current user must be logged in to Windows NT as an administrator.

1. In the **Administrative Tools** folder of the **Start** menu, select **User Manager**.
2. To add a new user, click **User— > New User**.
3. Give the new user an appropriate user name, for example `palmuser`, and password, for example `mqe`.
4. Clear the **User Must Change Password at Next Logon** check box and select **User Cannot Change Password** and **Password Never Expires**.
5. Click **Dialin**.
6. Select the **Grant dialin permission to user** check box and click **OK**.
7. Click **OK** to complete the addition of the new user to the system.
8. Close the User Manager application.

To start RAS:

1. Ensure that the HotSync manager is no longer running, because it uses the same port as RAS. If there is a red and blue HotSync icon in the system tray (on the task bar), right-click on it and click **Exit**.
2. Click **Start— > Administrative Tools— > Remote Access Admin**.
3. In the **Remote Access Admin** application, click **Server— > Start Remote Access Service**. A dialog appears with the PC name in the text field.
4. Click **OK** to start RAS.

Windows 2000 and Palm OS 3.5

On the PC:

1. Click **Start— > Settings— > Control Panel — > Network and Dial-up Connections**
2. If the **Incoming Connections** option is not available,

   you need to install a new modem type.

If the **Incoming Connections** option is available, check that the correct modem is installed.

3. From the **Network and Dial-Up Connections** dialog, select **Incoming Connections**. Select the **Users** tab and select ″Always allow directly connected devices such as palmtop computers to connect without providing a password″.

The procedures for these tasks are described in the following sections.

Installing the communications cable

To install the modem, click **Control Panel** dialog and open the **Phone and Modem Options** dialog.

If a modem called "Communication cable between two computers" is installed, you do not need to install a modem and may go directly to the “Configuring the ″Incoming Connections″ option” section.

If this modem is not installed,

1. Click **Add** to start the **Install New Modem Wizard**.
2. Select the **Don’t Detect My Modem** check box and click **Next**. Windows will search for a modem connection.
3. Click **Next**. **(Standard Modem Types)** Manufacturer and the **Communications Cable between two computers** model.
4. Click **Next**. A Port selection dialog is displayed.
5. Click the port that the Palm cradle is plugged into and click **Next** to install the modem.
6. Click **Finish** to close the wizard . The new modem is added to the list.
7. Click **Properties**. Set the Maximum Speed to 19200
8. Click **Close** to close the **Modems** dialog.

Configuring the ″Incoming Connections″ option

To install RAS:

1. Click **Control Panel— > Network and Dial-Up Connections**.
2. Click **Make New Connection** to start the **Network Connection Wizard** and click **Next**.
3. Click **Accept Incoming connections— > Next**.
4. Click the **Communications cable between two computers** connection device and click **Next**.
5. In the **Incoming Virtual Private Connection** dialog, click **Do not allow virtual private connections— > Next**.
6. In the**Allowed Users** dialog select **Guest** and click **Next**.
7. In the **Networking Connections** dialog, ensure that **Internet Protocol (TCP/IP)** is selected (you do not need to clear the other check boxes that are selected) and click **Properties**.
8. In the **Incoming TCP/IP Properties** dialog:
   - Select **Allow callers to access my local area network**.
   - Click **Assign TCP/IP addresses automatically using DHCP**. If your network is **not** DHCP enabled, click the **Specify TCP/IP addresses** radio button and specify a range of IP addresses to use.
   - Select **Allow calling computer to specify its own IP address**.
9. In the **Complete the Network Connection** dialog, click **Finish**.

To start RAS:

1. Ensure that the HotSync manager is no longer running, because it uses the same port as RAS. If there is a red and blue HotSync icon in the system tray (on the task bar), right-click on it and click **Exit**.
2. From the Control Panel, select **Administrative Tools— > Remote Access Admin**.
3. In the **Remote Access Admin** application, click **Server— > Start Remote Access Service**. A dialog appears with the PC name in the text field.
4. Click **OK** to start RAS.

Configure networking and MQSeries Everyplace on Palm OS

This section shows you how to configure the Palm network the Palm OS device and Windows NT/2000.

Palm OS 3.1 and Windows NT

To configure the Palm network:

1. Start the **Prefs** application on the Palm OS device.
2. Tap **Network**.
3. Tap **Menu— > Service— > New**.
4. Give the service a name, for example `Windows RAS` and type the user name and password that you previously set on the PC.
5. Click in the **Phone** area and type the phone number "00". This signifies that no number should be dialed.
6. Click **Details**.
7. Set **Connection type** `PPP`, **Idle time-out** to `Power Off` and select **Query DNS** and **IP Address: Automatic**.
8. Click **Script** and type the following script:

```
   Send: CLIENT
Send CR:
  Delay: 1
   Send: CLIENTSERVER
    End:
```

9. Click **OK**.
10. Ensure that the device is in its cradle and that RAS is running on the PC and click **Connect** to connect to RAS. You need to set the port speed to 19000 on both the client and server side.
11. When the connection is made, click **Disconnect** to disconnect from RAS.
12. To configure MQSeries Everyplace on the Palm, open the **Preference** panel.
13. Pull down a list of preferences and click the **IBM** menu item to display the **IBM preference** panel. The following text should appear:

```
QMgr.Name.Local=LocalQM
ExampleQM.Adapter.Url=TcpipHttp:xx.xx.xx.xx:8081
ExampleQM.Adapter.Cmd=?Channel
```

    The following information enables the MQSeries Everyplace queue manager on the Palm to make a connection to an MQSeries Everyplace server and perform **PutMsg** and **GetMsg()** operations.

    - Define the `QMgr.Name.Local=LocalQM` entry, to use MQeInitialize() and start a session with the MQSeries Everyplace queue manager.

The device queue manager name is `LocalQM` and this must be unique within the connected WebSphere MQ network so that the queue manager on the server knows how to route messages to the device.

- Enter the connection definition that MQSeries Everyplace queue manager uses to make the connection. `TcpipHttp:xx.xx.xx.xx:8081` is the IP address of the queue manager whose name is `ExampleQM`. This name is the input parameter, *pQMName*, to all Queue Manager APIs.

  For this example, if you are running the MQSeries Everyplace example server, the queue manager name is `ExampleQM`, so only the IP address needs to be changed. Set this to the IP address of the computer the server is running on. If you are using a different queue manager, change the lines in the preference panel to your queue manager name and IP address.

  Instead of the IP address, you may use the host name, for example `abc.com`.

- Enter the connection command definition. For the HTTP 1.0 protocol, insert this command into every **POST** command. **?Channel** is the default command that is recognized by the MQSeries Everyplace HTTP server. You may replace this command with the name of a servlet to communicate with a HTTP Web server.

Palm OS 3.5 with Windows 2000

To configure the Palm network:

1. From the applications window, select **Prefs**.
2. From the drop down menu, select **Network** and enter the following information:

   Service
   Windows RAS

   User Name
   Your userid on the PC

   Password
   Your PC password

   Phone 00
3. Tap **Details** and enter the following information:

   Connection type
   PPP

   Idle timeout
   Power Off

   Query DNS
   Select

   IP Automatic
4. Click **Script** and type the following script:

```
   Send: CLIENT
 Send: CLIENT
   Wait For: CLIENTSERVER
 End:
```

5. Click **OK**.
6. Ensure that the device is in its cradle and that RAS is running on the PC and click **Connect** to connect to RAS. You need to set the port speed to 19000 on both the client and server side.
7. When the connection is made, click **Disconnect** to disconnect from RAS.
8. To configure MQSeries Everyplace on the Palm, open the **Preference** panel.

9. Pull down a list of preferences and click the **IBM** menu item to display the **IBM preference** panel. The following text should appear:

   ```
   QMgr.Name.Local=LocalQM
   ExampleQM.Adapter.Url=TcpipHttp:xx.xx.xx.xx:8081
   ExampleQM.Adapter.Cmd=?Channel
   ```

   The following information enables the MQSeries Everyplace queue manager on the Palm to make a connection to an MQSeries Everyplace server and perform **PutMsg** and **GetMsg()** operations.

   - Define the `QMgr.Name.Local=LocalQM` entry, to use MQeInitialize() and start a session with the MQSeries Everyplace queue manager.

     The device queue manager name is `LocalQM` and this must be unique within the connected WebSphere MQ network so that the queue manager on the server knows how to route messages to the device.
   - Enter the connection definition that MQSeries Everyplace queue manager uses to make the connection. `TcpipHttp:xx.xx.xx.xx:8081` is the IP address of the queue manager whose name is `ExampleQM`. This name is the input parameter, *pQMName*, to all Queue Manager APIs.

     For this example, if you are running the MQSeries Everyplace example server, the queue manager name is `ExampleQM`, so only the IP address needs to be changed. Set this to the IP address of the computer the server is running on. If you are using a different queue manager, change the lines in the preference panel to your queue manager name and IP address.

     Instead of the IP address, you may use the host name, for example `abc.com`.
   - Enter the connection command definition. For the HTTP 1.0 protocol, insert this command into every **POST** command. **?Channel** is the default command that is recognized by the MQSeries Everyplace HTTP server. You may replace this command with the name of a servlet to communicate with a HTTP Web server.

Create an MQSeries Everyplace queue manager and start an MQSeries Everyplace server on the PC

1. In the Java MQSeries Everyplace code, there are a number of Windows .bat files. Edit the CreateExampleQM.bat and the ExamplesAWTMQeServer.bat files so that the following line:

   ```
   call JavaEnv %1
   ```

   becomes

   ```
   call JavaEnv JVM
   ```

   where *JVM* is MS, SUN or IBM, depending on which Java Virtual Machine you are using.
2. Run CreateExampleQM.bat to create a queue manager called `ExampleQM` that listens on port "8081".
3. Run ExamplesAWTMQeServer.bat to start the AWT MQSeries Everyplace server.
4. In the **Example MQSeries Everyplace trace** dialog, select all the check boxes.
5. In the **View** menu, click the **System.Err** command, so that you can see all trace messages.

Run the Palm program on Palm OS

This section shows you how to run the Palm program on the Palm OS device and connect to the MQSeries Everyplace server.

Palm OS 3.1 and Windows NT

1. With RAS and the AWT MQSeries Everyplace server running, place the palm in its cradle.
2. From the Applications screen, double-click the **Starter** application.

   The MQSeries Everyplace code runs, makes a network connection, and puts a message to the server queue manager. A series of trace messages run on the server as the message is put. The program finishes when the basic user interface appears on the Palm screen.
3. To check if the message was delivered, look in folder where the queue manager keeps its messages. The message should be in the folder for the target queue. If you used the ExampleQM and put the message to the `SYSTEM.DEFAULT.LOCAL.QUEUE`, the message is found in `ExampleQM\Queues\ExampleQM\SYSTEM.DEFAULT.LOCAL.QUEUE`.

Palm OS 3.5 and Windows 2000

Note: For further information on the following commands, access the online help for Windows 2000 at *http://www.microsoft.com*

You must use the Command prompt to configure RAS on Windows 2000:

1. Select **Start — > Programs — > Accessories — > Command Prompt**.
2. Type `netsh` and press Enter.
3. The following list shows the commands that you must enter, followed by the values that the Command Prompt should return for a working RAS setup:

```
netsh>ras show authmode
        authentication mode = nodcc

netsh>ras show authtype
        Enabled Authentication Types:

        Code Meaning

        PAP Password Authentication Protocol
        SPAP Shiva Password Authentication Protocol
        MSCHAP Microsoft Challenge-Handshake Authentication Protocol
        MSCHAPv2 Microsoft Challenge-Handshake Authentication Protocol version 2

netsh>ras show link
        Enabled Link Options:

        Code Meaning

        SWC Provides software compression(MPPC)
        LCP Provides Link Control Protocol extensions from the PPP suite of protocols

netsh>ras show multilink
        Enables Multilink Options:

        Code Meaning

        BACP Provides Bandwidth Allocation Control Protocol
```

```
netsh>ras show user your userid

        User name: your userid
        Dialin: permit
        Callback policy: caller
        Callback number:

netsh>ras ip show config

        RAS IP config:

        Negotiation mode: allow
        Access mode: all
        Address request mode: allow
        Assignment method:  auto
        Pool:  0.0.0.0 to 0.0.0.0

netsh>ras ipx show config

        RAS IPX config:

        Negotiation mode: deny
        Access mode: serveronly
        Node number request mode: deny
        Assignment method: autosame
        IPX address pool base: 0
        IPX address pool size: 1000

netsh>ras netbeui show config

        RAS NBF config:

        Negotiation mode: deny
        Access mode: serveronly
```

4. Type `Exit` and press Enter to leave the command prompt.
5. To check if the message was delivered, look in the folder where the queue manager keeps its messages. The message should be in the folder for the target queue. If you used the ExampleQM and put the message to the `SYSTEM.DEFAULT.LOCAL.QUEUE`, the message is found in `ExampleQM\Queues\ExampleQM\SYSTEM.DEFAULT.LOCAL.QUEUE`.

The MQSeries Everyplace code runs, makes a network connection, and puts a message to the server queue manager. A series of trace messages run on the server as the message is put. The program finishes when the basic user interface appears on the Palm screen.

You can change the RAS configuration by replacing `show` with `set` in the previous commands, and appending a question mark. For example, the following allows you to set the mode that determines client dial-in authentication:

```
netsh>ras set authmode ?

    set authmode[mode=]STANDARD|NODCC|BYPASS
```

For futher information, see *http://www.microsoft.com*

Part 2. General programming guidance

Chapter 2. Starting and terminating a session with the MQSeries Everyplace system

All the 'C' MQSeries Everyplace Queue Manager API and Fields API calls, except the system calls, take a session handle as the first parameter (MQeInitialize returns the session handle). Also, all the APIs take pointers to a *Completion code* and *Reason code* as their last two parameters. This allows the APIs to return better diagnostic information than would be available from just a return code. The header file hmq.h contains definitions for possible values returned in the Completion code and the Reason code. Typically, an application tests the Completion code for an error or warning value `MQECC_ERROR` or `MQECC_WARNING` and takes appropriate action (which involves testing the Reason code to determine the cause of the problem).

Initialization and termination

For an application to work with MQSeries Everyplace , it must first establish a session with the MQSeries Everyplace system. This is achieved by calling the MQeInitialize API and saving the returned `MQEHSESS` for use on later MQSeries Everyplace API calls.

When the application has finished making MQSeries Everyplace calls, it can terminate its connection to MQSeries Everyplace by calling MQeTerminate (passing the session handle `MQEHSESS` as a parameter).

The following sample code fragment shows a session initialization and termination.

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode;
MQEINT32  reason;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
if (hSess!=MQEHANDLE_NULL) {
   MQeTerminate(hSess, &compcode, &reason);
}
```

initialization and termination

Chapter 3. Building a message object

All MQSeries Everyplace messages are organized and constructed using MQeFields objects. An MQeFields object is a generic container of one or more fields, and each field is a name-value pair. There are also special classes of MQeFields object that contain predefined fields. For example, the MQSeries Everyplace message object is an MQeFields object that **MQeQMgrPutMsg()** accepts and that **MQeQMgrGetMsg()** and **MQeQMgrBrowseMsgs()** return. Each MQeFields object has a *type* associated with it so that all fields objects in the MQSeries Everyplace system are type identified and can be type checked.

The generic MQeFields object can be used to build and organize data in a hierarchical manner. A set of related name-value fields can be put into an MQeFields object, that is then put into another MQeFields object that is in turn put into a message object for sending.

A *filter* is an MQeFields object that looks for specific fields in a message. The filter is passed to the **MQeQMgrGetMsg()** and **MQeQMgrBrowseMsgs()** API calls to look for messages that contain the same fields.

When a message object is put into the MQSeries Everyplace system, it is tagged with a unique *ID* that is made up of a unique value field and the origin queue manager name field. In the "C" API, the message object is tagged every time it is put into the MQSeries Everyplace system with the **MQeQMgrPutMsg()** call. This tagging guarantees that multiple calls to the **MQeQMgrPutMsg()** function with the same message object do not introduce duplicate messages into the MQSeries Everyplace network. Since each message object is tagged with a unique ID (UID) every message object retrieved from the MQSeries Everyplace system has a UID tag associated with it.

Allocating and freeing a message object

Because an MQSeries Everyplace message object is an MQeFields object, its construction is fundamentally the same. Both MQeFields and message objects are constructed by calling the **MQeFieldsAlloc** API. The *Type* parameter specifies whether an MQeFields or a message object is created. The **MQeFieldsAlloc** API returns a handle that is passed back in all fields API calls. Specifying a type of `MQE_OBJECT_TYPE_MQE_FIELDS` creates a fields object and specifying `MQE_OBJECT_TYPE_MQE_MSGOBJECT` creates a message object. Other types such as `MQE_OBJECT_TYPE_MQE_ADMIN_MSG` are also available (see the hmq.h file).

A message or MQeFields object that is no longer required should be destroyed to free resources back to the operating system. The **MQeFieldsFree** API is provided to destroy MQeFields based objects that were created with the **MQeFieldsAlloc** API. **MQeFieldsFree** takes the handle to the object (to be destroyed) as a parameter.

The following code fragment shows MQeFields objects being created and destroyed.

```
#include <hmq.h>
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
```

```
MQEHFIELDS hFlds, hMsg;

hSess = MQeInitialize("MyAppsName", &compcode &reason);
hFlds = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_FIELDS, &compcode , &reason);hMsg  = MQeFieldsAllo
MQeFieldsFree(hSess, hFlds, &compcode, &reason);
MQeFieldsFree(hSess, hMsg , &compcode, &reason);
MQeTerminate (hSess, &compcode, &reason);
```

Note that it is the responsibility of the application to delete message objects that are returned from MQSeries Everyplace even if the application did not create the message. For example, MQSeries Everyplace returns a message object from an **MQeQMgrGetMsg** API call and this must be deleted by the application.

Note: The "C" API returns a reason code of `MQE_EXCEPT_INVALID_HANDLE` when a `NULL` or previously allocated handle is passed to an API. However, if an arbitrary handle is passed then the API behavior is not defined.

Note: A previously allocated handle is one that the MQSeries Everyplace API returned to the application but it has been deleted and is no longer valid. Examples are a *`Session Handle`* that has subsequently been deleted by **MQeTerminate()**, or an *`MQeFieldsHandle`* that has been deleted by **MQeFieldsFree()**.

Putting data into the message object

To put data into a fields object, use the following fields API calls:

- MQeFieldsPut()
- MQeFieldsPutArray()
- MQeFieldsPutByArrayOfFd()
- MQeFieldsPutByStruct()
- MQeFieldsWrite()

MQeFieldsPut() is the most basic API and its use is described here. The other functions are described in the Chapter 7, “Advanced MQeFields APIs” on page 33 section of this document.

Every piece of data put into an MQeFields object has an MQSeries Everyplace field type associated with it. The type gives hints to the MQSeries Everyplace system on how to handle the message when it passes between different host system (for example between a big-endian and a little-endian system). In other words, the MQSeries Everyplace system converts a primitive data type into a host-friendly format so that the correct integer value is retrieved from the MQeFields object regardless of the format of the host system.

Table 1 shows the data types that are available in MQSeries Everyplace

Table 1. MQeFields object data types

Type	Data Representation
MQE_TYPE_UNTYPED	1 byte (8 bits)
MQE_TYPE_ASCII	1 byte (8 bits)
MQE_TYPE_UNICODE	2 bytes (16 bits)
MQE_TYPE_BOOLEAN	1 byte (8 bits)
MQE_TYPE_BYTE	1 byte (8 bits)

Table 1. MQeFields object data types (continued)

Type	Data Representation
MQE_TYPE_SHORT	2 bytes (16 bits)
MQE_TYPE_INT	4 bytes (32 bits)
MQE_TYPE_LONG	4 bytes (32 bits)
MQE_TYPE_FLOAT	4 bytes (32 bits)
MQE_TYPE_DOUBLE	8 bytes (64 bits)
MQE_TYPE_ARRAYELEMENTS	4 bytes (32 bits)
MQE_TYPE_FIELDS	(a handle) (32 bits)

Example code fragment for putting data into a message object

```
#include <hmq.h>
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds, hMsg;
static const MQECHAR Echo[] = "Hello world!";
MQEBYTE          testBool =0x1;
MQEBYTE          testByte =0xab, testBytes[]={ 0x12, 0x34, 0x56 };
MQEINT16         testShort=0xabcd,
                 testShorts[]={ 0x1234, 0x3456, 0x5678 };
MQEINT32         testInt=0xabcdef12,
                 testInts[]={ 0x12121212, 0x34343434, 0x56565656 };
struct MQEINT64  testLong={0x12345678, 0x9abcdef0},
                 testLongs[]={ {0x12, 0x34}, {0x56,0xab} };
MQEINT32         testData[256];
MQEINT16         i;
     hSess = MQeInitialize("MyAppsName", &compcode, &reason);     hFlds = MQeFieldsAlloc( hSess, M
     hMsg  = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_MSGOBJ, &compcode, &reason);

     /* Put in an ASCII string */
     MQeFieldsPut( hSess, hFlds, "hello"   , MQE_TYPE_ASCII, (void*)Echo, strlen(Echo),
                   &compcode, &reason);

     /* Put in an primitive data type */
     MQeFieldsPut( hSess, hFlds, "aBool"   , MQE_TYPE_BOOLEAN, (void*)&testBool,1,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aByte"   , MQE_TYPE_BYTE , (void*)&testByte,  1,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aShort"  , MQE_TYPE_SHORT, (void*)&testShort, 1,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aInt"    , MQE_TYPE_INT  , (void*)&testInt,   1,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aLong"   , MQE_TYPE_LONG , (void*)&testLong,  1,
                   &compcode, &reason);

     /* Put in an array of primitive data type */
     MQeFieldsPut( hSess, hFlds, "aBytes"  , MQE_TYPE_BYTE , (void*)testBytes,  3,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aShorts" , MQE_TYPE_SHORT, (void*)testShorts, 3,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aInts"   , MQE_TYPE_INT  , (void*)testInts,   3,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "aLongs"  , MQE_TYPE_LONG , (void*)testLongs,  2,
                   &compcode, &reason);
     MQeFieldsPut( hSess, hFlds, "testData", MQE_TYPE_INT  , (void*)testData, 256,
                   &compcode, &reason);

     /* Put the fields object into a message object. */
     MQeFieldsPut( hSess, hMsg, "aFldsObj" , MQE_TYPE_FIELD, (void*)&hFlds, 1,
```

```
                    &compcode, &reason);

MQeFieldsFree(hSess, hMsg , &compcode, &reason);
MQeTerminate (hSess, &compcode, &reason);
```

Chapter 4. Putting messages onto a queue

To put a message object onto a queue, use the **MQeQMgrPutMsg** API. This function takes a queue manager name and queue name pair. Since MQSeries Everyplace on the Palm has no local queue capability, it runs as a synchronous client to an MQSeries Everyplace remote queue manager. The queue manager name input parameter must be a remote queue manager.

MQeQMgrPutMsg takes an `MQEPMO` data structure as an input.

Note: Only the *`ConfirmId`* option is supported in MQSeries Everyplace Version 1.2.7.

The *`ConfirmId`* option is used to implement assured message delivery between the MQSeries Everyplace client and the server. When you specify this option with **MQeQMgrPutMsg**, the message is put onto the queue, but it is not made accessible until an **MQeQMgrConfirmMsg** is called on the message object. The application issues an **MQeQMgrConfirmMsg** call only after the **MQeQMgrPutMsg** has successfully returned. If the communication link fails during an **MQeQMgrPutMsg** call, the application should first call an **MQeQMgrUndo** when connection with the MQSeries Everyplace server is reestablished. This call removes the message that may or may not have been put on the queue with the previous **MQeQMgrPutMsg**. The application can then safely call an **MQeQMgrPutMsg** again, followed by an **MQeQMgrConfirmMsg** call.

These procedures are shown in the following code fragment.

```
#include <hmq.h>
static const MQECHAR pHello[] = "Hello world.";
MQEHSESS   hSess;
MQEHFIELDS hMsg;
MQEINT32   rc;
MQEINT32   compcode;
MQEINT32   reason;
MQEPMO     pmo = MQEPMO_DEFAULT; /* Defaul option */
MQECHAR    * qm, *q;

qm = "aQM";
q  = "QQ";

hSess = MQeInitialize("MyAppsName", &compcode &reason);
hMsg  = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_MSGOBJECT, &compcode &reason);
MQeFieldsPut(hSess, hMsg, "hi", MQE_TYPE_ASCII, pHello, sizeof(pHello), &compcode &reason);

/* Put message WITHOUT confirm */
MQeQMgrPutMsg( hSess, qm, q, &pmo, hMsg, &compcode &reason);

/* Put it again.  This is equivalent to the previous call */
MQeQMgrPutMsg( hSess, qm, q, NULL, hMsg, &compcode &reason);


/* Put msg with confirmID, follow by a ConfirmMsg() */

pmo.ConfirmId.hi = 0x2222;
pmo.ConfirmId.lo = 0x1111;
pmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

MQeQMgrPutMsg( hSess, qm, q, &pmo, hMsg, &compcode &reason);
```

```
/* Confirms the message, i.e., delete it off the queue. */
MQeQMgrConfirmMsg( hSess, qm, q, MQE_QMGR_OPTION_CONFIRM_PUTMSG, hMsg, &compcode &reason);

/* Put msg with confirmID, follow by a Undo() */

pmo.ConfirmId.hi = 0xabab;
pmo.ConfirmId.lo = 0xcdda;
pmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

MQeQMgrPutMsg( hSess, qm, q, &pmo, hMsg, &compcode &reason);

/* Undo the PutMsg(), i.e., delete it off the queue. */
MQeQMgrUndo( hSess, qm, q, pmo.ConfirmId, &compcode &reason);

/* Free the message handle */
MQeFieldsFree( hSess, hMsg, &compcode &reason);

MQeTerminate( hSess, &compcode &reason);
```

Chapter 5. Retrieving messages from a queue

MQeQMgrGetMsg and **MQeQMgrBrowseMsgs** are used to retrieve message object from a remote queue. Like **MQeQMgrPutMsg()**, these calls support the *ConfirmId* option. **MQeQMgrBrowseMsgs** also has a *Browse_Lock* option. One major difference between these two APIs is that **MQeQMgrGetMsg** returns only one message object, while **MQeQMgrBrowseMsgs** can return more than one message object as an array of message objects. These functions and their options are described below.

MQeQMgrGetMsg()

This is the basic get message call. It returns the first available message on the queue, and the message is deleted from the queue.

MQeQMgrGetMsg() with Filter

A filter constructed from an MQeFields object can be given to **MQeQMgrGetMsg()**, so that the first message on the queue that matches the filter is returned.

MQeQMgrGetMsg() with ConfirmId

The message object is returned to the caller, but, unlike the previous case, the message is not deleted from the queue. An **MQeQMgrConfirmMsg()** deletes the message from the queue, and an **MQeQMgrUndo()** makes the message object reappear on the queue again. When the returned message is on the queue it is accessible to subsequent **MQeQMgrGetMsg** and **MQeQMgrDeleteMsgs()** calls only if they contain the *UID* of the message object. The message is inaccessible to subsequent **MQeQMgrBrowseMsgs**.

A filter can be specified with this option.

MQeQMgrBrowseMsgs()

An array of message objects is returned to the caller. The messages are not deleted from the queue and they are accessible to subsequent **MQeQMgrBrowseMsgs** and **MQeQMgrGetMsg()** operations.

MQeQMgrBrowseMsgs() with Filter

A filter constructed from an MQeFields object can be given to this function call, so that only the messages that match the filter are returned. The messages are not deleted from the queue, and they are accessible to future operations.

MQeQMgrBrowseMsgs() with BROWSE_LOCK

With this option, an array of message objects is returned to the caller, together with a *lockID*. This *lockID* is returned in the option data structure `struct tagBrowseMsgOpts`. The *lockID* and *UID* of the message object are used as an input parameter to the **MQeQMgrUnlockMsgs** API to unlock one or more message object on the queue and make them accessible again.

A filter can be specified with this option.

Messages locked on the queue are accessible to a subsequent **MQeQMgrGetMsg()** call only if it includes a filter that contains the *lockID* of the message. The messages are also accessible to an **MQeQMgrDeleteMsgs()** call only if it contains the message *UID* as

an input parameter. Locked messages are inaccessible to future **MQeQMgrBrowseMsgs()** operations.

MQeQMgrBrowseMsgs() with BROWSE_LOCK and ConfirmId
Using these two options in combination gives the application programmer the flexibility of using either **MQeQMgrUnlockMsgs()** to unlock a specific message or **MQeQMgrUndo()** to unlock a group of messages.

A filter can be specified with this option.

These procedures are shown in the following code fragments:

MQeQMgrGetMsg() sample code fragment

```
#include <hmq.h>
MQEHSESS   hSess;
MQEHFIELDS hMsg, hFilter;
MQEINT32   compcode;
MQEINT32   reason;
MQEGMO     gmo = MQEGMO_DEFAULT;
MQECHAR  * aKey = "aKey", * qm, *q;

qm = "aQM";
q  = "QQ";

hSess = MQeInitialize("MyAppsName", &compcode, &reason);

/* Get msg with filter and confirmID*/

gmo.ConfirmId.hi = 0x2222;
gmo.ConfirmId.lo = 0x1111;
gmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

hFilter = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_FIELDS,
                          &compcode, &reason);
MQeFieldsPut( hSess, hFilter, "FindThis", MQE_TYPE_ASCII, aKey, strlen(aKey),
              &compcode, &reason);

/* Get a message that contains the field-name "FindThis", field-type of ASCII, and */
/* a field-value of "aKey". */
hMsg  = MQeQMgrGetMsg( hSess, qm, q, &gmo, hFilter,
                       &compcode, &reason);

if (compcode==MQECC_OK) {   /* Do something with the message. */

   /* Confirms the message, i.e., delete it off the queue. */
   MQeQMgrConfirmMsg( hSess, qm, q, MQE_QMGR_OPTION_CONFIRM_GETMSG, hMsg, &compcode, &reason)

   /* Free the message handle */
   MQeFieldsFree( hSess, hMsg, &compcode, &reason);
}

MQeFieldsFree( hSess, hFilter, &compcode, &reason);
MQeTerminate( hSess, &compcode, &reason);
```

MQeQMgrBrowseMsgs() sample code fragment

```
/*=============================================================================================
#include <hmq.h>
MQEHSESS   hSess;
MQEHFIELDS hFilter = MQEHANDLE_NULL;
MQEINT32   i, n, nMsgs;
MQEINT32   compcode;
MQEINT32   reason;
MQEBMO     bmo = MQEBMO_DEFAULT;
MQEHFIELDS pMsgs[2];
MQECHAR   *qm, *q;
```

```
qm = "MyQM";
q  = "QQ";
hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
nMsgs  = 2;

/*-------------------------------------*/
/* Browse with no locking or confirm ID */
/*-------------------------------------*/
n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                       pMsgs, nMsgs, &compcode, &reason);

/* Now set the browse option for lock and confirm */
bmo.Option = MQE_QMGR_BROWSE_LOCK | MQE_QMGR_CONFIRMID;
/* Set the confirm ID */
bmo.ConfirmId.hi = bmo.ConfirmId.lo = 0x12345678;

/*-------------------------------------*/
/* Browse and undo                     */
/*-------------------------------------*/
n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                     pMsgs, nMsgs, &compcode, &reason);

MQeQMgrUndo(hSess, qm, q, bmo.ConfirmId, &compcode, &reason, );

/*-------------------------------------*/
/* Browse and delete                   */
/*-------------------------------------*/
/* Browse nMsgs at a time until no messages are left */
while (1) { /* do forever */
   /* Browse the nMsgs matching messages */
   n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                       pMsgs, nMsgs, &compcode, &reason);

   if (n==0) {
      /* Any resources held by the cookie has been released already */
      break;
   }

   for(i=0; i<n; i++) {
      /*****************************************/
      /* Process the message objects in pMsgs[] */
      /*****************************************/
   }

   /* Delete the n locked messages in pMsgs[] */
   MQeQMgrDeleteMsgs( hSess, qm, q, pMsgs, n, &compcode, &reason);

   /* free pMsgs[] handle resources */
   for(i=0; i<n; i++) {
      MQeFieldsFree(hSess, pMsgs[i], &compcode, &reason);
   }
};

/* Deallocate the filter fields object handle */
MQeTerminate(hSess, &compcode, &reason);
```

Chapter 6. Retrieving data from message objects

Use the following MQeFields functions to extract data from a message object that has been retrieved from a queue.

- MQeFieldsGet()
- MQeFieldsGetArray()
- MQeFieldsGetByArrayOfFd()
- MQeFieldsGetByIndex()
- MQeFieldsGetByStruct()
- MQeFieldsWrite()

MQeFieldsGet is the basic extraction call and it is described here. The other functions are described in the Chapter 7, “Advanced MQeFields APIs” on page 33 section of this document.

Like **MQeFieldsPut()**, a field is retrieved by its name using the **MQeFieldsGet** call.

This API has two modes of operation, the first allows the interrogation of the fields to retrieve its length and the second mode retrieves the contents of the field into a storage area provided by the application. In both modes of operation, the field being targeted is identified by its name which is passed on the API call. The recommended way to use this API is:

1. Retrieve the length of a field.
2. Allocate enough storage to hold the contents of the field.
3. Get the contents of the field into the storage area.

MQeFieldsGet - Mode 1: length retrieval

To get the length of a field in a message, call the **MQeFieldsGet** API passing a pointer to the memory which will receive the data as a NULL pointer. The length of the field is passed back as the return value from the call.

The length of the field is the number of elements of the field datatype. this is *not* the same as the number of bytes. For example, for a field that has a single element of datatype `MQE_TYPE_INT`, this call returns a field length of "1".

MQeFieldsGet - Mode 2: Data Retrieval

To get the content of a field in a message, call the **MQeFieldsGet** API, passing a valid pointer to the memory that will receive the data. The data is returned into this memory.

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    datatype;
MQEINT32   n;
MQEBYTE *  pdata;
MQEBYTE *  buf;
MQEINT32   rc;
```

```
hSess = MQeInitialize("MyAppsName", &compcode , &reason);
MQEGMO gmo = MQEGMO_DEFAULT;
/*Set default get message options*/
 ...
/*
 * Get a message from a queue.
*/
hFlds = MQeQMgrGetMsg( hSess, "ExampleQM", "SYSTEM.DEFAULT.LOCAL.QUEUE", &gmo, NULL, &compcode, &rea

/* Get the field data length (into n) and type (into datatype) of a field (name: "XYZ")*/
n = MQeFieldsGet( hSess, hFlds, "XYZ", &datatype, NULL, 0, NULL, +
&compcode, &reason);

/* Verfify that datatype is correct. */

/* Get some space to put the data */
buf = (MQEBYTE *)calloc(n, MQE_SIZEOF(datatype));

/* Get the field data */
rc = MQeFieldsGet( hSess, hFlds, "XYZ", NULL, buf, n, NULL, &compcode, &reason);

/* Do something with the data in buf */

/* Free buf */
free( buf );

/* Free the fields object */
MQeFieldsFree( hSess, hFlds, &compcode, &reason);

/*Terminate the session */
MQeTerminate( hSess, &compcode, &reason);
```

Chapter 7. Advanced MQeFields APIs

Three sets of advanced MQeFields APIs are provided for experienced programmers who want to put and get data in and out of the fields object more efficiently.

- MQeFieldsGetByArrayOfFd() and MQeFieldsPutByArrayOfFd()
- MQeFieldsGetByStruct() and MQeFieldsPutByStruct()
- MQeFieldsRead() and MQeFieldsWrite()

The three sets are described below:

MQeFieldsGetByArrayOfFd() and MQeFieldsPutByArrayOfFd()

These APIs enable an application programmer to put and get an array of fields into and out of an MQeFields object. Instead of doing individual **MQeFieldsGet** and **MQeFieldsPut** calls on each field, think of this API as batch processing. It calls into the MQSeries Everyplace system library only once, as opposed to multiple times for the individual get and put calls. If used properly, this API improves the performance of the MQeFields API usage.

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR textVal[]  = "The Owl and the Pussy Cat went to sea.";
/* template for fields */
static  const MQEFIELD PFDS[] = {
        {MQE_TYPE_BYTE, 0, 7, "fooByte", (MQEBYTE *)0, 0, (MQEBYTE *)0},
        {MQE_TYPE_SHORT, 0, 8, "fooShort", (MQEBYTE *)0, 0, (MQEBYTE *)0},
        {MQE_TYPE_LONG, 0, 7, "fooLong", (MQEBYTE *)0, 0, (MQEBYTE *)0},
        {MQE_TYPE_ASCII, 0, 7, "fooText", (MQEBYTE *)0, 0, (MQEBYTE *)0},
        };
#define NFDS (sizeof(PFDS)/sizeof(PFDS[0]))
MQEHSESS   hSess;
MQEINT32   compcode;
MQEFIELD   Fds[NFDS];
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    byteVal;
MQEINT16   int16Val;
MQEINT32   int32Val;
MQEBYTE    datatype;
MQEINT32   rc;
MQEINT32   nFlds,i;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);


/* Put some fields in the fields object using MQeFieldsPutByArrayOfFd() */
byteVal  = 0xAE;
int16Val = 0x9876;
int32Val = 0x12345678;

/* Copy template */
memcpy(Fds,PFDS,sizeof(Fds));
Fds[0].fd_data = &byteVal;
Fds[0].fd_datalen = 1;
Fds[1].fd_data = &int16Val;
Fds[1].fd_datalen = 1;
Fds[2].fd_data = &int32Val;
Fds[2].fd_datalen = 1;
Fds[3].fd_data = &textVal;[0];
Fds[3].fd_datalen = sizeof(textVal);
```

```
compcode = MQECC_OK, reason = 0;
MQeFieldsPutByArrayOfFd( hSess, hFlds, Fds, NFDS , &compcode, &reason);

/* Copy template */
memcpy(Fds,PFDS,sizeof(Fds));

/* Get data lengths */
rc = MQeFieldsGetByArrayOfFd( hSess, hFlds, Fds, NFDS, &compcode, &reason);

/* Get space for each field data */
for( i=0; i<rc; i++) {
  int len = Fds[i].fd_datalen*MQE_SIZEOF(Fds[i].fd_datatype);
  if (len > 0) {
    Fds[i].fd_data = (MQEBYTE *) malloc(len);
  }
}

/* Get all the fields defined in field descriptor array in one shot */
compcode = MQECC_OK, reason = 0;
MQeFieldsGetByArrayOfFd( hSess, hFlds, Fds, NFDS, &compcode, &reason);
```

MQeFieldsGetByStruct() and MQeFieldsPutByStruct()

These APIs enable an application programmer to map a C data structure in the application program directly to a set of fields in the MQeFields object. By defining a fields structure descriptor for the C data structure, these two APIs automatically move the data between the C data structure and an MQeFields object.

The following code sample shows the use of these APIs:

```
#include <hmq.h>
   struct myData_st {
   MQEINT32 x;           /* simple variable */
   MQECHAR *name  ;      /* pointer to name buffer */
   MQEINT32 namelen;     /* length of name */
   MQEBYTE  buf[8];      /* fixed buffer in struct */
   MQEINT32 fieldlen;    /* length of a field, buffer not in struct */
};

MQEINT32 field[10];      /* buffer whose length is in a structure */

#ifndef MQE_OFFSETOF
#define MQE_OFFSETOF(_struct,_field) (&((struct _struct *)0._field))
#endif

/* A possible sample definition of MQEFIELDDESC for myData_st       */

static MQEFIELDDESC myDataStruct_fd[] = {
             {"x", 1, MQE_TYPE_INT, 0, MQE_OFFSETOF(myData_st,x), 1},
             {"name", 4, MQE_TYPE_ASCII, MQSTRUCT_LEN|MQSTRUCT_DATA,
              MQE_OFFSETOF(myData_st,name), MQE_OFFSETOF(myData_st,namelen)},
             {"buf",  3, MQE_TYPE_BYTE, 0, MQE_OFFSETOF(myData_st,buf), 8},
             {"field",5, MQE_TYPE_INT, MQSTRUCT_LEN|MQSTRUCT_NODATA,
              0, MQE_OFFSETOF(myData_st,fieldlen) }
};

static  MQECHAR * textVal  = "The Owl and the Pussy Cat went to sea.";
static  MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44 };
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
struct myData_st myData;
MQEINT32  int32Val;
MQEINT32  rc;

for (rc=0; rc<sizeof(field)/sizeof(field[0]); rc++) field[rc]=rc;
```

```
hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_FIELDS,
                          &compcode, &reason);

/* Put some fields into the fields object. */
int32Val = 0xABABBABA;
rc = MQeFieldsPut( hSess, hFlds, "x", MQE_TYPE_INT, &int32Val, 1,
                  &compcode, &reason);

rc = MQeFieldsPut( hSess, hFlds, "name", MQE_TYPE_ASCII, textVal, strlen(textVal),
                  &compcode, &reason);

rc = MQeFieldsPut( hSess, hFlds, "buf", MQE_TYPE_BYTE, textBuf,
                   sizeof(textBuf)/sizeof(textBuf[0]),
                  &compcode, &reason);

rc = MQeFieldsPut( hSess, hFlds, "field", MQE_TYPE_INT, &field,
                   sizeof(field)/sizeof(field[0]),
                  &compcode, &reason);

/* Retrieve all the fields out at once and populate the user data structure. */
rc = MQeFieldsGetByStruct( hSess, hFlds, &myData, myDataStruct_fd,
                           sizeof(myDataStruct_fd)/sizeof(myDataStruct_fd[0]),
                          &compcode, &reason);

printf("x = 0x%x, name = \"%s\", buf[0..3]=0x%08x-%08x\n",
        myData.x, myData.name, &myData.buf[0],
       &myData.buf[4]);

/* Output of printf() should look something like this */
/* "x = 0xABABBABA, name = "The Owl and the Pussy Cat went to sea.",
                           buf[0..3]=0xABCD1244-ABCD1248" */
```

MQeFieldsRead() and MQeFieldsWrite()

These APIs enable an application to stream data in and out of a field in an MQeFields object, so that data can be written a chunk at a time into a field or read a chunk at a time from a field. This enables the application to use a small intermediate transfer buffer to move large chunks of data.

```
#include <hmq.h>

static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEHFIELDS hFlds;
MQEINT32   compcode;
MQEINT32   reason;
MQEINT32   i, nread;
MQECHAR    buf[64];
MQEINT32   rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Allocate a 128 byte buffer field */
rc = MQeFieldsPut( hSess, hFlds, "y" , MQE_TYPE_BYTE, NULL , 128, &compcode , &reason);

/* Fill the buffer with values 0-127 */
for (i=0; i<128; i++) {
   char c=i;
   MQeFieldsWrite( hSess, hFlds, "y" , i, &c, 1,
                  &compcode, &reason);
}

/* Read 64 byte out into an output buf, nread = 64 */
nread  = MQeFieldsRead( hSess, hFlds, "y", MQE_TYPE_BYTE, buf, 0, 64, NULL,
                        &compcode , &reason);
```

advanced MQeFields API

Chapter 8. Starting and stopping the trace

The MQSeries Everyplace system has a built-in tracing capability for its own runtime tracing, and this tracing capability is also available to the application programmer.

An application needs to explicitly start and stop the trace using the **MQeTraceCmd()**API.

Note: For the Palm platform, a trace API, **MQeTrace()** is provided to write a trace string to the Palm MemoPad.

```
#include <hmq.h>

MQEHSESS hSess;
MQEINT32 compcode, reason;

hSess =MQeInitialize("MyAppsName",&compcode, &reason);

/*Start the trace */
MQeTraceCmd (hSess, MQE_TRACE_CMD_START, 0, &compcode, &reason);
MQeTraceCmd (hSess, MQE_TRACE_CMD_SET_MASK, MQE_TRACE_OPTION_SYS_ERROR, +
MQE_TRACE_OPTION_APP_MSG, &compcode, &reason);

MQeTrace(hSess, MQTS("Starting MQe..."));
MQeTrace(hSess, MQTS("This is an information trace message"));

/*Stop the trace */
MQeTraceCmd (hSess, MQE_TRACE_CMD_STOP, 0, &compcode, &reason);

/* Terminate the MQe session */
MQeTerminate (hSess, &compcode, &reason);
```

starting and stopping trace

Chapter 9. Administration using the administration message object

A queue manager is administered by sending messages to a special administration queue *AdminQ* that is owned by the local queue manager. The messages sent to this queue are interpreted and, if found to be valid administration messages, the commands contained within them are executed.

Before a synchronous client can send messages to a queue manager in the MQSeries Everyplace, system, the client must be configured with the IP address and other information for the target queue manager.

The client device can obtain this connection definition by putting a connection administration message (MQeConnectionAdminMsg) to its own local administration queue. This message contains all the addressing information required for the client to establish a connection to the server. Specifically, it contains an MQSeries Everyplace style Url indicating the address, and an additional command used to specify a servlet when interacting with a Web server.

The native code acts as a synchronous client (no visible queues), so the server is not able to put messages onto the client's administration queue directly. However, a client program can remove these connection administration messages from the server queue and put them onto the client administration queue.

The following actions can be used with connection administration messages:

MQE_MAM_ACTION_CREATE
: If no definition exists for the target queue manager, create one, else return an error code in any reply message

MQE_MAM_ACTION_DELETE
: If a definition exists for the target queue manager, delete it, else return an error code in any reply message

MQE_MAM_ACTION_INQUIRE
: If a definition exists for the target queue manager, then return the url and command, else return an error code in any reply message

The connection definitions, once created, are persistent. This means that when a connection definition is successfully put onto the local administration queue, the remote queue manager connection definition remains available until an application removes it.

See the *MQSeries Everyplace for Multiplatforms, Programming Guide* for further information about administration messages and their processing.

The following code shows an example of sending a connection administration message to a local synchronous client.

```
#include <hmq.h>
/* MQeMsgObject styles */
#define MQE_MSG_STYLE_DATAGRAM 0
#define MQE_MSG_STYLE_REQUEST  1
#define MQE_MSG_STYLE_REPLY    2

/* AdminMsg action codes (generic) */
```

```
#define MQE_MAM_ACTION_CREATE          1
#define MQE_MAM_ACTION_DELETE          2
#define MQE_MAM_ACTION_INQUIRE         4
#define MQE_MAM_ACTION_INQUIRE_ALL     5
#define MQE_MAM_ACTION_UPDATE          6
#define MQE_MAM_ACTION_UPDATE_REGISTRY 7

#define MQE_QAM_ACTION_ADD_ALIAS 52
#define MQE_QAM_ACTION_REMOVE_ALIAS 53

#define MQE_CAM_ACTION_ADD_ALIAS 52
#define MQE_CAM_ACTION_REMOVE_ALIAS 53

/* AdminMsg return codes */
#define MQE_MAM_RC_SUCCESS 0
#define MQE_MAM_RC_FAIL    1
#define MQE_MAM_RC_MIXED   2



/* Return a request admin message object of TYPE, targeted to QM,
   with the specified ACTION, STYLE, and CHARACTERISTICS.
 */
MQEHFIELDS hmqAdminMsg(MQEHSESS hSess, MQECHAR *Type, MQECHAR *qm,
       MQEINT32 action, MQEINT32 style,
       MQEHFIELDS characteristics,
       MQEINT32 *pCompCode, MQEINT32 *pReason)
{
  MQEHFIELDS res = MQEHANDLE_NULL;
  MQEBYTE rc = MQE_MAM_RC_SUCCESS;
  MQEINT32 cc,reason;
  struct MQeField_st mam_fd[] = {
    {  MQE_TYPE_INT, 0,  0, "admact", (MQEVOID *)0, 1, (MQEVOID *)0},
    {  MQE_TYPE_INT, 0,  0, MQE_MSG_STYLE, (MQEVOID *)0, 1, (MQEVOID *)0},
    {  MQE_TYPE_BYTE, 0, 0, "admrc", (MQEVOID *)0, 1, (MQEVOID *)0 },
    {  MQE_TYPE_ASCII, 0,  0, "admqmgr", (MQEVOID *)0, 0, (MQEVOID *)0},
    {  MQE_TYPE_FIELDS, 0,  0, "admparms", (MQEVOID *)0, 1,
       (MQEVOID *)0}
  };

  mam_fd[0].fd_data = (MQEVOID *)&action;
  mam_fd[1].fd_data = (MQEVOID *)&style;
  mam_fd[2].fd_data = (MQEVOID *)&rc;
  mam_fd[3].fd_data = (MQEVOID *)qm;
  mam_fd[3].fd_datalen = StrLen(qm);
  mam_fd[4].fd_data = (MQEVOID *)&characteristics;

  res = MQeFieldsAlloc(hSess,Type,pCompCode,pReason);
  if (*pCompCode != MQECC_OK) { goto exit; }

  cc = MQeFieldsPutByArrayOfFd(hSess,res,&;mam_fd,
       sizeof(mam_fd)/sizeof(mam_fd[0]),
       pCompCode,pReason);

 exit:
  if (*pCompCode != MQECC_OK && res != MQEHANDLE_NULL) {
    MQeFieldsFree(hSess,res,&cc,&reason);
    res = MQEHANDLE_NULL;
  }
  return res;

}


/* Return a connection admin message suitable for setting up a synchronous
   client connection to a queue manager.
   The client will use URL and CMD to establish a connection to
```

```
   queue manager QM.
 */
MQEHFIELDS hmqConnectionAdminMsg(MQEHSESS hSess, MQECHAR *qm,
 MQECHAR *url, MQECHAR *cmd,
 MQEINT32 *pCompCode, MQEINT32 *pReason)
{
  MQEINT32 cc, reason, n;
  MQEHFIELDS h1 = MQEHANDLE_NULL;
  MQEHFIELDS h2 = MQEHANDLE_NULL;
  MQEHFIELDS res = MQEHANDLE_NULL;
  struct MQeField_st fd[3];

  /* Allocate adapter fields object. */
  h1 = MQeFieldsAlloc(hSess,MQE_OBJECT_TYPE_MQE_FIELDS,pCompCode,pReason);
  if (*pCompCode != MQECC_OK) { goto exit; }

  /* Fill in adapter info */
  fd[0].fd_name = "cad";
  fd[0].fd_namelen = 3;
  fd[0].fd_datatype = MQE_TYPE_ASCII;
  fd[0].fd_base = (MQEVOID *)0;
  fd[0].fd_data = url;
  fd[0].fd_datalen = StrLen(url);
  fd[1].fd_name = "cadap";
  fd[1].fd_namelen = 5;
  fd[1].fd_datatype = MQE_TYPE_ASCII;
  fd[1].fd_base = (MQEVOID *)0;
  fd[1].fd_data = cmd;
  fd[1].fd_datalen = StrLen(cmd);

  cc = MQeFieldsPutByArrayOfFd(hSess,h1,&amp;fd,2,pCompCode,pReason);
  if (*pCompCode != MQECC_OK) { goto exit; }

  /* Allocate characeristics fields object. */
  h2 = MQeFieldsAlloc(hSess,MQE_OBJECT_TYPE_MQE_FIELDS,pCompCode,pReason);
  if (*pCompCode != MQECC_OK) { goto exit; }

  /* Fill in characteristics */
  n = 1; /* number of adapters */
  fd[0].fd_name = "cads:0";
  fd[0].fd_namelen = 6;
  fd[0].fd_datatype = MQE_TYPE_FIELDS;
  fd[0].fd_base = (MQEVOID *)0;
  fd[0].fd_data = (MQEVOID *)&h1;
  fd[0].fd_datalen = 1;
  fd[1].fd_name = "cads";
  fd[1].fd_namelen = 4;
  fd[1].fd_datatype = MQE_TYPE_ARRAYELEMENTS;
  fd[1].fd_base = (MQEVOID *)0;
  fd[1].fd_data = (MQEVOID *)&n;
  fd[1].fd_datalen = 1;
  fd[2].fd_name = "admname";
  fd[2].fd_namelen = 7;
  fd[2].fd_datatype = MQE_TYPE_ASCII;
  fd[2].fd_base = (MQEVOID *)0;
  fd[2].fd_data = qm;
  fd[2].fd_datalen = StrLen(qm);

  cc = MQeFieldsPutByArrayOfFd(hSess,h2,&fd,3,pCompCode,pReason);
  if (*pCompCode != MQECC_OK) { goto exit; }

  res = hmqAdminMsg(hSess,MQE_OBJECT_TYPE_MQE_CONNECTION_ADMIN_MSG,
    qm, MQE_MAM_ACTION_CREATE, MQE_MSG_STYLE_REQUEST, h2,
    pCompCode,pReason);
 exit:

  if (h1 != MQEHANDLE_NULL) {
```

```
    MQeFieldsFree(hSess,h1,&cc,&reason);
  }
  if (h2 != MQEHANDLE_NULL) {
    MQeFieldsFree(hSess,h2,&cc,&reason);
  }
  return res;
}

/* Configure local client to establish connections to QM via URL and CMD.
   Return zero on success.
*/
MQEINT32
hmqSetConnection(MQEHSESS hSess, MQECHAR *qm, MQECHAR *url, MQECHAR *cmd) {
  MQEHFIELDS cam;
  MQEINT32 cc,reason,res;

  cam = hmqConnectionAdminMsg(hSess,qm,url,cmd,&cc,&reason);
  if (cam != MQEHANDLE_NULL) {
    MQeQMgrPutMsg(hSess,"","AdminQ",(MQEVOID *)0,cam,&res,&reason);
    MQeFieldsFree(hSess,cam,&cc,&reason);
  }
  return res;
}
```

Part 3. Programming reference

Chapter 10. MQSeries Everyplace C API

This section contains details of the C language data types and the following C language API calls:

C language data types

This section contains information on the following data types used in the MQSeries Everyplace C client:

- Primitive data types
- Endian data types
- MQeFields Data Types

Primitive

The following platform-independent primitive data types are used throughout the C native APIs of MQSeries Everyplace:

Typedef name	Size (no. of bytes)	Alignment	Equivalent C data type	Equivalent Java data type
MQEBYTE	1	byte	Unsigned char	n/a
MQECHAR	1	byte	char	byte
MQEINT32	4	Even-byte	long	int
MQEINT64	8	Even-byte	longlong	long
PMQE*	4	Even-byte	"*"	n/a.
MQEH*	4	Even-byte	long	n/a

`MQEHANDLE_NULL` represents an invalid handle value for all handle types and functions that return handle values may return this value when an error occurs.

Endian

The endian of the data types is platform-dependent. For example, on an x86 based machine a multibytes data type is ordered in little-endian, (the least-significant byte occupies the lower memory address). The opposite is true on a big-endian 68k based machine. The data on a transmission medium is always big endian.

MQeFields data types

The following MQeFields data types are provided with MQSeries Everyplace:

Fields data type	Size in bytes
MQETYPE_UNTYPED	n/a
MQETYPE_ASCII	1
MQETYPE_UNICODE	2
MQETYPE_BOOLEAN	1
MQETYPE_BYTE	1
MQETYPE_SHORT	2
MQETYPE_INT	4
MQETYPE_LONG	8
MQETYPE_FLOAT	4
MQETYPE_DOUBLE	8
MQETYPE_ARRAYELEMENTS	4
MQETYPE_FIELDS	4

MQeFields API

Primitive

The MQeFields object is a container of zero or more fields. A field is identified by its field name, a null terminated string, a data type, and field data.

- Use **MQeFieldsPut()** to put a field into an MQeFields object.
- Use **MQeFieldsGet()** to get a copy of a field from an MQeFields object.
- Use **MQeFieldsDelete()** to remove a field from the MQeFields object.

General constraint

With MQSeries Everyplace Version 1.2.7 on a PalmOS device, the maximum number of MQeFields object handles is limited to 13, and the maximum MQeFields object (message object) size is 12 KB.

Array APIs

Two sets of APIs are defined to encode the MQeFields arrays. One set of APIs starts with **MQeFieldsArrayOf*** and the other starts with **MQeFields*Array**. These two sets of APIs look alike, but the underlying encoding scheme for the elements of the array is different.

MQeFieldsArrayOf*

These APIs treat the entire array as a single block of data, and a single field is used to hold this block. Once this block of data is included in the MQeFieldsArrayOf* APIs, the application program cannot delete or append to the individual elements in the array. These APIs operate on primitives data types, and they are:

- MQeFieldsGetArrayOfByte
- MQeFieldsGetArrayOfShort (`MQEINT16`)
- MQeFieldsGetArrayOfInt (`MQEINT32`)
- MQeFieldsGetArrayOfLong (`MQEINT64`)
- MQeFieldsGetArrayOfFloat
- MQeFieldsGetArrayOfDouble
- MQeFieldsPutArrayOfByte
- MQeFieldsPutArrayOfShort (`MQEINT16`)
- MQeFieldsPutArrayOfInt (`MQEINT32`)
- MQeFieldsPutArrayOfLong (`MQEINT64`)
- MQeFieldsPutArrayOfFloat
- MQeFieldsPutArrayOfDouble

MQeFields*Array

This set of APIs encodes each element of the array as a separate field, plus an extra field that holds the array length. In other words, this encoding scheme treats the array as a *vector*. The benefit of this encoding is that it allows the programmer to modify the individual element and to dynamically adjust the array size.

The encoding for each element in the array to a field consists of the following parts:

Field name
: This is created from the concatenation of the field name of the array; a separator character (which is a colon, ":"), and the index of the element.

For example, if the field name of the array is `foo`, then the field name for the first, second, and the nth elements are `foo:0`, `foo:1` and `foo:n-1`.

Field type
The same type as the array.

Field data
Individual element of the array.

The array length, (the number of elements), is encoded in a separate field and is accessible to the programmer using the **MQeFieldsGetArrayLength** and **MQeFieldsPutArrayLength** APIs.

This set of APIs includes the following:

- MQeFieldsGetAsciiArray
- MQeFieldsGetByteArray
- MQeFieldsGetShortArray (`MQEINT16`)
- MQeFieldsGetIntArray (`MQEINT32`)
- MQeFieldsGetLongArray (MQEINT64)
- MQeFieldsGetFloatArray
- MQeFieldsGetDoubleArray
- MQeFieldsGetUnicodeArray
- MQeFieldsPutAsciiArray
- MQeFieldsPutByteArray
- MQeFieldsPutShortArray (`MQEINT16`)
- MQeFieldsPutIntArray (`MQEINT32`)
- MQeFieldsPutLongArray (`MQEINT64`)
- MQeFieldsPutFloatArray
- MQeFieldsPutDoubleArray
- MQeFieldsPutUnicodeArray

Base APIs

Table 2 lists the core MQeFields APIs.

Table 2. MQeFields base API

API	Description
MQeFieldsAlloc()	Allocate a new MQeFields object and returns a handle to it.
MQeFieldsDelete()	Delete an existing field in the MQeFields object.
MQeFieldsDump()	Serialize the internal name/value pair fields into a byte array for storage or communication.
MQeFieldsDumpLength()	Get the total number of bytes needed to hold the serialized fields in the MQeFields object.
MQeFieldsEquals()	Compare two MQeFields object and determines if they are the same.
MQeFieldsFields()	Return the number of fields in the MQeFields object.

Table 2. MQeFields base API (continued)

API	Description
MQeFieldsFree()	Deallocate an MQeFields object and recovers its resources.
MQeFieldsGet()	Given a field name, return the field.
MQeFieldsGetArray()	Given a name, returns an array from fields generated by the name.
MQeFieldsGetByArrayOfFd()	Get an array of fields.
MQeFieldsGetByIndex()	Given an index, return the field at the index.
MQeFieldsGetByStruct()	Given a data structure and its fields structure descriptor, populate the data structure with the fields.
MQeFieldsHide()	Exclude a field from an MQeFields comparison API, **MQeFieldsEquals()**
MQeFieldsPut()	Put a field into an MQeFields object.
MQeFieldsPutArray()	Given a name, put an array as fields generated by the name.
MQeFieldsPutByArrayOfFd()	Given an array of field descriptors and associated field data, put them into the fields.
MQeFieldsPutByStruct()	Given a data structure and its fields structure descriptor, create the fields.
MQeFieldsRead()	Read from a field as an output stream.
MQeFieldsRestore()	Resolve a byte array into name/value pair fields and store them in an MQeFields object.
MQeFieldsType()	Extract the object type of an MQeFields object.
MQeFieldsWrite()	Write to a field as an input stream.

MQeFields macros and helper APIs

The APIs and macros listed in Table 3 are supplied for compatibility with the Java API. These APIs are built on top of the APIs listed in Table 2 on page 48.

Table 3. MQeFields macros and helper APIs

API	Description
MQeFieldsContains()	Determine if the MQeFields object contains a specific field.
MQeFieldsCopy()	Copy one or all fields from one MQeFields object to another.
MQeFieldsDataLength()	Determine the size of the data.
MQeFieldsDataType()	Determine the data type of a field.
MQeFieldsGetArrayLength()	Extract the length of an array.
MQeFieldsGetArrayOfByte()	Extract an array of byte from an MQeFields object.
MQeFieldsGetArrayOfDouble()	Extract an array of doubles (`MQEDOUBLE`) from an MQeFields object.

MQeFields APIs

Table 3. MQeFields macros and helper APIs (continued)

API	Description
MQeFieldsGetArrayOfFloat()	Extract an array of floats (MQEFLOAT) from an MQeFields object.
MQeFieldsGetArrayOfInt()	Extract an array of 32 bit integers (MQEINT32) from an MQeFields object.
MQeFieldsGetArrayOfLong()	Extract an array of 64 bit integers (MQEINT64) from an MQeFields object.
MQeFieldsGetArrayOfShort()	Extract an array of 16 bit integers (MQEINT16) from an MQeFields object.
MQeFieldsGetAscii()	Extract the data from an MQeFields object as an ASCII string.
MQeFieldsGetAsciiArray()	Extract the data from an MQeFields object as an array of ASCII strings.
MQeFieldsGetBoolean()	Extract the data from an MQeFields object as a boolean value.
MQeFieldsGetByte()	Extract data from an MQeFields object as a byte (MQEBYTE).
MQeFieldsGetByteArray()	Extract data from an MQeFields object as an array of byte arrays.
MQeFieldsGetDouble()	Extract data from an MQeFields object as a double floating point (MQEDOUBLE).
MQeFieldsGetDoubleArray()	Extract data from an MQeFields object as a double floating point array.
MQeFieldsGetFields()	Extract a field object handle (MQEHFIELD) from an MQeFields object.
MQeFieldsGetFloat()	Extract data from an MQeFields object as a float (MQEFLOAT).
MQeFieldsGetFloatArray()	Extract data from an MQeFields object as a float (MQEFLOAT) array.
MQeFieldsGetInt()	Extract data from an MQeFields object as an integer (MQEINT32).
MQeFieldsGetIntArray()	Extract data from an MQeFields object as an integer (MQEINT32) array.
MQeFieldsGetObject()	Extract the object type of an MQeFields object.
MQeFieldsGetLong()	Extract data from an MQeFields object as a 64 bit (MQEINT64) integer.
MQeFieldsGetLongArray()	Extract data from an MQeFields object as a 64 bit (MQEINT64) integer array.
MQeFieldsGetShort()	Extract data from an MQeFields object as a 16 bit (MQEINT16) short.
MQeFieldsGetShortArray()	Extract data from an MQeFields object as a 16 bit (MQEINT16) short array.
MQeFieldsGetUnicode()	Extract data from an MQeFields object as a Unicode string.
MQeFieldsGetUnicodeArray()	Extract data from an MQeFields object as a Unicode array.
MQeFieldsPutArrayLength()	Put an array length.

Table 3. MQeFields macros and helper APIs (continued)

API	Description
MQeFieldsPutArrayOfByte()	Put an array of byte (MQEBYTE) into an MQeFields object.
MQeFieldsPutArrayOfDouble()	Put an array of double (MQEDOUBLE) into an MQeFields object.
MQeFieldsPutArrayOfFloat()	Put an array of float (MQEFLOAT) into an MQeFields object.
MQeFieldsPutArrayOfInt()	Put an array of 32 bit (MQEINT32) integer into an MQeFields object.
MQeFieldsPutArrayOfLong()	Put an array of 64 bit (MQEINT64) integer into an MQeFields object.
MQeFieldsPutArrayOfShort()	Put an array of 16 bit (MQEINT16) integer into an MQeFields object.
MQeFieldsPutAscii()	Put an ascii string into an MQeFields object.
MQeFieldsPutAsciiArray()	Put an array of ascii strings into an MQeFields object.
MQeFieldsPutBoolean()	Put a boolean value into an MQeFields object.
MQeFieldsPutByte()	Put a byte (MQEBYTE) value into an MQeFields object.
MQeFieldsPutByteArray()	Put an array of byte (MQEBYTE) arrays into an MQeFields object.
MQeFieldsPutDouble()	Put a double (MQEDOUBLE) into an MQeFields object.
MQeFieldsPutDoubleArray()	Put an array of doubles (MQEDOUBLE) into an MQeFields object.
MQeFieldsPutFields()	Put a field object handle into an MQeFields object.
MQeFieldsPutFloat()	Put a float (MQEFLOAT) into an MQeFields object.
MQeFieldsPutFloatArray()	Put an array of floats (MQEFLOAT) into an MQeFields object.
MQeFieldsPutInt()	Put a 32 bit (MQEINT32) integer into an MQeFields object.
MQeFieldsPutIntArray()	Put an array of 32 bit (MQEINT32) integers into an MQeFields object.
MQeFieldsPutLong()	Put a 64 bit (MQEINT64) integer into an MQeFields object.
MQeFieldsPutLongArray()	Put an array of 64 bit (MQEINT64) integers into an MQeFields object.
MQeFieldsPutShort()	Put a 16 bit (MQEINT16) short integer into an MQeFields object.
MQeFieldsPutShortArray()	Put an array of 16 bit (MQEINT16) short integers into an MQeFields object.
MQeFieldsPutUnicode()	Put an Unicode string into an MQeFields object
MQeFieldsPutUnicodeArray()	Put an array of Unicode strings into an MQeFields object.

Data type definitions

The data types shown in Table 4 are used in the definitions of the APIs.

Table 4. MQeFields data type definitions

API	Description
MQECHAR	A signed 8-bit integer.
MQETCHAR	A platform dependent character.
MQEBYTE	An unsigned 8-bit integer.
MQEINT16	A two-byte integer that is aligned on even-byte boundary.
MQEINT32	A four-byte integer that is aligned on even-byte boundary.
MQEINT64	An eight-byte integer that is aligned on even-byte boundary.
MQEFLOAT	A four-byte floating point that is aligned on even-byte boundary.
MQEDOUBLE	An eight-byte floating point that is aligned on quad-byte boundary.
MQECHAR *	A null terminated ASCII character array of `MQECHAR`.
MQETCHAR *	A null terminated Unicode character array of `MQETCHAR`.

MQeField data structure

The field descriptor data structure contains information about a field in the MQeFields object. It is used as an input and an output parameter with **MQeFieldsGetByArrayOfFd** and as an output parameter with **MQeFieldsGetByIndex**.

```
  MQEFIELD {
  MQEBYTE  fd_datatype;           /* Field data type */
  MQEBYTE  __pad;              /* Unused padding byte */
  MQEINT16  fd_namelen;            /* Field name */
  MQECHAR * fd_name;             /* Pointer to the field name */
  MQEBYTE * fd_data;             /* Pointer to the field data */
  MQEINT32  fd_datalen;            /* Number of datatype elements in */
                       /* the field data */
  MQEBYTE * fd_base;             /* Base pointer (platform specific) */
};
```

MQECHAR * *fd_name*

A pointer to the null terminated string name of the field. Application programs should use the following guidelines for field names:

- At least 1 character long.
- Conform to the ASCII character set, (characters with values between 20 and 128)
- Should not include any of the characters **{}[]#():;,'=**

MQEINT32 *fd_namelen*

The length of the *fd_name*. The input value specifies the size (in `MQECHAR`) of the *fd_name* buffer for operations that retrieve the name of a field. The output value specifies the size (in `MQECHAR`) of the *fd_name* of the field, for operations that retrieve the name of a field. These sizes do not include a terminating `NULL`.

MQEBYTE *fd_type*
The data type of the field data.

MQEBYTE * *fd_data*
A pointer to the field data.

MQEINT32 *fd_datalen*
The number of data elements (not bytes) in *fd_data* . The input values of *fd_datalen* and *fd_datatype* specify the size of the buffer provided by *fd_data* (when not NULL). The output value specifies the total number of elements for the field.

MQEBYTE * *fd_base*
The platform specific base pointer for data, should be NULL unless specifically being used.

MQeField structure descriptor

The MQeField structure descriptor holds information about a field to be added to or retrieved from an MQeFields object using the **MQeFieldsPutByStruct** and **MQeFieldsGetByStruct** APIs.

```
typedef struct MQeFieldStructDescriptor_st {
  PMQECHAR  sd_name;    /* Pointer to the field name */
  MQEINT32  sd_namelen;   /* Length of field name */
  MQEBYTE  sd_datatype;  /* Type of field */
  MQEBYTE  sd_flags;    /* flags describing field layout in struct */
  MQEINT32  sd_dataoff;   /* data offset in struct */
  MQEINT32  sd_datalen;   /* (offset of) data length for field */
} MQEFIELDDESC;
```

PMQECHAR *sd_name*
A pointer to the null terminated string name of the field. Application programs should use the following guidelines for field names:

- At least 1 character long.
- Conform the ASCII character set, (characters with values between 20 and 128)
- Should not include any of the characters **{}[]#():;,'=**

MQEINT32 *sd_namelen*
The length of the *sd_name*.

MQEBYTE *sd_datatype*
The data type of the field data.

MQEBYTE * *sd_flags*
Flags that describe the type of data to put or get. See **MQeField Structure Descriptor Flags**

MQEINT32 *sd_dataoff*
The offset of the element to get or put.

MQEBYTE * *sd_datalen*
The length of the element to get or put.

MQeFields structure descriptor flags

The field structure descriptor *sd_flags* field can be initialized with flags that define the operation of the **MQeFieldsPutByStruct** and **MQeFieldsGetByStruct** APIs.

Name	Value	Action
MQSTRUCT_LEN	0x1	*struct offset sd_datalen* is a pointer to length, not number of elements
MQSTRUCT_DATA	0x2	*struct offset sd_dataoff* is a pointer to data, not start of data block
MQSTRUCT_NODATA	0x4	Get operations only extract the length of the field's data. Storage for the data is managed separately. Put operations ignore descriptors with this bit set.

Field data types

Each field in the MQeFields object is tagged with one of the data types defined below. The size of a single element of the data type is specified below.

Type	Value	Data Representation
MQE_TYPE_UNTYPED	0xC0	1 byte (8 bits)
MQE_TYPE_ASCII	0xC1	1 byte (8 bits)
MQE_TYPE_UNICODE	0xC2	2 byte (16 bits)
MQE_TYPE_BOOLEAN	0xC3	1 byte (8 bits)
MQE_TYPE_BYTE	0xC4	1 byte (8 bits)
MQE_TYPE_SHORT	0xC5	2 byte (16 bits)
MQE_TYPE_INT	0xC6	4 byte (32 bits)
MQE_TYPE_LONG	0xC7	4 byte (32 bits)
MQE_TYPE_FLOAT	0xC8	4 byte (32 bits)
MQE_TYPE_DOUBLE	0xC9	8 byte (64 bit)
MQE_TYPE_ARRAYELEMENTS	0xCA	4 byte (32 bits)
MQE_TYPE_FIELDS	0xCB	4 byte (a handle) (32 bits)

Base pointers

Many of the base APIs include a platform specific **base pointer** that has a platform specific interpretation. For platforms without any interpretation, it should be set to `NULL`.

Platform interpretations

The following platforms have an interpretation for the base pointer.

PalmOS

Under PalmOS, the base pointer is interpreted as the base of a locked database record when the corresponding destination buffer is a location within the locked record. If the destination buffer is regular memory, the base pointer should be `NULL`.

MQeFieldsAlloc

Description

Allocates a new MQeFields object and returns a handle to it. The handle represents an MQeFields object. It must be specified on all subsequent calls from the application that access the MQeFields object. This handle ceases to be valid when the **MQeFieldsFree** call is issued, or when the unit of processing that defines the scope of the handle terminates.

Syntax

```
#include <hmq.h>
MQEHFIELDS MQeFieldsAlloc( MQEHSESS hSess, MQECHAR * Type,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by MQeInitialize.

MQECHAR * *Type* - input

"" or NULL
An untyped MQeFields object that is used for restore, **MQeFieldsRestore**.

com.ibm.mqe.MQeFields
The base MQeFields object type

com.ibm.mqe.MQeMsgObject
A field object with two additional MQeFields, a 64 bit unique identifier, and the string name of the origin queue manager.

com.ibm.mqe.MQeAdminMsg

com.ibm.mqe.MQeQueueAdminMsg

com.ibm.mqe.MQeQueueManagerAdminMsg

com.ibm.mqe.MQeFragmentor

A non-recognized type string defaults to the base field object type with its type string set to the input type string.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEHFIELDS *hFlds*
The handle to an MQeFields object. If any error occurs during the allocation, then an MQEHANDLE_NULL is returned.

Implementation

On PalmOS 3.0 the underlying storage allocation element comes from a record in the database.

Example

```
#include <hmq.h>
static static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
```

See Also

- **MQeFieldsFree**

MQeFieldsDelete

Description

Delete a field in the MQeFields object.

Given a field name, remove its associated field from the MQeFields object.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsDelete( MQEHSESS hSess,MQEHFIELDS hFlds, MQECHAR * pName,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
Handle to an MQeFields object.

MQECHAR * *pName* - input
Null terminated string name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND

The field was not found in the MQeFields object.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32
Returns "0" on success, or "-1" on failure.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/*
 * Add some fields to the fields object... and one of them is "XYZ"
 */
 ...

/*
 * Now delete field named "XYZ"
 */
rc = MQeFieldsDelete( hSess, hFlds, "XYZ", &compcode, &reason);
```

See Also

MQeFieldsPut

MQeFieldsDump

Description

Serializes the encoded fields in an MQeFields object into a byte array.

This functions supports partial dumps. Between partial dumps, the programmer should not add or delete any field in the MQeFields object. If fields are added or deleted, inconsistencies may occur between the data that has already been copied out and the data that is waiting to be copied. This causes errors when the user tries to restore the MQeFields object from the byte array.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsDump( MQEHSESS hSess,MQEHFIELDS hFlds, MQEINT32 srcOff,
            MQEBYTE pBuf[], MQEINT32 BufLen, MQEVOID *pBase,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
Handle to an MQeFields object.

MQEINT32 *srcOff* - input
The offset into internal byte array representation of the MQeFields object at which the dump should start

MQEBYTE *pBuf[]* - output
The buffer to hold the dumped bytes

MQEINT32 *BufLen* - input
The number of bytes to dump

MQEVOID *pbase* - input
The base pointer for the output buffer *pBuf*

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned *pCompCode* equals MQECC_ERROR, *pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of bytes copied into the buffer.
- On failure, returns "-1".

Implementation

The byte order in which the primitive data types are dumped is big-endian.

Example

```
/**
   This example shows how to dump the fields object into an array of fix-size buffers.
 */
#include <hmq.h>
```

```
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";

static   MQECHAR * textBuf  = "The Owl and the Pussy Cat went to sea.";
static   MQEBYTE   byteBuf[] = { 0xAB, 0xCD, 0x12, 0x44};
#define  MAX_CHUNK_SIZE 16
MQEHSESS   hSess;
MQEHFIELDS hFlds;
MQEINT32   compcode;
MQEINT32   reason;
MQEINT32   int32Val;
MQEINT32   i, offset;
MQEINT32   chunk, nchunks;
MQEBYTE ** buf_array;
MQEINT32   nbytes;
MQEINT32   rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put some fields into the fields object */
int32Val = 0x12345678;
rc = MQeFieldsPut( hSess, hFlds, "x" , MQE_TYPE_INT, &int32Val, 1, &compcode, &reason);
rc = MQeFieldsPut( hSess, hFlds, "nm", MQE_TYPE_ASCII, textBuf, strlen(textBuf), &compcode
rc = MQeFieldsPut( hSess, hFlds, "b" , MQE_TYPE_BYTE, byteBuf, 4, &compcode, &reason);

nbytes  = MQeFieldsDumpLength( hSess, hFlds, &compcode, &reason);

offset  = 0;
i       = 0;

/*
 * Calc number of chunks needed to hold the dump byte array
 */
nchunks = nbytes/MAX_CHUNK_SIZE;
chunk = nbytes%MAX_CHUNK_SIZE;

/*
 * Allocate the buf array.
 */

while (nchunks != 0)
{
   buf_array[i] = (MQEBYTE*) malloc(MAX_CHUNK_SIZE);
   rc = MQeFieldsDump( hSess, hFlds, offset, &buf_array;[i][0], MAX_CHUNK_SIZE, NULL,&comp
   offset  += MAX_CHUNK_SIZE;
   nchunks--;
   i++;
}

buf_array[i] = (MQEBYTE*) malloc(chunk);
rc = MQeFieldsDump( hSess, hFlds, offset, &buf_array;[i][0], chunk, NULL, &compcode, &reas

/* Do something with the buf_array[], like store it into a file. */
```

See Also

- **MQeFieldsDumpLength**
- **MQeFieldsRestore**

MQeFieldsDumpLength

Description

Returns the total number of bytes that are used to hold the fields in this MQeFields object. The application programmer can use this number to allocate a memory chunk to hold all the fields.This API is used in conjunction with **MQeFieldsDump()**.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsDumpLength( MQEHSESS hSess,MQEHFIELDS hFlds,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *Flds* - input
The handle to an MQeFields object.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values::

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- On success, returns the number of bytes used to hold the field object data.
- On failure, returns "-1".

Example

See example in MQeFieldsDump.

See Also

- **MQeFieldsDump**
- **MQeFieldsRestore**

MQeFieldsEquals

Description

Compares two typed MQeFields objects and determine if they both have the same fields.

This API determines if two MQeFields objects are the same by comparing every visible field in the first object with the corresponding visible field in the second object. If the second object does not have a corresponding visible field, or its value is different, then the two MQeFields objects are considered unequal, and the result is "0". If the two MQeFields objects are not unequal, then the result depends on whether the second MQeFields object contains exactly the same number of visible fields, in which case the result is "1", or more visible fields, in which case the result is "2". This comparison does not depend on the order in which the fields are inserted or stored in each of the MQeFields objects; all that matters is that both MQeFields objects contain the same fields. The types of the MQeFields objects do not affect the result of the comparison, however both MQeFields objects must be typed (they may not be allocated with type MQE_OBJECT_TYPE_MQE_FIELDS_UNTYPE). The test recurses into nested fields.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsEquals( MQEHSESS hSess, MQEHFIELDS hFlds1, MQEHFIELDS hFlds2,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds1* - input
The first MQeFields object handle.

MQEHFIELDS *hFlds2* - input
The second MQeFields object handle.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

- **2** Every visible field in *hFlds1* has an equivalent visible field in *hFlds2*, but *hFlds2* has additional fields.
- **1** Every visible field in *hFlds1* has an equivalent visible field in *hFlds2*, and *hFlds2* has no other visible fields.
- **0** At least one visible field in *hFlds1* is either not present, hidden, or visible but not equivalent in *hFlds2*.
- **–1** Error.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds1, hFlds2;
```

MQeFieldsEquals

```
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason );
hFlds1 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
hFlds2 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/*
 * Add some fields to the fields objects... and one of them is "XYZ"
 */
 ...

/*
 * Now test their equivalence
 */
rc = MQeFieldsEquals( hSess, hFlds1, hFlds2, &compcode, &reason);
```

See Also MQeFieldsHide.

MQeFieldsFields

Description

Returns the total number of fields in an MQeFields object.

From this number, the application can use **MQeFieldsGetByIndex** to iterate through the indices and retrieve all the fields in the MQeFields object.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsFields( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32
On success, returns the number of fields.

Example

```
#include <hmq.h>
static  MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static  MQECHAR * textBuf  = "The Owl and the Pussy Cat went to sea.";
static  MQEBYTE   byteBuf[] = { 0xAB, 0xCD, 0x12, 0x44 };
MQEHSESS   hSess;
MQEHFIELDS hFlds;
MQEINT32   compcode;
MQEINT32   reason;
MQEINT32   int32Val;
MQEINT32   nFlds;
MQEINT32   rc,i;
MQEFIELD   fd;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

int32Val = 0x12345678;
rc = MQeFieldsPut( hSess, hFlds, "x" , MQE_TYPE_INT,  &int32Val;, 1, &compcode, &reason);
rc = MQeFieldsPut( hSess, hFlds, "nm", MQE_TYPE_ASCII, textBuf, strlen(textBuf), &compcode
rc = MQeFieldsPut( hSess, hFlds, "b" , MQE_TYPE_BYTE,  byteBuf, 4, &compcode, &reason);

nFlds = MQeFieldsFields( hSess, hFlds, &compcode, &reason);
/* nFlds is 4 (3 added above + 1 field object identifier field)  */
/* Ignore the first field (field object identifier) - start at 1 */
for (i=1; i<nFlds; i++) {

   memset( &fd, 0, sizeof(fd));

   /* Get each field by index */
```

```
rc = MQeFieldsGetByIndex( hSess, hFlds, i, &fd, 1, &compcode, &reason);
fd.fd_name = (MQECHAR *) malloc(fd.fd_namelen+1);
fd.fd_data = (MQEBYTE *) malloc(fd.fd_datalen * MQE_SIZEOF(fd.fd_datatype));
rc = MQeFieldsGetByIndex( hSess, hFlds, i, &fd, 1, &compcode, &reason);
fd.fd_name[fd.fd_namelen] = '\0';

free(fd.fd_name);
free(fd.fd_data);
```

See Also

MQeFieldsGetByIndex

MQeFieldsFree

Description

Deallocates an MQeFields object and recovers its resources.

Syntax

```
#include <hmq.h>
MQEVOID MQeFieldsFree( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input

The handle to an MQeFields object.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEVOID

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
MQeFieldsFree( hSess, hFlds, &compcode, &reason);
```

See Also

MQeFieldsAlloc

MQeFieldsGet

Description

Given a field name, retrieves the data type, length of field data and field data. This API is used to retrieve information about a field with a given name. The datatype is returned in the pointer. **MQeFieldsGetByArrayOfFd** should be used to get a field with a specific datatype.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsGet( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQECHAR * pName, MQEBYTE * pDataType,
            MQEVOID * pData, MQEINT32 nElements, MQEVOID *pBase,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
The null terminated string name of the field.

MQEBYTE * *pDataType* - input and output
The input value is used with *nElements* to specify the size of the data buffer. The output value is the type of the field.

MQEVOID * *pData* - input and output
The destination buffer to receive the copy of the field data. If this parameter is a NULL, then the number of the elements of datatype is returned.

If data type is MQE_TYPE_FIELDS, then a single field object handle MQEHFIELDS is returned.

If data type is MQE_TYPE_UNTYPED, then it is treated as an array of bytes.

MQEINT32 *nElements* - input
Specifies the size of the ***pData*** buffer in the number of elements of the input value of **pDataType*. If *pDataType* is NULL, the default is MQE_TYPE_BYTE. If *pData* is NULL, then this parameter is ignored.

MQEVOID * *pBase* - input
A platform specific base pointer.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

- If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

  MQE_EXCEPT_NOT_FOUND

  MQE_EXCEPT_INVALID_HANDLE

  MQE_EXCEPT_ALLOCATION_FAILED

  MQE_EXCEPT_TYPE
  Field type is incorrect

- If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

  MQE_WARN_FIELDS_DATA_TRUNCATED
  The application has asked for (and been returned) less data than is available in the field.

Return Value

MQEINT32

- On success, returns the number of elements.
- On failure, returns "-1".

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  datatype;
MQEINT32  n;
MQEBYTE * pdata;
MQEBYTE * buf;
MQEINT32  rc;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/*
 * Add some fields to the fields object... and one of them is "XYZ"
 */
 ...

/* Get the field data length */
n = MQeFieldsGet( hSess, hFlds, "XYZ", &datatype, NULL, 0, NULL,
         &compcode, &reason);

/* Verfify that datatype is correct. */

/* Get some space to put the data */
buf = (MQEBYTE *)calloc(n, MQE_SIZEOF(datatype));

/* Get the field data */
rc = MQeFieldsGet( hSess, hFlds, "XYZ", &datatype, &buf, n, NULL,
         &compcode, &reason);
```

See Also

MQeFieldsPut

MQeFieldsGetArray

Description

Given a field name, retrieves the data type and a portion of an encoded array into a buffer. The output data type is the data type of the initial source array element. All remaining source array elements must be of the same type for this call to complete successfully. Returns the number of elements on success.

If an error occurs, returns the source count of the offending element or "-1".

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsGetArray( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEBYTE * pDataType,
              MQEINT32 sOff, MQEVOID * pDstBuf, MQEINT32 dstLen,
              MQEVOID *pBase, MQEINT32 * pCompCode,
              MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the array. A null or a zero length string is invalid. Field names for each array element are constructed as described in “MQeFields*Array” on page 47.

MQEBYTE * *pDataType* - input and output
The data type for the buffer. The input value is used with *dstLen* to specify the size of the data buffer. The output value is the type of the initial source array element. If this parameter is NULL, MQE_TYPE_BYTE is used as the input value.

MQEINT32 *sOff* - input
The index of the initial source array element. the data to be copied.

MQEVOID * *pDstBuf* - input
The destination buffer to receive the array data. The initial source array element is copied to *pDstBuf[0]* (not *pDstBuf[sOff]*). The size of the buffer is specified by the combination of *dstLen* and the input value of **pDataType*. If this parameter is NULL, then no data is copied.

MQEINT32 *dstLen* - input
Specifies the size of the *pData* buffer in terms of the number of elements of the input value of **pDataType*. If this parameter is less than or equal to 0, no data is copied.

MQEVOID * *pBase* - input
A platform specific base pointer.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values::

MQE_EXCEPT_NOT_FOUND

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_TYPE
The data type of an array element does not match the type of the initial source array element or the number of array elements encoded in *hFlds* is invalid.

MQE_EXCEPT_DATA
The field containing the size of the array contains an invalid value.

MQE_EXCEPT_INVALID_ARGUMENT
sOff is less than "0" or greater than or equal to the number of elements in the source array.

Return Value

MQEINT32

- On success, returns the number of elements in the source array.
- On failure, returns a count of the number of elements processed in the source array including the failing element.
- If an error occurs prior to any elements being processed, "-1" is returned.

Example

```
#include <hmq.h>;
static     MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static     MQEINT32 intBuf[4]  = { 0x12345678, 0xDEADBEEF, 0xC0D1F1ED, 0x1DEC0DED};
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    datatype;
MQEINT32   n;
MQEINT32   rc;
MQEINT32 * buf;
#define NULL 0

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/*
* Add some fields to the fields object... and one of them is the array "XYZ"
*/
MQeFieldsPutArray( hSess, hFlds, "XYZ", MQE_TYPE_INT, intBuf, 4, &compcode, &reason);

/* Get the field data length and datatype */
n = MQeFieldsGetArray( hSess, hFlds, "XYZ", &datatype, 0, NULL, 0, NULL, &compcode, &reasc

/* Get some space to put the data */
buf = malloc( n * MQE_SIZEOF(datatype));

/* Get the field data */
rc = MQeFieldsGetArray( hSess, hFlds, "XYZ", &datatype, 0, buf, n, NULL, &compcode, &reasc
```

See Also

MQeFieldsPut

MQeFieldsGetByArrayOfFd

Description

Get fields data and data lengths from an MQeFields object for the names specified in the array of field descriptors. For each descriptor, both the field name and datatype must match to be successful.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsGetByArrayOfFd( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQEFIELD pFds[], MQEINT32 nFds,
                MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

(MQEFIELD *) *pFds* - input/output
An array of `MQeField_st` data structures. For each descriptor, the size destination buffer is determined by the input values of *fd_datatype* and *fd_datalen*. For successful descriptors, the output value of *fd_datalen* is set to the number of elements (not bytes) of the specified field.

The length of a field name is determined from the `fd_name` field, the `fd_namelen` field is ignored.

MQEINT32 *nFds* - input
Number of fields in the *pFds* array.

MQEINT32 * *pCompCode* - output
`MQECC_OK`, `MQECC_WARNING` or `MQECC_ERROR`. The error for the corresponding index to fail.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals `MQECC_ERROR`, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_TYPE
The field for a descriptor did not match the datatype.

MQE_EXCEPT_NOT_FOUND
No field was found for a descriptor.

Return Value

MQEINT32

- On success, returns the number of descriptors successfully updated .
- On failure, returns a count of the number of descriptors processed, including the failing descriptor.
- If an error occurs prior to any descriptors being processed, "-1" is returned.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR textVal[] = "The Owl and the Pussy Cat went to sea.";
/* template for fields */
static const MQEFIELD PFDS[] = {
    {MQE_TYPE_BYTE, 0, 7, "fooByte", (MQEBYTE *)0, 0, (MQEBYTE *)0},
    {MQE_TYPE_SHORT, 0, 8, "fooShort", (MQEBYTE *)0, 0, (MQEBYTE *)0},
    {MQE_TYPE_LONG, 0, 7, "fooLong", (MQEBYTE *)0, 0, (MQEBYTE *)0},
    {MQE_TYPE_ASCII, 0, 7, "fooText", (MQEBYTE *)0, 0, (MQEBYTE *)0},
    };
#define NFDS (sizeof(PFDS)/sizeof(PFDS[0]))
MQEHSESS  hSess;
MQEINT32  compcode;
MQEFIELD  Fds[NFDS];
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  byteVal;
MQEINT16  int16Val;
MQEINT32  int32Val;
MQEBYTE  datatype;
MQEINT32  rc;
MQEINT32  nFlds,i;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);


/* Put some fields in the fields object using MQeFieldsPutByArrayOfFd() */
byteVal = 0xAE;
int16Val = 0x9876;
int32Val = 0x12345678;

/* Copy template */
memcpy(Fds,PFDS,sizeof(Fds));

Fds[0].fd_data =
Fds[0].fd_datalen = 1;
Fds[1].fd_data =
Fds[1].fd_datalen = 1;
Fds[2].fd_data =
Fds[2].fd_datalen = 1;
Fds[3].fd_data = [0];
Fds[3].fd_datalen = sizeof(textVal);

compcode = MQECC_OK, reason = 0;
MQFieldsPutByArrayOfFd( hSess, hFlds, Fds, NFDS , &compcode, &reason);

/* Copy template */
memcpy(Fds,PFDS,sizeof(Fds));

/* Get data lengths */
rc = MQeFieldsGetByArrayOfFd( hSess, hFlds, Fds, NFDS, &compcode, &reason);

/* Get space for each field data */
for( i=0; i<rc; i++) {
 int len = Fds[i].fd_datalen*MQE_SIZEOF(Fds[i].fd_datatype);
 if (len > 0) {
  Fds[i].fd_data = (MQEBYTE *) malloc(len);
 }
}

/* Get all the fields defined in field descriptor array in one shot */
compcode = MQECC_OK, reason = 0;
MQFieldsGetByArrayOfFd( hSess, hFlds, Fds, NFDS, &compcode, &reason);
```

See Also

MQeFieldsPutByArrayOfFd,

MQeFieldsGetByIndex

Description

Copies information about some fields in an MQeFields object into an array of descriptors. This call is used to discover information about the fields in an MQeFields object without providing the field names. This is useful if the contents of the MQeFields object are fully defined. If an MQeFields object has `N` fields indexed from `0` to `N-1` , then **MQeFieldsGetByIndex** returns information about the *`nFlds`* starting at index `startIndex`. The indices of the individual fields are guaranteed to stay the same for successive calls to **MQeFieldsGetByIndex** only as long as there are no other intervening operations on the MQeFields object.

`Index 0` is special, it is a field with an empty name (*`fd_namelen`* is "0") that contains the encoded type name (`MQE_TYPE_ASCII`) of the MQeFields object. It is provided primarily to support the debugging of communication problems with a peer MQSeries Everyplace system. Programs that are trying to enumerate the fields in an MQeFields object would usually start with `index 1`. The number of fields returned by **MQeFieldsFields** includes this special field.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsGetByIndex( MQEHSESS hSess, MQEHFIELDS hFlds,
               MQEINT32 startIndex, MQEFIELD pFds[],
               MQEINT32 nFlds, MQEINT32 * pCompCode,
               MQEINT32 * pReason);
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQEINT32 *startIndex* - input
The starting index field to begin processing the descriptors.

(MQEFIELD) *pFds* - input and output
An array of fields descriptors data structures. The input values of each descriptor determines how much information is copied on output:

Name Is copied into *`fd_name[0..fd_namelen]`*, if *`fd_name`* is not `NULL`.

Data Is copied into the (byte) buffer *`fd_data[0..fd_datalen*MQE_SIZEOF(fd_datatype)]`* if *`fd_data`* is not `NULL`. An integral number of the field's data type elements are copied. The input values of *`fd_namelen`* , *`fd_datatype`* and *`fd_datalen`* are used, not the field's actual datatype and length values.

On output, each descriptor is modified to reflect the field's actual values:

fd_datatype
is set to the field's datatype.

fd_namelen
is set to the length of the field's name.

fd_datalen
is set to the number of elements in the field (of the field's datatype, not the input datatype).

MQEINT32 *nFlds* - input
The number of fields to copy, (the number of elements in *pFds*).

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_ARGUMENT
An index greater than the number of fields, *startIndex*<= 0 , *nFlds*<= 0, or *pFlds* is NULL.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of descriptors successfully updated .
- On failure, returns a count of the number of descriptors processed, including the failing descriptor.
- If an error occurs prior to any descriptors being processed, "-1" is returned.

Example

```
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
   static  const MQECHAR * textVal  = "The Owl and the Pussy Cat went to sea.";
   /* template for fields */
   static  const MQEFIELD PFDS[] = {
           {MQE_TYPE_BYTE, 0, 7, "fooByte", (MQEBYTE *)0, 0, (MQEBYTE *)0},
           {MQE_TYPE_SHORT, 0, 8, "fooShort", (MQEBYTE *)0, 0, (MQEBYTE *)0},
           {MQE_TYPE_LONG, 0, 7, "fooLong", (MQEBYTE *)0, 0, (MQEBYTE *)0},
           {MQE_TYPE_ASCII, 0, 7, "fooText", (MQEBYTE *)0, 0, (MQEBYTE *)0},
           };
   #define NFDS 4
   MQEHSESS   hSess;
   MQEINT32   compcode;
   MQEINT32   reason;
   MQEHFIELDS hFlds;
   MQEBYTE    byteVal;
   MQEINT16   int16Val;
   MQEINT32   int32Val;
   MQEFIELD   Fds[NFDS], fd;
   MQEINT32   rc, nFlds, i;

   hSess = MQeInitialize("MyAppsName", &compcode, &reason);
   hFlds = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

   /* Put some fields in the fields object using MQeFieldsPutByArrayOfFd() */
   byteVal  = 0xAE;
   int16Val = 0x9876;
   int32Val = 0x12345678;

   /* Copy template */
   memcpy(Fds,PFDS,sizeof(Fds));

   Fds[0].fd_data = &byteVal;
```

```
Fds[0].fd_datalen = 1;
Fds[1].fd_data = &int16Val;
Fds[1].fd_datalen = 2;
Fds[2].fd_data = &int32Val;
Fds[2].fd_datalen = 4;
Fds[3].fd_data = (void *) &textVal[0];
Fds[3].fd_datalen = strlen(textVal);

rc = MQeFieldsPutByArrayOfFd( hSess, hFlds, Fds, NFDS, &compcode, &reason);

/* Get the fields out by index*/
nFlds     = MQeFieldsFields( hSess, hFlds, &compcode, &reason);

/* Get the fields one by one (without knowing the field names) */
/* Start at 1 - ignore index 0 (field object identifier)       */
for (i=1; i<nFlds; i++)
{

   fd.fd_name = NULL;
   fd.fd_namelen = 0;
   fd.fd_datatype = MQE_TYPE_BYTE;
   fd.fd_data = NULL;
   fd.fd_datalen = 0;
   fd.fd_base = 0;

   /* Use get by index to get datatype, namelen and datalen */
   MQeFieldsGetByIndex(hSess, hFlds, i, &fd, 1, &compcode, &reason);

   /* Allocate space for the field name */
   fd.fd_name = malloc( fd.fd_namelen+1 );

   /* Allocate space for the data */
   fd.fd_data = malloc( fd.fd_datalen * MQE_SIZEOF(fd.fd_datatype));

   /* Get all the data and the name, now we have allocated space */
   MQeFieldsGetByIndex(hSess, hFlds, i, &fd, 1, &compcode, &reason);

   /* Null terminate the name */
   fd.fd_name[fd.fd_namelen] = '\0';

   free( fd.fd_data );
   free( fd.fd_name );
}
```

See Also

- **MQeFieldsFields**
- **MQeFieldsGet**

MQeFieldsGetByStruct

Description

Copy one or more fields from an MQeFields object directly into a data structure.

Given a pointer to a user data structure and its corresponding struct descriptors, this API gets all the MQeFields data into the data structure. Processing stops as soon as a descriptor fails or when all descriptors are extracted. This API is similar to **MQeFieldsGetByArrayOfFd**, as a match is only successful if both the field name and data type match the input descriptor. It differs by constructing the data buffers for the various fields from a single pointer value, as appropriate when extracting fields into a data structure. The platform specific base pointer is not available with this call (treated as NULL).

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsGetByStruct( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQEVOID * pStruct,
                (struct MQFieldStructDescriptor_st) pfsd[],
                MQEINT32 nSds, MQEINT32 *
                pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

PMQEVOID *pStruct* - output
A pointer to the target data structure.

(struct MQFieldStructDescriptor_st *) *pfsd* - input and output
A definition that defines the relation between the elements in the *pStruct* and the fields in the MQeFields object. On output, if the input value of the *sd_flags*parameter has the MQSTRUCT_LEN bit set, then *sd_datalen* is updated to contain the number of elements in the field.

MQEINT32 *nSds* - input/output
The number of elements in the array pointed to by *pfsd*

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
One or more fields needed to populate the data structure may be missing .

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of descriptors successfully updated .
- On failure, returns a count of the number of descriptors processed, including the failing descriptor.
- If an error occurs prior to any descriptors being processed, "-1" is returned.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
struct myData_st
{
   MQEINT32 x;           /* simple variable */
   MQECHAR *name  ;      /* pointer to name buffer */
   MQEINT32 namelen;     /* length of name */
   MQEBYTE  buf[8];      /* fixed buffer in struct */
   MQEINT32 fieldlen;    /* length of a field, buffer not in struct */
};

MQEINT32 field[10];      /* buffer whose length is in a structure */

/* A possible sample definition of MQEFIELDDESC for myData_st       */
static MQEFIELDDESC myDataStruct_fd[] = {

   {"x", 1, MQE_TYPE_INT, 0, 0, 1},

   {"name", 4, MQE_TYPE_ASCII, MQSTRUCT_LEN|MQSTRUCT_DATA, 4, 64},

   {"buf",  3, MQE_TYPE_BYTE, 0, 12, 8},

   {"field",5, MQE_TYPE_INT, MQSTRUCT_LEN|MQSTRUCT_NODATA, 20, 0}
};

static  MQECHAR * textBuf  = "The Owl and the Pussy Cat went to sea.";
static  MQEBYTE   byteBuf[] = { 0xAB, 0xCD, 0x12, 0x44};
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
struct myData_st myData;
MQEINT32  int32Val;
MQEINT32  rc;

for (rc=0; rc<sizeof(field)/sizeof(field[0]); rc++) field[rc]=rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put some fields into the fields object. */
int32Val = 0xABABBABA;
rc = MQeFieldsPut( hSess, hFlds, "x", MQE_TYPE_INT, &int32Val, 1, &compcode, &reason);

rc = MQeFieldsPut( hSess, hFlds, "name", MQE_TYPE_ASCII, textBuf, strlen(textBuf), &compcode

rc = MQeFieldsPut( hSess, hFlds, "buf", MQE_TYPE_BYTE, byteBuf, 4, &compcode, &reason);

rc = MQeFieldsPut( hSess, hFlds, "field", MQE_TYPE_INT, field, 10, &compcode, &reason);

/* Retrieve all the fields out at once and populate the user data structure. */
rc = MQeFieldsGetByStruct( hSess, hFlds, &myData, myDataStruct_fd, 4, &compcode, &reason);
```

See Also

MQeFieldsPutByStruct

MQeFieldsHide

Description

Excludes this field from the field comparison API, **MQeFieldsEquals**. Each field has a hide bit (initially "0") associated with it. This API allows the application to set or clear the hide bit. The hide bit is considered part of the value of a field, it is cleared if a field with the same name is put into the MQeFields object. The value of the hide bit is exported when the MQeFields object is serialized with **MQeFieldsDump**, so hidden fields remain hidden when MQeFields objects are transported to a different MQSeries Everyplace system.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsHide( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQECHAR * pName, MQEINT32 hide,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by MQeInitialize.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQEINT32 *hide* - input

"0" Clears field element's hide bit, rendering it eligible for comparison by **MQeFieldsEquals**.

nonzero
Sets field element's hide bit, rendering it ineligible for comparison by **MQeFieldsEquals**.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
The specified field is not present in the MQeFields object.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32
Returns "0" on success, or "-1" on failure.

Example

See example in **MQeFieldsEquals**.

See Also

MQeFieldsEquals.

MQeFieldsPut

Description

Puts a field into the MQeFields object.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsPut( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQECHAR * pName, MQEBYTE DataType,
            MQEVOID * pData, MQEINT32 nElements,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQEBYTE *DataType - input*
The data type of the field data. This parameter cannot be a null and its value must be one of the defined values. See “Field data types” on page 54.

MQEVOID * *pData* - input
The data buffer. If NULL, an internal buffer is allocated whose size is specified by the *nElements* parameter. You can then use **MQeFieldsWrite** to put data into this pre-allocated buffer. And this internal buffer is initialized to zeros for the data types,

- MQE_TYPE_BYTE
- MQE_TYPE_SHORT
- MQE_TYPE_INT
- MQE_TYPE_LONG
- MQE_TYPE_ASCII
- MQE_TYPE_UNICODE
- MQE_TYPE_UNTYPED
- MQE_TYPE_FLOAT
- MQE_TYPE_DOUBLE

If *DataType* is MQE_TYPE_FIELDS, *pData* must not be null.

MQEINT32 *nElements* - input
The number of elements of type *DataType* in *pData*. This must be greater than "0". If the *DataType* is MQE_TYPE_FIELDS, MQE_TYPE_ARRAY_ELEMENTS, or MQE_TYPE_BOOLEAN, *nElements* must "1".

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If MQECC_ERROR, **pReason* could be:

MQE_EXCEPT_INVALID_HANDLE
If either *hSess* or *hFlds* are invalid handles.

MQE_EXCEPT_INVALID_ARGUMENT
If an invalid argument is used.

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32
Returns "0" on success, or "-1" on failure.

Valid input parameter combinations

pName	DataType	DataLen	Data	Comment
! null	*	>0	! null	Normal usage
! null	*	>0	null	Preallocate a field data.
null	*	*	*	Error

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEHFIELDS hFlds;
MQEBYTE  datatype;
MQEINT32  n;
MQEINT32  data;
MQEINT32  compcode;
MQEINT32  reason;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put a 4-bytes integer into the fields object. */
datatype = MQE_TYPE_INT;
n    = 1;
data   = 0x12345678;
rc = MQeFieldsPut( hSess, hFlds, "MyData", datatype, (MQEBYTE *) &data, n,
         &compcode, &reason);
```

See Also

MQeFieldsGet

MQeFieldsPutArray

Description

Given a name, put an array as individual fields with the field names derived from the name.

Syntax

```
#include <hmq.h>
MQEVOID MQeFieldsPutArray( MQEHSESS hSess, MQEHFLDS hFlds,
              MQECHAR* pName, MQEBYTE DataType, MQEVOID * pData,
              MQEINT32 nElements, MQEINT32 * pCompCode,
              MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQEBYTE *DataType* - input
The data type of the field data. See "Field data types" on page 54. This may not be MQE_TYPE_ASCII or MQE_TYPE_UNICODE as the length of each ascii or unicode string in the array is required. Use **MQeFieldsPutAsciiArray** or **MQeFieldsPutUnicodeArray** for these field types.

MQEVOID * *pData* - input
A data buffer whose size is determined from *DataType* and *nElements* .

MQEINT32 *nElements* - input
Number of elements of type DataType in pData . This must be greater than or equal to 0.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE
If either *hSess* or *hFlds* are invalid handles.

MQE_EXCEPT_INVALID_ARGUMENT
If an invalid argument is used.

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEVOID

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEHFIELDS hFlds;
MQEBYTE  datatype;
MQEINT32  n = 5;
```

```
MQEINT32  data[5];
MQEINT32  compcode;
MQEINT32  reason;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put an array of 32 bit integers into the fields object. */
datatype = MQE_TYPE_INT;
data[0]   = 0x12345678;
data[1]   = 0xFEEDBABE;
data[2]   = 0xCAFEBABE;
data[3]   = 0xCOD1F1ED;
data[4]   = 0x1DEC0DED;
MQeFieldsPutArray( hSess, hFlds, "MyData", datatype, (MQEBYTE *)
         &data, n, &compcode, &reason);
```

See Also

MQeFieldsGetArray

MQeFieldsPutByArrayOfFd

Description

Creates a set of fields in the MQeFields object given an array of field descriptors. Returns the number of successfully processed descriptors, or "-1" if an error occurred before any descriptors were processed. Descriptors are processed in order and the call fails as soon as the first descriptor fails.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsPutByArrayOfFd( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECONST MQEFIELD pFds[], MQEINT32 nFds,
                MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
A session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQEFIELD * *pFds* - input
An array of `struct MQeField_st` field descriptors. Puts a field named *fd_name* into an MQeFields object with *fd_datalen* elements of type *fd_datatype*. The field data is taken from *fd_data* if it is not `NULL`, otherwise, the field data is set to zero. The *fd_namelen* field is not used by this call. The field name's length is determined from *fd_name* .

MQEINT32 *nFds* - input
Number of descriptors in the *pFds* array.

MQEINT32 * *pCompCode* - output
`MQECC_OK`, `MQECC_WARNING` or `MQECC_ERROR`.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals `MQECC_ERROR`, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of descriptors successfully put.
- On failure, returns a count of the number of descriptors processed including the failing descriptor.
- If an error occurs prior to any descriptors being processed, -1 is returned.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textVal = "The Owl and the Pussy Cat went to sea.";
static const MQEFIELD PFDS[] = {
          {MQE_TYPE_BYTE, 0, 0, "fooByte", 0, 0, (MQEBYTE *)0},
          {MQE_TYPE_SHORT, 0, 0, "fooShort", 0, 0, (MQEBYTE *)0},
          {MQE_TYPE_LONG, 0, 0, "fooLong", 0, 0, (MQEBYTE *)0},
          {MQE_TYPE_ASCII, 0, 0, "fooText", 0, 0, (MQEBYTE *)0}
    };
MQEHSESS  hSess;
```

```
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  byteVal;
MQEINT16  int16Val;
MQEINT32  int32Val, pDataLen[2], *pDataLen2;
MQEVOID * ppData[4], ** ppData2, **ppData3;
MQEINT32  rc, nFlds;
MQEINT32  i;
MQEBYTE  datatype;
MQEFIELD * pFds;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);


/* Put some fields in the fields object using MQeFieldsPutByArrayOfFd() */
byteVal = 0xAE;
int16Val = 0x9876;
int32Val = 0x12345678;
PFDS[0].fd_fd_datalen = sizeof(byteVal);
PFDS[1].fd_fd_datalen = sizeof(int16Val);
PFDS[2].fd_fd_datalen = sizeof(int32Val);
PFDS[3].fd_fd_datalen = strlen(textVal);
PFDS [0].fd_data = (MQEVOID *)&byteVal;
PFDS [1].fd_data = (MQEVOID *)&int16Val;
PFDS [2].fd_data = (MQEVOID *)&int32Val;
PFDS [3].fd_data = (MQEVOID *) textVal;

MQFieldsPutByArrayOfFd( hSess, hFlds, PFDS, 4, &compcode, &reason);

/* Get the field lengths, not data */
for (i=0;i<4;i++) {
 PFDS[i].fd_fd_datalen = 0;
 PFDS[i].fd_data = (MQEVOID *)0;
}
nFlds = MQFieldsGetByArrayOfFd( hSess, hFlds, PFDS, 4, &compcode, &reason);

if (nFlds > 0) {
 /* Get space for field data */
 for( i=0; i<nFlds; i++) {
   PFDS[i].fd_data = (MQEVOID *)
   malloc(PFDS[i].fd_datalen*mqe_sizeof(PFDS[i].fd_datatype));
 }
 /* Get all the fields defined in field descriptor array in one shot */
 nFlds = MQFieldsGetByArrayOfFd( hSess, hFlds, PFDS, nFds, &compcode, &reason);
}
```

See Also

MQeFieldsGetByArrayOfFd,

MQeFieldsPutByStruct

Description

Given a pointer to a user data structure and an array of structure descriptors, this API puts all the elements in the data structure that are identified by the structure descriptors into the MQeFields object. Returns the number of descriptors successfully processed, or "-1", if an error occurred before any descriptor was processed.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsPutByStruct( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQEVOID * pStruct,
                struct MQeFieldStructDescriptor pfsd[],
                MQEINT32 nSds, MQEINT32 * pCompCode, MQEINT32 *
                pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

PMQEVOID *pStruct* - input
A pointer to the data structure that is the source of the data.

struct MQeFieldStructDescriptor * *pfsd* **- input**
A definition that defines the relation between the elements in the *pStruct* and the fields in the MQeFields object.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of fields put successfully.
- On failure, returns a count of the number of descriptors processed including the failing descriptor.
- If an error occurs prior to any descriptors being processed, "-1" is returned.

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
struct myData_st
{
   MQEINT32 x;           /* simple variable */
   MQECHAR *name  ;      /* pointer to name buffer */
   MQEINT32 namelen;     /* length of name */
   MQEBYTE  buf[8];      /* fixed buffer in struct */
   MQEINT32 fieldlen;    /* length of a field, buffer not in struct */
```

```
};

MQEINT32 field[10];      /* buffer whose length is in a structure */


/* A possible sample definition of MQEFIELDDESC for myData_st       */
static MQEFIELDDESC myDataStruct_fd[] = {
   {"x", 1, MQE_TYPE_INT, 0, 0, 1},
   {"name", 4, MQE_TYPE_ASCII, MQSTRUCT_LEN|MQSTRUCT_DATA, 4, 64},
   {"buf",  3, MQE_TYPE_BYTE, 0, 12, 8},
   {"field",5, MQE_TYPE_INT, MQSTRUCT_LEN|MQSTRUCT_NODATA, 20, 0}
};

static  MQECHAR * textVal  = "The Owl and the Pussy Cat went to sea.";
static  MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44};
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
struct myData_st myData;
MQEINT32  int32Val;
MQEINT32  rc;

/* Initialize data */
myData.x = 20;
myData.name = textVal;
myData.namelen = strlen(textVal);
myData.fieldlen = 10;
for (rc=0; rc<4; rc++) myData.buf[rc] = textVal[rc];
for (rc=0; rc<sizeof(myData.buf); rc++) myData.buf[rc] = 0;
for (rc=0; rc<myData.fieldlen; rc++) field[rc] = rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put the data structure into the fields object. */
rc = MQeFieldsPutByStruct( hSess, hFlds, &myData , myDataStruct_fd, 4, &compcode, &reason)
/* Add "field" whose length is in myData.fieldlen */
rc = MQeFieldsPut( hSess, hFlds, "field", MQE_TYPE_INT, &field, myData.fieldlen, &compcode
```

See Also

MQeFieldsGetByStruct

MQeFieldsRead

Description

Reads a portion of a field's data block. Returns the number of elements read, or "-1" if an error occurred.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsRead( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
            MQEBYTE DataType, MQEVOID * pDestBuf, MQEINT32 srcOff,
            MQEINT32 srcLen, MQEVOID * pBase, MQEINT32 * pCompCode,
            MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQEBYTE *DataType* - input
The data type of the named field. It must match the data type of the field in the MQeFields object. The value MQE_TYPE_FIELDS in not a valid argument.

MQEBYTE * *pDestBuf* - output
The destination buffer for the read operation

MQEINT32 *srcOff* - input
The offset position into the field data to start the read.

MQEINT32 *srcLen* - input
Number of bytes to read

MQEVOID * *pBase* - input
The base pointer for the destination buffer *pDestBuf* .

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_ARGUMENT
Invalid inputs, for example, *pDestBuf* is a NULL.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_NOT_FOUND
The named field is not in the MQeFields object.

MQE_EXCEPT_TYPE
The type of the named field does not match *DataType* .

MQE_EXCEPT_DATA
The field data is not suitable for reading, (for example too short or null)

MQE_EXCEPT_EOF
The *srcOff* starts beyond the end of the field's data block.

Return Value

MQEINT32

- On success, returns the number of elements read successfully.
- On failure, returns "-1".

Example

```
#include <hmq.h>

static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEHFIELDS hFlds;
MQEINT32  compcode;
MQEINT32  reason;
MQEINT32  i, nread;
MQECHAR  buf[64];
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Allocate a 128 byte buffer field */
rc = MQeFieldsPut( hSess, hFlds, "y" , MQE_TYPE_BYTE, NULL , 128,
         &compcode, &reason);

/* Fill the buffer with values 0-127 */
for (i=0; i<128; i++) {
  char c=i;
  MQeFieldsWrite( hSess, hFlds, "y" , i, &c, 1, &compcode, &reason);
}

/* Read 64 byte out into an output buf, nread = 64 */
nread = MQeFieldsRead( hSess, hFlds, "y", MQE_TYPE_BYTE, buf, 0, 64, NULL,
            &compcode, &reason);
```

See Also

- **MQeFieldsWrite**
- **MQeFieldsPut**

MQeFieldsRestore

Description

An MQeFields object can be restored from a logical byte array to an MQeFields handle using a sequence of **MQeFieldsRestore** calls. Each individual call does a partial restore of the MQeFields object, specifying the next subarray of the logical byte array. This allows a large MQeFields object to be restored using a smaller buffer. The first call specifies the total length of logical byte array as well as the first partial restore length. The MQeFields handle maintains some restore state in between partial restore calls. It returns the number of bytes consumed by this partial restore.

If the MQeFields handle has a type initially, then the type of the restored MQeFields object must match it, or an error occurs. If not, then the type of the MQeFields handle is set to the type of the restored MQeFields object.

If an error occurs during one of the partial restores, the MQeFields object's internal restore state enters an invalid state, and no further updates are made to the MQeFields handle. The remaining calls should be made with valid arguments (except that the content of the data buffer is ignored), in order to return the MQeFields handle to an inactive restore state. A partially restored field handle (the restore aborted with only some of the fields added) reverts to an inactive state if any other MQeFields operations use the MQeFields handle.

Syntax

```
#include <hmq.h>
MQINT32 MQeFieldsRestore( MQEHSESS hSess, MQEHFIELDS hFlds,
             MQEINT32 dumpLen, MQEBYTE data[], MQEINT32 dataLen,
             MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
The field object handle that is being restored. An MQE_HANDLE_NULL handle is a invalid input. This field object handle should be allocated by **MQeFieldsAlloc** with "" as the type input parameter to restore an arbitrary MQeFields object.

MQEINT32 *dumpLen* - input
The total dump length of the MQeFields object. This parameter is only used on the first partial restore, although it is recommended that subsequent calls use the same original value.

MQEBYTE *data[]* - input
The data byte array from which to perform a partial restore of the MQeFields object.

MQEINT32 *dataLen* - input
The number of bytes to restore. This is the length of the current partial restore.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE
The fields object handle is invalid.

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_DATA
The byte array could be corrupted. The restore operation could not reconstruct the MQeFields object.

Return Value

MQINT32

- On success, returns the number of bytes restored.
- On failure, returns "-1".

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
struct myData_st
{
   MQEINT32 x;           /* simple variable */
   MQECHAR *name  ;      /* pointer to name buffer */
   MQEINT32 namelen;     /* length of name */
   MQEBYTE  buf[8];      /* fixed buffer in struct */
   MQEINT32 fieldlen;    /* length of a field, buffer not in struct */
};

MQEINT32 field[10];      /* buffer whose length is in a structure */


/* A possible sample definition of MQEFIELDDESC for myData_st       */
static MQEFIELDDESC myDataStruct_fd[] = {
   {"x", 1, MQE_TYPE_INT, 0, 0, 1},
   {"name", 4, MQE_TYPE_ASCII, MQSTRUCT_LEN|MQSTRUCT_DATA, 4, 64},
   {"buf",  3, MQE_TYPE_BYTE, 0, 12, 8},
   {"field",5, MQE_TYPE_INT, MQSTRUCT_LEN|MQSTRUCT_NODATA, 20, 0}
};

static  MQECHAR * textVal  = "The Owl and the Pussy Cat went to sea.";
static  MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44};
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
struct myData_st myData;
MQEINT32  int32Val;
MQEINT32  rc;

/* Initialize data */
myData.x = 20;
myData.name = textVal;
myData.namelen = strlen(textVal);
myData.fieldlen = 10;
for (rc=0; rc<4; rc++) myData.buf[rc] = textVal[rc];
for (rc=0; rc<sizeof(myData.buf); rc++) myData.buf[rc] = 0;
for (rc=0; rc<myData.fieldlen; rc++) field[rc] = rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put the data structure into the fields object. */
rc = MQeFieldsPutByStruct( hSess, hFlds, &myData , myDataStruct_fd, 4, &compcode, &reason)
/* Add "field" whose length is in myData.fieldlen */
rc = MQeFieldsPut( hSess, hFlds, "field", MQE_TYPE_INT, &field, myData.fieldlen, &compcode
```

MQeFieldsRestore

See Also

- **MQeFieldsDump**
- **MQeFieldsDumpLength**

MQeFieldsType

Description

Determines the string name of an MQeFields object. Returns the length of the name (not including the terminating NULL) on success. Returns "0" on error.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsType( MQEHSESS hSess, MQEHFIELDS hFlds,
            MQECHAR * pTypeName, MQEINT32 typeLen,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pTypeName* input and output
The output buffer that the MQeFields object type string name is to be copied into. If NULL, no data is returned.

MQEINT32 *typeLen* - input
The size of the *pTypeName* buffer in MQECHAR format. If *pTypeName* is a NULL, this parameter is ignored.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- On success, returns the length of the type name (not including the terminating NULL).
- On failure, returns "-1".

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQECHAR * pname;
MQEINT32  datalen, rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Get the length of object type name */
datalen = MQeFieldsType( hSess, hFlds, 0, NULL, &compcode, &reason);
pname  = (MQECHAR *) malloc(datalen+1);
/* Get the object type name */
rc   = MQeFieldsType( hSess, hFlds, pname, datalen, &compcode, &reason);
```

MQeFieldsWrite

Description

Writes into the data block of an existing field in an MQeFields object. Returns the number of elements written, or "-1" if an error occurs.

Syntax

```
#include <hmq.h>
MQEINT32 MQeFieldsWrite( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
            MQEBYTE DataType, MQEINT32 dstOffset,
            MQEBYTE * pSrcBuf, MQEINT32 srcLen,
            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS hSess - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
The field name. A null or a zero length string is invalid.

MQEINT32 *DataType* - input
The data type of field. The types MQE_TYPE_FIELDS and MQE_TYPE_BOOLEAN are invalid.

MQEINT32 * *dstOffset* - input
The offset into the field data to start the write

MQEBYTE * *pSrcBuf* - input
The source buffer

MQEINT32 *srcLen* - input
The number of elements of type *DataType* to write.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_ARGUMENT
Invalid inputs. For example, *pSrcBuf* is a NULL

MQE_EXCEPT_TYPE
DataType does not match data type of the field.

MQE_EXCEPT_NOT_FOUND
No field with name *pName* found in *hFlds*

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_EOF
End of field reached.

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of elements written.
- On failure, returns "-1".

Example

```
#include <hmq.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEHFIELDS hFlds;
MQEINT32  compcode;
MQEINT32  reason;
MQEINT32  i, nread;
MQECHAR  buf[64];
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Allocate a 128 byte buffer field */
rc = MQeFieldsPut( hSess, hFlds, "y" , MQE_TYPE_BYTE, NULL, 128,
         &compcode, &reason);

/* Fill the buffer with values 0-127 */
for (i=0; i<128; i++) {
char c=i;
MQeFieldsWrite( hSess, hFlds, "y" , MQE_TYPE_BYTE, i, &c, 1,
        &compcode, &reason);
}

/* Read 64 byte out into an output buf, nread = 64 */
nread = MQeFieldsRead( hSess, hFlds, "y", MQE_TYPE_BYTES, buf, 0, 64, NULL,
            &compcode, &reason);
```

See Also

- **MQeFieldsRead**
- **MQeFieldsPut**

MQeFieldsContains

Description

Determines whether the MQeFields object contains a specific field.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsContains( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEINT32 * pCompCode,
              MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input

A handle to an MQeFields object.

MQECHAR * *pName* - input

A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- **"1"** the MQeFields object contains the given field
- **"0"** the field is not found.
- **"-1"** failure.

See Also

MQeFieldsGet

MQeFieldsCopy

Description

Copy one or all fields from one MQeFields object to another.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsCopy( MQEHSESS hSess, MQEHFIELDS hSrcFlds,
            MQEHFIELDS hDstFlds, MQEINT32 Option,
            MQECHAR * pName, MQEINT32 * pCompCode,
            MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hSrcFlds* - input

The handle of the source MQeFields object.

MQEHFIELDS *hDstFlds* - input

The handle of the destination MQeFields object.

MQEINT32 *Option* - input

MQE_FIELDS_OPTION_NONE

This is the default option. It copies the specified field from the source MQeFields object to the destination MQeFields object, but does not replace the data if the field is found in the destination MQeFields object.

MQE_FIELDS_OPTION_ALL_FIELDS

If specified, this API copies all fields from the source MQeFields object to the destination MQeFields object.

MQE_FIELDS_OPTION_REPLACE

If specified, this API overwrites any fields in the destination MQeFields object that have the same field name as the field from the source MQeFields object.

MQECHAR * *pName* - input

A null terminated string containing the name of the field. If MQE_FIELDS_OPTION_ALL_FIELDS is specified, then this parameter is ignored. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

Field name too short or too long.

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

Returns "0" on success, or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textVal = "The Owl and the Pussy Cat went to sea";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds1, hFlds2;
MQEINT32  n;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds1 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
rc   = MQeFieldsPut( hSess, hFlds1, "ibm", MQE_TYPE_UNICODE, strlen(textVal)),
            textVal, &compcode, &reason);

MQeFieldsCopy( hSess, hFlds1, hFlds2, MQE_FIELDS_OPTION_ALL_FIELDS, NULL,
       &compcode, &reason);

n    = MQeFieldsDataLen( hSess, hFlds2, "ibm", &compcode, &reason);

pData  = (MQEBYTE *) calloc(n, MQE_SIZEOF(datatype));

/* Copy out the data */
rc   = MQeFieldsGetAscii( hSess, hFlds2, "ibm", pData, n,
              &compcode, &reason);
```

See Also

MQeFieldsGet

MQeFieldsDataLength

Description

Return the number of elements in a field, in units of the field's data type, or "-1" on error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsDataLength(MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- On success, returns the number of elements in the field.
- On failure, returns "-1".

Pseudo-code

```
MQEINT32 MQeFieldsDataLength( hSess, hFlds, pName, pCompCode, pReason) {
  MQEBYTE datatype=0;
  MQEINT32 datalen;
  datalen = MQeFieldsGet( hSess, hFlds, pName, &datatype, NULL, 0, NULL,
              pCompCode, pReason);
  return datalen;
}
```

See Also

MQeFieldsGet

MQeFieldsDataType

Description

Returns the field data type, or "-1" on error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEBYTE MQeFieldsDataType( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input

A handle to an MQeFields object.

MQECHAR * *pName* - input

A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEBYTE

Returns the field data type or "-1" on failure.

See Also

MQeFieldsGet

MQeFieldsGetArrayLength

Description

Gets the number of elements in an encoded array. Returns the number of elements, or "-1" on error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetArrayLength( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT32 * pCompCode,
                MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by MQeInitialize.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_TYPE
The field is not an encoded array.

MQE_EXCEPT_DATA
The field is not a valid encoded array.

Return Value

MQEINT32

- On success, returns the number of elements in the encoded array.
- On failure, returns "-1".

Example

```
#include <hmq.h>
#include <:hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    datatype;
MQEINT32   data[2], n;
MQEINT32 * pData;
MQEINT32   rc;


hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
```

```
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
data[0] = 0x12345678;
data[1] = 0xDEADBEEF;
rc      = MQeFieldsPutIntArray(hSess, hFlds, "foo", data, 2, &compcode, &reason);

/* Get the data length */
n  = MQeFieldsGetArrayLength( hSess, hFlds, "foo", &compcode, &reason );

datatype= MQE_TYPE_INT;
pData   = malloc(n * MQE_SIZEOF(datatype));

/* Copy out the data */
rc      = MQeFieldsGetIntArray( hSess, hFlds, "foo", pData, 0, n, &compcode, &reason );
```

See Also

MQeFieldsPutArrayLength

MQeFieldsGetBoolean, MQeFieldsGetByte, MQeFieldsGetShort, MQeFieldsGetInt, MQeFieldsGetLong, MQeFieldsGetDouble, MQeFieldsGetFloat

Description

Extracts typed data from the MQeFields object as a single 1, 2, 4, or 8-byte integer, float or double.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetBoolean( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEBYTE * pBoolean, MQEINT32 * pCompCode,
              MQEINT32 * pReason)

MQEINT32 MQeFieldsGetByte( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
             MQEBYTE * pByte, MQEINT32 * pCompCode,
             MQEINT32 * pReason)

MQEINT32 MQeFieldsGetShort( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEINT16 * pShort, MQEINT32 * pCompCode,
              MQEINT32 * pReason)

MQEINT32 MQeFieldsGetInt( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
             MQEINT32 * pInt, MQEINT32 * pCompCode,
             MQEINT32 * pReason)

MQEINT32 MQeFieldsGetLong( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
             MQEINT64 * pLong, MQEINT32 * pCompCode,
             MQEINT32 * pReason)

MQEINT32 MQeFieldsGetFloat( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEBYTE * pFloat, MQEINT32 * pCompCode,
              MQEINT32 * pReason)

MQEINT32 MQeFieldsGetDouble( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEBYTE * pDouble, MQEINT32 * pCompCode,
              MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEBYTE * *pBoolean* - output
The returned boolean value.

MQEBYTE * *pByte* - output
The returned byte value.

MQEINT16 * *pShort* - output
The returned short value.

MQEINT32 * *pInt* - output
The returned 4 byte integer value.

MQEINT64 * *pLong* - output
The returned 8 byte integer value.

MQEFLOAT * *pFloat* - output
The returned double value.

MQEDOUBLE * *pDouble* - output
Returned float value.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

- If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

  MQE_EXCEPT_NOT_FOUND
  Field name not found.

  MQE_EXCEPT_TYPE
  Field type is incorrect.

  MQE_EXCEPT_INVALID_HANDLE

  MQE_EXCEPT_INVALID_ARGUMENT

- If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

  MQE_WARN_FIELDS_DATA_TRUNCATED
  The application has asked for (and been returned) less data than is available in the field.

Return Value

MQEINT32
Returns "0" on success or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  booleanVal;
MQEBYTE  byteVal;
MQEINT16  int16Val;
MQEINT32  int32Val;
MQEINT64  int64Val;
MQFLOAT  floatVal;
MQDOUBLE  doubleVal;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

byteVal  = 1;
rc    = MQeFieldsPut( hSess, hFlds, "bool",  MQE_TYPE_BYTE, &booleanVal, 1,
             &compcode, &reason);
byteVal  = 0x45;
rc    = MQeFieldsPut( hSess, hFlds, "b",  MQE_TYPE_BYTE, &byteVal, 1,
             &compcode, &reason);
int16Val  = 32000;
rc    = MQeFieldsPut( hSess, hFlds, "sh", MQE_TYPE_SHORT, &int16Val, 1,
            &compcode, &reason);
int32Val  = 2000000000;
rc    = MQeFieldsPut( hSess, hFlds, "int", MQE_TYPE_INT, &int32Val, 1,
            &compcode, &reason);
```

```
int64Val.hi = 265;
int64Val.lo = 2000000000;
rc    = MQeFieldsPut( hSess, hFlds, "lg", MQE_TYPE_LONG, &int64Val, 1,
            &compcode, &reason);
floatVal  = 2.55;
rc    = MQeFieldsPut( hSess, hFlds, "f",  MQE_TYPE_FLOAT, &floatVal, 1,
            &compcode, &reason);
doubleVal = 2.3413453231e-63;
rc    = MQeFieldsPut( hSess, hFlds, "d",  MQE_TYPE_DOUBLE, &doubleVal, 1,
            &compcode, &reason);

booleanVal = 0;
byteVal  = 0;
int16Val  = 0;
int32Val  = 0;
int64Val.lo  = 0;
int64Val.hi  = 0;
floatVal = 0.0;
aDouble = 0.0;

/* Get the data */
MQeFieldsGetBoolean ( hSess, hFlds, "bool", &booleanVal, &compcode, &reason);
MQeFieldsGetByte ( hSess, hFlds, "b", &byteVal, &compcode, &reason);
MQeFieldsGetShort ( hSess, hFlds, "sh" , &int16Val, &compcode, &reason);
MQeFieldsGetInt  ( hSess, hFlds, "int", &int32Val , &compcode, &reason);
MQeFieldsGetLong ( hSess, hFlds, "lg", &int64Val , &compcode, &reason);
MQeFieldsGetFloat ( hSess, hFlds, "f", &floatVal , &compcode, &reason);
MQeFieldsGetDouble( hSess, hFlds, "d", &doubleVal , &compcode, &reason);
```

See Also

- **MQeFieldsGet**
- **MQeFieldsPutShort**
- **MQeFieldsPutInt**
- **MQeFieldsPutLong**
- **MQeFieldsPutFloat**
- **MQeFieldsPutDouble**

MQeFieldsGetFields

Description

Extracts a nested MQeFields object from an MQeFields handle.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEHFIELDS MQeFieldsGetFields( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT32 * pCompCode,
                MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by MQeInitialize.

MQEHFIELDS *hFlds* - input

A handle to an MQeFields object.

MQECHAR * *pName* - input

A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

- If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

  MQE_EXCEPT_TYPE

  The field was not the correct type.

  MQE_EXCEPT_NOT_FOUND

  Field name not found.

  MQE_EXCEPT_INVALID_HANDLE

- If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

  MQE_WARN_FIELDS_DATA_TRUNCATED

  The application has asked for (and been returned) less data than is available in the field.

Return Value

MQEHFIELDS

- Returns the field object handle of the given field.
- On error returns MQEHANDLE_NULL.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
const char * hello = "Hello World";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds1, hFlds2, hFlds3;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
```

```
hFlds1 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
hFlds2 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put hFlds1 into hFlds2 */
rc   = MQeFieldsPut( hSess, hFlds1, "ibm", MQE_TYPE_ASCII, hello,
           strlen(hello), &compcode, &reason);
rc   = MQeFieldsPutFields( hSess, hFlds2, "ibmFields", hFlds1,
              &compcode, &reason);
/* hFlds1 is no longer valid */

/* Retrieve hFlds1 as hFlds3 from hFlds2 */
hFlds3 = MQeFieldsGetFields( hSess, hFlds2, "ibmFields",
                          &compcode, &reason);

/* Extract the "ibm" field */
datalen = MQeFieldsGet( hSess, hFlds3, "ibm", &datatype, NULL, 0, NULL,
            &compcode, &reason);
pData  = malloc(datalen+1);
datalen = MQeFieldsGet( hSess, hFlds3, "ibm", &datatype, pData, 0, datalen,
            &compcode, &reason);
pData[datalen] = '\0';
printf("Field is %s\n", pData);

/* Free the fields resources */
MQeFieldsFree( hSess, hFlds3, &compcode, &reason);
MQeFieldsFree( hSess, hFlds2, &compcode, &reason);
```

See Also

MQeFieldsPutFields

MQeFieldsGetArrayOfByte, MQeFieldsGetArrayOfShort, MQeFieldsGetArrayOfInt, MQeFieldsGetArrayOfLong, MQeFieldsGetArrayOfFloat, MQeFieldsGetArrayOfDouble

Description

Extracts the data from the MQeFields object as an array of 1, 2, 4, and 8-byte integers, floats and doubles.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetArrayOfByte( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEVOID * pBytes,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsGetArrayOfShort( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEVOID * pShorts,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsGetArrayOfInt( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEVOID * pInts,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsGetArrayOfLong( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEVOID * pLongs,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsGetArrayOfFloat( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEVOID * pFloats,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsGetArrayOfDouble( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEVOID * pDoubles,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEVOID * *pBytes* - output
The returned byte value.

MQEVOID * *pShorts* - output
The returned short value.

MQEINT32 * *pInts* - output
The returned 4 byte integer value.

MQEVOID * *pLongs* - output
The returned 8 byte integer value.

MQEVOID * *pFloats* - output
The returned double value.

MQEVOID * *pDoubles* - output
The returned float value.

MQEINT32 *n* - input
The size of the input buffer, in elements of the corresponding call.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

- If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

  MQE_EXCEPT_NOT_FOUND
  Field name not found.

  MQE_EXCEPT_DATA
  srcOff is out of range.

  MQE_EXCEPT_INVALID_HANDLE

  MQE_EXCEPT_TRUNCATED

- If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

  MQE_WARN_FIELDS_DATA_TRUNCATED
  The application has asked for (and been returned) less data than is available in the field.

Return Value

MQEINT32

- On success, returns the number of elements in the array.
- On failure, returns "-1".

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    bytes[5];
MQEINT16   shorts[2];
MQEINT32   ints[3];
MQEINT64   longs[2];
MQEINT32   rc;

hSess    = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds    = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
bytes[0] = 0x30;
bytes[1] = 0x31;
bytes[2] = 0x32;
bytes[3] = 0x33;
bytes[4] = 0x34;
rc       = MQeFieldsPut( hSess, hFlds, "b",  MQE_TYPE_BYTE, bytes, 5, &compcode, &reason);
shorts[0]= 32000;
shorts[1]= 32020;
rc       = MQeFieldsPut( hSess, hFlds, "sh", MQE_TYPE_SHORT, shorts, 2, &compcode, &reason
ints[0]  = 2000100000;
ints[1]  = 2000020000;
```

```
ints[2]  = 2000003000;
rc       = MQeFieldsPut( hSess, hFlds, "int", MQE_TYPE_INT, ints, 3, &compcode, &reason);
longs[0].hi = 265;
longs[0].lo = 2000000000;
longs[1].hi = 2000000000;
longs[1].lo = 255;
rc       = MQeFieldsPut( hSess, hFlds, "lg", MQE_TYPE_LONG, longs, 2, &compcode, &reason);

memset(bytes , 0, sizeof(bytes));
memset(shorts, 0, sizeof(shorts));
memset(ints  , 0, sizeof(ints));
memset(longs , 0, sizeof(longs));

/* Get the data */
MQeFieldsGetArrayOfByte ( hSess, hFlds, "b"  , bytes  , 5, &compcode, &reason );
MQeFieldsGetArrayOfShort( hSess, hFlds, "sh" , shorts , 2, &compcode, &reason );
MQeFieldsGetArrayOfInt  ( hSess, hFlds, "int", ints   , 3, &compcode, &reason );
MQeFieldsGetArrayOfLong ( hSess, hFlds, "lg" , longs  , 2, &compcode, &reason );
```

See Also

- **MQeFieldsGet**
- **MQeFieldsPutArrayOfByte**
- **MQeFieldsPutArrayOfShort**
- **MQeFieldsPutArrayOfInt**
- **MQeFieldsPutArrayOfLong**
- **MQeFieldsPutArrayOfFloat**
- **MQeFieldsPutArrayOfDouble**,

MQeFieldsGetAscii, MQeFieldsGetUnicode, MQeFieldsGetObject

Description

Extracts an array of MQECHAR, MQEINT16, or MQEBYTE from a single field in the MQeFields object. The extracted arrays are not terminated with an additional NULL, so if the field data is not terminated the extracted string is not. Returns the length of the field data (not the extracted string) as the number of elements (not bytes) or "-1" on error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetAscii( MQEHSESS hSess, MQEHFIELDS hFlds,
             MQECHAR* pName, MQEBYTE* pData,
             MQEINT32 DataLen, MQEINT32*
             pCompCode, MQEINT32* pReason)

MQEINT32 MQeFieldsGetUnicode( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR* pName, MQEINT16 pData,
              MQEINT32 DataLen,
              MQEINT32* pCompCode, MQEINT32* pReason)

MQEINT32 MQeFieldsGetObject( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR* pName,
              MQEBYTE* pData, MQEINT32 DataLen,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEBYTE * *pData* - output
The caller supplied destination buffer to receive the output data.

MQEINT16 * *pData* - output
The caller supplied destination buffer to receive the output data.

MQEINT32 *DataLen* - input
The maximum number of elements to copy

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

- If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

  MQE_EXCEPT_TYPE .
  The field is not type MQE_TYPE_ASCII, MQE_TYPE_UNICODE, or MQE_TYPE_UNTYPED respectively

  MQE_EXCEPT_NOT_FOUND
  Field name not found.

  MQE_EXCEPT_INVALID_HANDLE

  MQE_EXCEPT_INVALID_ARGUMENT

- If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

  MQE_WARN_FIELDS_DATA_TRUNCATED
  The application has asked for (and been returned) less data than is available in the field.

Return Value

MQEINT32

- On success, returns the number of elements in the field.
- On failure, returns "-1".

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textVal = "The Owl and the Pussy Cat went to sea";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT32  n;
MQEBYTE  datatype;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
rc   = MQeFieldsPut( hSess, hFlds, "ibm", MQE_TYPE_ASCII, strlen(textVal)),
           textVal, &compcode, &reason);

/* Get the data length */
n    = MQeFieldsDataLen( hSess, hFlds, "ibm", &compcode, &reason);

datatype= MQE_TYPE_ASCII;
pData  = (MQEBYTE *) calloc(n, MQE_SIZEOF(datatype));

/* Copy out the data */
rc   = MQeFieldsGetAscii( hSess, hFlds, "ibm", pData, n, &compcode, &reason);
```

See Also

- **MQeFieldsPutAscii**
- **MQeFieldsPutUnicode**
- **MQeFieldsPutObject**

MQeFieldsGetShortArray, MQeFieldsGetIntArray, MQeFieldsGetLongArray, MQeFieldsGetFloatArray, MQeFieldsGetDoubleArray

Description

Extracts an encoded array from the MQeFields object as an array of 2, 4, or 8-byte integers, floats, or doubles.

Returns the number of elements successfully extracted, or "-1" on an error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetShortArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEINT16 * pData,
                 MQEINT32 srcOff, MQEINT32 n,
                 MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsGetIntArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT32 * pData,
                MQEINT32 srcOff, MQEINT32 n,
                MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsGetLongArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT64 * pData,
                MQEINT32 srcOff, MQEINT32 n,
                MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsGetFloatArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEFLOAT * pData,
                 MQEINT32 srcOff, MQEINT32 n,
                 MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsGetDoubleArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEDOUBLE * pData,
                 MQEINT32 srcOff, MQEINT32 n,
                 MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQEINT16 * *pData* - output
The returned short value.

MQEINT32 * *pData* - output
The returned 4 byte integer value.

MQEINT64 * *pData* - output
The returned 8 byte integer value.

MQEFLOAT * *pData* - output
The returned double value.

MQEDOUBLE * *pData* - output
The returned float value.

MQEINT32 *srcOff* - input
The starting index for source array.

MQEINT32 *n* - input
The number of elements to get.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_DATA
srcOff is out of range, or invalid array encoding.

MQE_EXCEPT_TYPE
Field element does not match requested type.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- On success, returns the number of elements in the source array.
- On failure, returns a count of the number of elements processed in the source array including the failing element.
- If an error occurs prior to any elements being processed, "-1" is returned.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEINT16   shorts[2];
MQEINT16*  gotShorts;
MQEINT32   ints[3];
MQEINT32*  gotInts;
MQEINT64   longs[2];
MQEINT64*  gotLongs;
MQEINT32   rc;
MQEINT32   length;
#define NULL 0

hSess    = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds    = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

shorts[0]= 32000;
shorts[1]= 32020;
MQeFieldsPutArray( hSess, hFlds, "sh", MQE_TYPE_SHORT, &shorts[0], 2, &compcode, &reason);
ints[0]  = 2000100000;
ints[1]  = 2000020000;
ints[2]  = 2000003000;
MQeFieldsPutArray( hSess, hFlds, "int", MQE_TYPE_INT, &ints[0], 3, &compcode, &reason);
longs[0].hi = 265;
longs[0].lo = 2000000000;
longs[1].hi = 2000000000;
longs[1].lo = 255;
```

```
MQeFieldsPutArray( hSess, hFlds, "lg", MQE_TYPE_LONG, &longs[0], 2, &compcode, &reason);

/* Get the data */
length = MQeFieldsGetShortArray ( hSess, hFlds, "sh", NULL, 0, 0, &compcode, &reason );
gotShorts = malloc(length * MQE_SIZEOF( MQE_TYPE_SHORT ));
MQeFieldsGetShortArray ( hSess, hFlds, "sh", gotShorts, 0, length, &compcode, &reason );

length = MQeFieldsGetIntArray   ( hSess, hFlds, "int", NULL, 0, 0, &compcode, &reason );
gotInts = malloc(length * MQE_SIZEOF( MQE_TYPE_INT ));
MQeFieldsGetIntArray   ( hSess, hFlds, "int", gotInts, 0, length, &compcode, &reason );

length = MQeFieldsGetLongArray  ( hSess, hFlds, "lg", NULL, 0, NULL, &compcode, &reason );
gotLongs = malloc(length * MQE_SIZEOF( MQE_TYPE_LONG ));
MQeFieldsGetLongArray  ( hSess, hFlds, "lg", gotLongs, 0, length, &compcode, &reason );
```

See Also

- **MQeFieldsGetArray**
- **MQeFieldsPutByteArray**
- **MQeFieldsPutShortArray**
- **MQeFieldsPutIntArray**
- **MQeFieldsPutLongArray**
- **MQeFieldsPutFloatArray**
- **MQeFieldsPutLongArray**

MQeFieldsGetAsciiArray, MQeFieldsGetUnicodeArray, MQeFieldsGetByteArray

Description

Extracts an encoded two dimensional array of MQECHAR, MQESHORT, or MQEBYTE. Starting at source index *srcOff*. Extract at most *n* arrays, into each of the buffers provided by *ppData*. The input values in *pDataLen* indicate the size of the buffer. The output values indicate the size of the corresponding string in the field, or "-1" if an error occurred for the string at the corresponding index. Both *ppStr* and *pDataLen* start at base "0" regardless of the value of *srcOff* .

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsGetAsciiArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQECHAR * ppData[],
                MQEINT32 pDataLen[], MQEINT32 srcOff,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)
MQEINT32 MQeFieldsGetUnicodeArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEINT16 * ppData[],
                 MQEINT32 pDataLen[], MQEINT32 srcOff,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)
MQEINT32 MQeFieldsGetByteArray( MQEHSESS hSess, MQEHFIELDS hFlds,
               MQECHAR * pName, MQEBYTE * ppData[],
               MQEINT32 pDataLen[],
               MQEINT32 srcOff, MQEINT32 n,
               MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by MQeInitialize.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQECHAR * *ppData*[] - output
Array of *n* buffers, starting at index "0". If any buffer is NULL, then data is not extracted for that buffer. If NULL, then all buffers are treated as NULL.

MQEINT16 * *ppData*[] - output
Array of *n* buffers, starting at index "0". If any buffer is NULL, then data is not extracted for that buffer. If NULL, then all buffers are treated as NULL.

MQEBYTE * *ppData*[] - output
Array of *n* buffers, starting at index "0". If any buffer is NULL, then data is not extracted for that buffer. If NULL, then all buffers are treated as NULL.

MQEINT32 *pDataLen[]* - input and output
Array of *n* buffer lengths. The input values specify the length of the buffer in MQECHAR. The output values specify the length of the array element in the MQeFields object in MQECHAR. If NULL, then all buffer lengths are considered to be "0".

MQEINT32 *srcOff* - input
The starting source index from which to copy the array elements.

MQEINT32 *n* - input
The number of elements to get. If "0", the number of elements in the field is returned.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
A field was not found.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_TYPE
The data type of an array element does not match the type of the initial source array element, or the number of array elements encoded in *hFlds* is invalid.

MQE_EXCEPT_DATA
The field containing the size of the array contains an invalid value.

MQE_EXCEPT_INVALID_ARGUMENT
scrOff is less than "0" or greater than or equal to the number of elements in the source array.

Return Value

MQEINT32

- On success, returns the number of in the encoded array.
- On failure, returns a count of the number of elements processed in the array including the failing element.
- If an error occurs prior to any elements being processed, "-1" is returned.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textArray[] =
        { "The Owl and the Pussy Cat went to sea",
         "Here we go round the Mulberry bush",
         "Jack and Jill went up the hill" };
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT32  pStrLen[3], n, *pStrLen2;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

pStrLen[0] = strlen(textArray[0]);
pStrLen[1] = strlen(textArray[1]);
```

```
pStrLen[2] = strlen(textArray[2]);
rc = MQeFieldsPutAsciiArray( hSess, hFlds, "ibm", textArray, pStrLen, 3,
             &compcode, &reason);

/* 1. Get number of elements */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", NULL, NULL, 0, 0,
             &compcode, &reason);

/* Get space for array of string length */
pStrLen2 = (MQEINT32 *) malloc(n * sizeof(MQEINT32));
memset(pStrLen2, 0, n * sizeof(MQEINT32));

/* 2. Get array of string length */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", NULL, pStrLen2, 0, n,
             &compcode, &reason);

/* Get space for array of string */
for (i=0; i<n; i++) {
  pStr[i] = (MQECHAR *) malloc(pStrLen[j]+1);
  memset(pStr[i], 0, pStrLen[j]+1);
}

/* 2. Get array of strings */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", pStr, pStrLen2, 0, n,
             &compcode, &reason);
```

See Also

- **MQeFieldsPutAsciiArray**
- **MQeFieldsPutUnicodeArray**
- **MQeFieldsPutByteArray**

MQeFieldsPutArrayLength

Description

Puts the number of elements in an encoded array.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutArrayLength( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT32 nElements,
                MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input

A handle to an MQeFields object.

MQECHAR * *pName* - input

A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 *nElements* - input

The number of array elements

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

Returns "0" on success, or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>

static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS   hSess;
MQEINT32   compcode;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEINT32   n, data0, data1;
MQEINT32 * pData;
MQEINT32   rc;

hSess   = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds   = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Manual construction of an integer array (vector) with two elements) */
data0 = 0x12345678;
data1 = 0xBEEFDEAD;
MQeFieldsPut( hSess, hFlds, "foo:0", MQE_TYPE_INT, &data0, 1, &compcode, &reason);
MQeFieldsPut( hSess, hFlds, "foo:1", MQE_TYPE_INT, &data1, 1, &compcode, &reason);
MQeFieldsPutArrayLength( hSess, hFlds, "foo", 2, &compcode, &reason);

/* Get the data length */
n       = MQeFieldsGetArrayLength( hSess, hFlds, "foo", &compcode, &reason );
```

```
pData   = malloc(n * MQE_SIZEOF(MQE_TYPE_INT));

/* Get back the data */
rc      = MQeFieldsGetIntArray( hSess, hFlds, "foo", pData, 0, n, &compcode, &reason );
```

See Also

MQeFieldsGetArrayLength

MQeFieldsPutBoolean

Description

Puts a boolean value into an MQeFields object.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutBoolean( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEBYTE aBool,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEBYTE *aBool* - input
A boolean value

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32
Returns "0" on success, or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textVal = "The Owl and the Pussy Cat went to sea";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQBOOL   aBool;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
aBool  = 1;
MQeFieldsPutBoolean( hSess, hFlds, "ibm", aBool, &compcode, &reason);
```

See Also

- **MQeFieldsGetBoolean**
- **MQeFieldsPut**

MQeFieldsPutFields

Description

Puts an MQeFields object into another MQeFields object. The MQeFields object that is being put into the other MQeFields object becomes invalid after the API call. An MQeFields object cannot be inserted into itself.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutFields( MQEHSESS hSess, MQEHFIELDS hFlds1,
             MQECHAR * pName, MQEHFIELDS hFlds2,
             MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds1* - input
The MQeFields object that is receiving *hFlds2*.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEHFIELDS *hFlds2* - input
The MQeFields object that is being moved into *hFlds1*. This MQeFields object becomes invalid after this API returns.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT
Invalid *pName* or *hFlds1* is the same as *hFlds2* .

Return Value

MQEINT32
Returns "0" on success or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
const char * hello = "Hello World";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds1, hFlds2, hFlds3;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds1 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
hFlds2 = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

/* Put hFlds1 into hFlds2 */
rc   = MQeFieldsPut( hSess, hFlds1, "ibm", MQE_TYPE_ASCII, hello,
            strlen(hello), &compcode, &reason);
```

```
rc   = MQeFieldsPutFields( hSess, hFlds2, "ibmFields",
                hFlds1, &compcode, &reason);
/* hFlds1 is no longer valid */

/* Retrieve hFlds1 as hFlds3 from hFlds2 */
hFlds3 = MQeFieldsGetFields( hSess, hFlds2, "ibmFields", &compcode,
                &reason);

/* Extract the "ibm" field */
datalen = MQeFieldsGet( hSess, hFlds3, "ibm", &datatype, NULL, 0, NULL,
            &compcode, &reason);
pData  = malloc(datalen+1);
datalen = MQeFieldsGet( hSess, hFlds3, "ibm", &datatype, pData, 0, datalen,
            &compcode, &reason);
pData[datalen] = '\0';
printf("Field is %s\n", pData);

/* Free the fields resources */
MQeFieldsFree( hSess, hFlds3, &compcode, &reason);
MQeFieldsFree( hSess, hFlds2, &compcode, &reason);
```

See Also

MQeFieldsGetFields

MQeFieldsPutByte, MQeFieldsPutShort, MQeFieldsPutInt, MQeFieldsPutLong, MQeFieldsPutFloat, MQeFieldsPutDouble

Description

Puts an 8, 16, 32, or 64 bit integer, float, or double into the MQeFields object.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutByte( MQEHSESS hSess, MQEHFIELDS hFlds,
             MQECHAR * pName, MQEBYTE* aByte,
             MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutShort( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEINT16* int16Val,
              MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutInt( MQEHSESS hSess, MQEHFIELDS hFlds,
             MQECHAR * pName, MQEINT32* anInt,
             MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutLong( MQEHSESS hSess, MQEHFIELDS hFlds,
             MQECHAR * pName, MQEINT64 * pLong,
             MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutFloat( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEFLOAT* aFloat,
              MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutDouble( MQEHSESS hSess, MQEHFIELDS hFlds,
              MQECHAR * pName, MQEDOUBLE * pDouble,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEBYTE * *aByte* - input
A pointer to a byte value.

MQEINT16 * *int16Val* - input
A pointer to a 16 bit short integer value.

MQEINT32 **anInt* - input
A pointer to a 32 bit integer value.

MQEINT64 * *pLong* - output
A pointer to a 64 bit integer value.

MQEFLOAT **aFloat* - input
A pointer to a float value.

MQEDOUBLE * *aDouble* - input
A pointer to a double value.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

Returns "0" on success or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  byteVal;
MQFLOAT  floatVal;
MQDOUBLE  doubleVal;
MQEINT16  int16Val;
MQEINT32  int32Val;
MQEINT64  int64Val;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

byteVal = 0x45;
rc    = MQeFieldsPutByte( hSess, hFlds, "b", &byteVal,
                &compcode, &reason);
floatVal  = 2.55;
rc    = MQeFieldsPutFloat( hSess, hFlds, "f", &floatVal,
                &compcode, &reason);
doubleVal = 2.3413453231e-63;
rc    = MQeFieldsPutDouble( hSess, hFlds, "d", &doubleVal,
                &compcode, &reason);
int16Val  = 32000;
rc    = MQeFieldsPutShort( hSess, hFlds, "sh", &int16Val,
                &compcode, &reason);
int32Val  = 2000000000;
rc    = MQeFieldsPutInt( hSess, hFlds, "int", &int32Val,
               &compcode, &reason);
int64Val.hi = 265;
int64Val.lo = 2000000000;
rc    = MQeFieldsPutLong( hSess, hFlds, "lg", &int64Val,
                &compcode, &reason);
```

See Also

- **MQeFieldsGetByte**
- **MQeFieldsGetShort**
- **MQeFieldsGetInt**
- **MQeFieldsGetLong**
- **MQeFieldsGetFloat**
- **MQeFieldsGetDouble**

MQeFieldsPutAscii, MQeFieldsPutUnicode, MQeFieldsPutObject

Description

Put an array of MQECHAR, MQESHORT, or MQEBYTE into a single field of the MQeFields object.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutAscii( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQECHAR * pData, MQEINT32 DataLen,
              MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutUnicode( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQESHORT * pData, MQEINT32 DataLen,
              MQEINT32 * pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutObject( MQEHSESS hSess, MQEHFIELDS hFlds, MQECHAR * pName,
              MQEBYTE * pData, MQEINT32 DataLen,
              MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string name of the field. A null or a zero length string is invalid.

MQECHAR * *pStr* - input
Field data.

MQESHORT * *pStr* - input
Field data.

MQEBYTE * *pStr* - input
Field data.

MQEINT32 *DataLen* - input
The maximum number of MQECHAR, MQESHORT, or MQEBYTE to copy.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32
Returns "0" on success or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const MQECHAR * textVal = "The Owl and the Pussy Cat went to sea";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT32  n;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
rc   = MQeFieldsPutAscii( hSess, hFlds, "ibm", strlen(textVal)), textVal,
             &compcode, &reason);

/* Get the data length */
n    = MQeFieldsDataLen( hSess, hFlds, "ibm", &compcode, &reason);

pData  = (MQEBYTE *) calloc(n, , MQE_SIZEOF(datatype));

/* Copy out the data */
rc   = MQeFieldsGetAscii( hSess, hFlds, "ibm", pData, n,
             &compcode, &reason);
```

See Also

- **MQeFieldsGetAscii**
- **MQeFieldsGetUnicode**
- **MQeFieldsGetObject**

MQeFieldsPutArrayOfByte, MQeFieldsPutArrayOfShort, MQeFieldsPutArrayOfInt, MQeFieldsPutArrayOfLong, MQeFieldsPutArrayOfFloat, MQeFieldsPutArrayOfDouble

Description

Puts an array of 8, 16, 32, or 64 bit integers, floats or doubles into a single field in an MQeFields object. Return "0" on success, "-1" on error.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutArrayOfByte( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEBYTE * pByte,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsPutArrayOfShort( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT16 * pShort,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsPutArrayOfInt( MQEHSESS hSess, MQEHFIELDS hFlds,
               MQECHAR * pName, MQEINT32 * pInt,
               MQEINT32 n, MQEINT32 * pCompCode,
               MQEINT32 * pReason)

MQEINT32 MQeFieldsPutArrayOfLong( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT64 * pLong,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsPutArrayOfFloat( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEFLOAT * pFloat,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)

MQEINT32 MQeFieldsPutArrayOfDouble( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEDOUBLE * pDouble,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEBYTE * *pByte* - input
An array of bytes.

MQEINT16 * *pShort* - input
An array of 2 byte integers.

MQEINT32 * *pInt* - input
An array of 4 byte integers.

MQEINT64 * *pLong* - output
An array of 8 byte integers.

MQEFLOAT * *pFloat* - input
An array of floats.

MQEDOUBLE * *pDouble* - input
An array of doubles.

MQEINT32 *n* - input
The number of elements to put. If "0", the number of elements in the field is returned.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32
Returns "0" on success or "-1" on failure.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEBYTE  bytes[4];
MQEFLOAT  floats[2];
MQEDOUBLE doubles[2];
MQEINT16  shorts[2];
MQEINT32  ints[3];
MQEINT64  longs[2];
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

*(MQEINT32 *)bytes = 0x30313233;
rc    = MQeFieldsPutByte( hSess, hFlds, "b", 4, [0],
              &compcode, &reason);
floats[0]= 2.55;
floats[1]= 3.14;
rc    = MQeFieldsPutFloat( hSess, hFlds, "f", 2, [0],
             &compcode, &reason);
doubles[0]= 2.3413453231e-63;
doubles[1]= 3.3413453231e-44;
rc    = MQeFieldsPut( hSess, hFlds, "d", [0], 2,
            &compcode, &reason);
shorts[0]= 32000;
shorts[1]= 32020;
rc    = MQeFieldsPutArrayOfShort( hSess, hFlds, "sh", [0], 2,
                &compcode, &reason);
ints[0] = 2000100000;
ints[1] = 2000020000;
ints[2] = 2000003000;
rc    = MQeFieldsPutArrayOfInt( hSess, hFlds, "int", [0], 3,
                &compcode, &reason);
longs[0].hi = 265;
longs[0].lo = 2000000000;
longs[1].hi = 2000000000;
```

```
longs[1].lo = 255;
rc    = MQeFieldsPutArrayOfLong( hSess, hFlds, "lg", [0], 2,
                &compcode, &reason);
```

See Also

- **MQeFieldsGetArrayOfByte**
- **MQeFieldsGetArrayOfShort**
- **MQeFieldsGetArrayOfInt**
- **MQeFieldsGetArrayOfLong**
- **MQeFieldsGetArrayOfFloat**
- **MQeFieldsGetArrayOfDouble**

MQeFieldsPutShortArray, MQeFieldsPutIntArray, MQeFieldsPutLongArray, MQeFieldsPutFloatArray, MQeFieldsPutDoubleArray

Description

Puts an array of MQEINT16, MQEINT32, MQEINT64, MQEFLOAT, MQEDOUBLE, or MQEHFIELDS as an encoded array into an MQeFields object. The array elements are inserted in order as encoded fields followed by the array length. Returns the total number of fields put into the MQeFields object.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutShortArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEINT16 * pData,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsPutIntArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT32 * pData,
                MQEINT32 n, MQEINT32 *
                pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutLongArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEINT64 * pData,
                MQEINT32 n, MQEINT32 *
                pCompCode, MQEINT32 * pReason)

MQEINT32 MQeFieldsPutFloatArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEFLOAT * pData,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsPutDoubleArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEDOUBLE * pData,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsPutFieldsArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQEHFIELDS * pData,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT16 * *pData* - input
An input array.

MQEINT32 * *pData* - input
An input array.

MQEINT64 * *pData* - input
An input array.

MQEFLOAT * *pData* - input
An input array.

MQEDOUBLE * *pData* - input
An input array.

MQEHFIELDS * *pData* - input
An input array.

MQEINT32 *n* - input
The number of elements to put.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of fields successfully put.
- On failure, returns a count of the number of fields processed including the failing field.
- If an error occurs prior to any fields being processed, "-1" is returned.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT16  shorts[2];
MQEINT32  ints[3];
MQEINT64  longs[2];
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);
shorts[0]= 32000;
shorts[1]= 32020;
rc    = MQeFieldsPutShortArray( hSess, hFlds, "boo", 2, [0],
                  &compcode, &reason);
ints[0] = 2000100000;
ints[1] = 2000020000;
ints[2] = 2000003000;
rc    = MQeFieldsPutIntArray( hSess, hFlds, "foo", 3, [0],
                 &compcode, &reason);
longs[0].hi = 265;
longs[0].lo = 2000000000;
longs[1].hi = 2000000000;
longs[1].lo = 255;
rc    = MQeFieldsPutLongArray( hSess, hFlds, "poo", 2, [0],
                  &compcode, &reason);

memset(shorts, 0, sizeof(shorts);
```

```
memset(ints , 0, sizeof(ints);
memset(longs , 0, sizeof(longs);

/* Get individual data element */
MQeFieldsGetShort( hSess, hFlds, "boo:0" , [0] , &compcode, &reason);
MQeFieldsGetShort( hSess, hFlds, "boo:1" , [1] , &compcode, &reason);
MQeFieldsGetInt ( hSess, hFlds, "foo:0" , [0]  , &compcode, &reason);
MQeFieldsGetInt ( hSess, hFlds, "foo:1" , [1]  , &compcode, &reason);
MQeFieldsGetInt ( hSess, hFlds, "foo:2" , [2]  , &compcode, &reason);
MQeFieldsGetLong ( hSess, hFlds, "poo:0" , [0] , &compcode, &reason);
MQeFieldsGetLong ( hSess, hFlds, "poo:1" , [1] , &compcode, &reason);
```

See Also

- MQeFieldsGetShortArray
- MQeFieldsGetIntArray
- MQeFieldsGetLongArray
- MQeFieldsGetFloatArray
- MQeFieldsGetDoubleArray

MQeFieldsPutAsciiArray, MQeFieldsPutUnicodeArray, MQeFieldsPutByteArray

Description

Puts a 2 dimensional array of MQEINT16, MQECHAR, or MQEBYTE as an encoded array into an MQeFields object. The array elements are inserted in order as encoded fields, followed by the array length. Returns the total number of fields added to the MQeFields object.

Syntax

```
#include <hmq.h>
#include <hmqHelper.h>
MQEINT32 MQeFieldsPutAsciiArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                 MQECHAR * pName, MQECHAR * ppData[],
                 MQEINT32 pDataLen[], MQEINT32 srcOff,
                 MQEINT32 n, MQEINT32 * pCompCode,
                 MQEINT32 * pReason)

MQEINT32 MQeFieldsPutUnicodeArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                  MQECHAR * pName, MQEINT16 * ppData[],
                  MQEINT32 pDataLen[], MQEINT32 srcOff,
                  MQEINT32 n, MQEINT32 * pCompCode,
                  MQEINT32 * pReason)

MQEINT32 MQeFieldsPutByteArray( MQEHSESS hSess, MQEHFIELDS hFlds,
                MQECHAR * pName, MQEBYTE * ppData[],
                MQEINT32 pDataLen[], MQEINT32 srcOff,
                MQEINT32 n, MQEINT32 * pCompCode,
                MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQEHFIELDS *hFlds* - input
A handle to an MQeFields object.

MQECHAR * *pName* - input
A null terminated string containing the name of the field. A null or a zero length string is invalid.

MQEINT32 *n* - input
The number of elements to put. If "0", the number of elements in the field is returned.

MQECHAR * *ppData[]* - input
An array of MQECHAR arrays.

MQEINT16 * *ppData[]* - input
An array of MQESHORT arrays.

MQEBYTE * *ppData[]* - input
An array of MQEBYTE.

MQEINT32 *pDataLen[]* - input
An array of lengths of each data element, corresponding to each element of *ppData[]* .

MQEINT32 *srcOff* - input
The starting index from which to copy the array element.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned ***pCompCode*** equals MQECC_ERROR, *pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success, returns the number of fields successfully put.
- On failure, returns a count of the number of fields processed including the failing field.
- If an error occurs prior to any fields being processed, "-1" is returned.

Example

```
#include <hmq.h>
#include <hmqHelper.h>
static MQECHAR const * FieldsType = "com.ibm.mqe.MQeFields";
static const char * textArray[] =
        { "The Owl and the Pussy Cat went to sea",
         "Here we go round the Mulberry bush",
         "Jack and Jill went up the hill" };
MQEHSESS  hSess;
MQEINT32  compcode;
MQEINT32  reason;
MQEHFIELDS hFlds;
MQEINT32  pStrLen[3], n, *pStrLen2;
MQEBYTE * pData;
MQEINT32  rc;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
hFlds  = MQeFieldsAlloc( hSess, FieldsType, &compcode, &reason);

pStrLen[0] = strlen(textArray[0]);
pStrLen[1] = strlen(textArray[1]);
pStrLen[2] = strlen(textArray[2]);
rc = MQeFieldsPutAsciiArray( hSess, hFlds, "ibm", textArray, pStrLen, 3,
             &compcode, &reason);

/* 1. Get number of elements */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", NULL, NULL, 0, 0,
             &compcode, &reason);

/* Get space for array of string length */
pStrLen2 = (MQEINT32 *) malloc(n * sizeof(MQEINT32));
memset(pStrLen2, 0, n * sizeof(MQEINT32));

/* 2. Get array of string length */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", NULL, pStrLen2, 0, n,
             &compcode, &reason);

/* Get space for array of string */
for (i=0; i<n; i++) {
  pStr[i] = (MQECHAR *) malloc(pStrLen[j]+1);
  memset(pStr[i], 0, pStrLen[j]+1);
}

/* 3. Get array of strings */
n = MQeFieldsGetAsciiArray( hSess, hFlds, "ibm", pStr, pStrLen2, 0, n,
             &compcode, &reason);
```

MQeFieldsPutByteArray

See Also

- **MQeFieldsGetAsciiArray**
- **MQeFieldsGetUnicodeArray**
- **MQeFieldsGetByteArray**

System

The following APIs are used to interact with MQSeries Everyplace:

MQeInitialize
Initiates a session with the MQSeries Everyplace client library.

MQeTerminate
Terminates a session with the MQSeries Everyplace client library.

MQeGetVersion
Gets the version number of current MQSeries Everyplace software.

MQeConfigCreateQMgr
Initializes and creates a queue manager presence on the system.

MQeConfigDeleteQMgr
Terminates and removes the presence of a queue manager in the system.

MQeTraceCmd
Enables trace.

MQeTrace
Writes a trace string to default trace output.

General constraints

A queue manager name must:

- Not be `NULL`. An empty string "" defaults to the local queue manager.
- Conform to the ASCII character set. That is characters with values that are greater than 31 but less than 128 and must not include any of the characters "{}[]#():;,'=" .
- Be less than 48 characters long if it is required to interoperate with WebSphere MQ.

A queue name must

- Be at least one character long.
- Conform to the ASCII character set. That is characters with values that are greater than 31 but less than 128 and must not include any of the characters "{}[]#():;,'=" .
- Be less than 48 characters long if it is required to interoperate with WebSphere MQ.

MQeInitialize

Description

Initializes MQSeries Everyplace for the application. If the initialization is successful, this API creates a handle to the session object for use in subsequent calls to the MQSeries Everyplace subsystem. This handle must be specified on all subsequent message queuing calls issued by the application. The handle ceases to be valid when the **MQeTerminate** call is issued.

Syntax

```
#include <hmq.h>
MQEHSESS MQeInitialize( MQECHAR * SessionName, MQEINT32 * pCompCode,
                        MQEINT32 * pReason)
```

Parameters

MQECHAR * *SessionName* - input

This is the null terminated string name that identifies this application or a component of this application. Because this name is used to identify the session, any open session with the same name is closed and all resources associated with it are released. This allows the library to recover from applications that crash without calling the **MQeTerminate** API.

The SessionName must be:

- At least one character long (a null or a zero length string is invalid)
- Conform to the ASCII character set except "{ } [ ] # ( ) : ; , ' ="

There is no limit to the length of the name but you are recommended to keep it short, preferably less than 20 characters.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

MQE_WARN_SESSION_DELETED

A session with the same name was deleted. This could happen if a session was left open because the application that opened it crashed or exited without calling the **MQeTerminate** API.

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_ARGUMENT

Invalid session name, too short or too long.

MQE_EXCEPT_ALLOCATION_FAILED

MQSeries Everyplace library has too few session handles or system storage resources.

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

Local queue manager name is not set.

MQE_EXCEPT_QMGR_NOT_ACTIVE
PalmOS At least one of the three MQSeries Everyplace resources, hmqLib.prc, hmqFields.prc or hmqIni.prc, is not installed on this device.

Return Value

MQEHSESS *hSess*
A session handle. If any error occurs during the initialization, then an MQEHANDLE_NULL is returned.

Example

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode;
MQEINT32  reason;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
if (hSess!=MQEHANDLE_NULL) {
   MQeTerminate(hSess, &compcode, &reason, );
}
```

See Also

MQeTerminate

MQeTerminate

Description

Terminates an application's session with the MQSeries Everyplace subsystem.

Syntax

```
#include<hmq.h>
MQEVOID MQeTerminate( MQEHSESS hSess, MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input

The session handle returned by **MQeInitialize**

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_WARNING, **pReason* may have any of the following values:

MQE_WARN_SESSION_DELETED

A session with the same name was deleted. This could happen if a session was left open because the application that opened it crashed or exited without calling the **MQeTerminate** API.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEVOID

None

Example

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode;
MQEINT32  reason;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
if (hSess!=NULL) {
   MQeTerminate(hSess, &compcode, &reason);
}
```

See Also

MQeInitialize

MQeGetVersion

Description

Get the version number of the MQSeries Everyplace software running on the device.

Syntax

```
#include <hmq.h>
MQEINT32 MQeGetVersion ( MQEINT32 * pCompCode, MQEINT32 * pReason);
```

Parameters

MQEINT32 * *pCompCode*- output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

Return Value

MQEINT32

Four ASCII character value representing the current version, such as "1.00".

Example

```
#include <hmq.h>
MQEINT32  compcode;
MQEINT32  reason;
MQEINT32  version;

version = MQeGetVersion(&compcode, &reason);
```

MQeConfigCreateQMgr

Description

Initialize and create a queue manager on the device.

Syntax

```
#include <hmq.h>
MQEVOID MQeConfigCreateQMgr ( MQECHAR * pQMgrName, MQEINT32 * pCompCode,
                              MQEINT32 * pReason);
```

Parameters

MQECHAR * *pQMgrName* - input

The name of the local queue manager to be created.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_QMGR_ALREADY_EXISTS

An existing queue manager name is defined. Call **MQeQMgrGetName** to retrieve the current local queue manager name, then call **MQeMQeConfigDeleteQMgr** to delete the local queue manager, and then call this function again.

Return Value

None.

Example

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode, reason;
MQEINT16  len;
MQECHAR   name[128];

hSess     = MQeInitialize( "aSession", &compcode, &reason);
len       = MQeQMgrGetName( hSess, name, 128, &compcode, &reason);
name[len] = '\0';

MQeConfigDeleteQMgr( name, &compcode, &reason);

MQeConfigCreateQMgr( "MyOwnQMgr", &compcode, &reason);
```

See Also

- **MQeQMgrGetName**
- **MQeConfigDeleteQMgr**

MQeConfigDeleteQMgr

Description

Terminate and remove the presence of MQSeries Everyplace queue manager on the device.

Syntax

```
#include <hmq.h>
MQEVOID MQeConfigDeleteQMgr ( MQECHAR * pQMgrName, MQEINT32 * pCompCode,
                              MQEINT32 * pReason);
```

Parameters

MQECHAR * *pQMgrName*- input

Name of the local queue manager to be created.

MQEINT32 * *pCompCode* - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

The queue manager name provided does not match the current MQSeries Everyplace queue manager name.

Return Value

None.

Example

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode, reason;
MQEINT16  len;
MQECHAR   name[128];

hSess     = MQeInitialize( "aSession", &compcode, &reason, );
len       = MQeQMgrGetName( hSess, name, 128, &compcode, &reason, );
name[len] = '\0';

MQeConfigDeleteQMgr( name, &compcode, &reason, );

MQeConfigCreateQMgr( "MyOwnQMgr", &compcode, &reason, );
```

See Also

- **MQeQMgrGetName**
- **MQeConfigCreateQMgr**

MQeTraceCmd

MQeTraceCmd

Description

Starts, stops and sets the option of the MQSeries Everyplace runtime tracing facility. The destination of the trace output is platform dependent. On PalmOS, the trace is written to a standard MemoPad database and can be viewed by calling the **MemoPad** application.

Syntax

```
#include <hmq.h>
MQEVOID MQeTraceCmd ( MQEHSESS hSess, MQEINT32 Cmd, MQEINT32 Parm,
                      MQEINT32 * pCompCode, MQEINT32 * pReason);
```

Parameters

MQEHSESS *hSess* - input
This session handle returned by **MQeInitialize**.

MQEINT32 *Cmd* - input

MQE_TRACE_CMD_START
Starts the trace. *Parm* is ignore.

MQE_TRACE_CMD_STOP
Stops the trace. *Parm* is ignore

MQE_TRACE_CMD_SET_MASK
Set the trace mask bits specified in *Parm*

MQEINT32 *Parm* - input
If *Cmd* is MQE_TRACE_CMD_SET_MASK then this parameter is

MQE_TRACE_OPTION_APP_MSG
Write out an application trace string that starts with a character

MQE_TRACE_OPTION_APP_INFO
Write out an application trace string that starts with character "I"

MQE_TRACE_OPTION_APP_WARNING
Write out an application trace string that starts with character "W"

MQE_TRACE_OPTION_APP_ERROR
Write out an application trace string that starts with character "E"

MQE_TRACE_OPTION_APP_DEBUG
Write out an application trace string that starts with character "D"

MQE_TRACE_OPTION_APP_ALL
Write out all application trace strings

MQE_TRACE_OPTION_SYS_MSG
Write out a system trace string that starts with character "_"

MQE_TRACE_OPTION_SYS_INFO
Write out a system trace string that starts with character "i".

MQE_TRACE_OPTION_SYS_WARNING
Write out a system trace string that starts with character "w"

MQE_TRACE_OPTION_SYS_ERROR
Write out a system trace string that starts with character "e"

MQE_TRACE_OPTION_SYS_DEBUG
Write out a system trace string that starts with character "d"

MQE_TRACE_OPTION_SYS_ALL
Write out all system trace strings.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAIL
The MQSeries Everyplace library has too few resources.

Return Value

None.

Example

```
#include <hmq.h>
MQEHSESS hSess;
MQEINT32  compcode, reason;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);

/* Start the trace */
MQeTraceCmd ( hSess, MQE_TRACE_CMD_START, 0,
              &compcode, &reason );
MQeTraceCmd ( hSess, MQE_TRACE_CMD_SET_MASK,
              MQE_TRACE_OPTION_SYS_ERROR | MQE_TRACE_OPTION_APP_MSG,
              &compcode, &reason);

MQeTrace( hSess, MQTS(" Starting MQe..."));
MQeTrace( hSess, MQTS("IThis is a information trace msg."));

/* Stop the trace */
MQeTraceCmd ( hSess, MQE_TRACE_CMD_STOP, 0, &compcode, &reason );

/* Terminate the MQe session */
MQeTerminate( hSess, &compcode, &reason);
```

See Also

MQeTrace

MQeTrace

Description

Writes a string to the MQSeries Everyplace trace facility. The size of the string character MQETCHAR and the destination of the trace output is platform dependent. On PalmOS, the string character is a single byte and the trace is written to a standard MemoPad database and can be viewed by calling the **MemoPad** application. For code portability, it is recommended that the trace string be wrapped in an **MQTS()** macro.

Syntax

```
#include <hmq.h>
MQEVOID MQeTrace ( MQEHSESS hSess, MQETCHAR * pTStr);
```

Parameters

MQEHSESS *hSess* - input
The session handle returned by **MQeInitialize**.

MQETCHAR *pTStr* - input
A null terminated trace string.

Return Value

None.

Example

```
#include <hmq.h>
MQHSESS hSess;
MQEINT32  compcode;
MQEINT32  reason;

hSess = MQeInitialize("MyAppsName", &compcode, &reason );

/* Start the trace */
MQeTraceCmd ( hSess, MQE_TRACE_CMD_START, 0, &compcode, &reason);
MQeTraceCmd ( hSess, MQE_TRACE_CMD_SET_MASK,
              MQE_TRACE_OPTION_SYS_ERROR | MQE_TRACE_OPTION_APP_MSG,
             &compcode, &reason );

MQeTrace( hSess, MQTS(" Starting MQe..."));
MQeTrace( hSess, MQTS("IThis is a information trace msg."));

/* Stop the trace */
MQeTraceCmd ( hSess, MQE_TRACE_CMD_STOP, 0, &compcode, &reason  );

/* Terminate the MQe session */
MQeTerminate( hSess, &compcode, &reason);
```

See Also

MQeTraceCmd

MQeQMgr APIs

The following APIs are used to interact with the MQSeries Everyplace queue manager.

MQeQMgrBrowseMsgs
Browses messages on a queue.

MQeQMgrConfirmMsg
Deletes a message already retrieved from a queue or makes a previously put message available.

MQeQMgrDeleteMsgs
Deletes an array of messages on the queue.

MQeQMgrGetMsg
Gets a message from a queue.

MQeQMgrGetName
Gets the name of the local queue manager.

MQeQMgrPutMsg
Puts a message onto a queue.

MQeQMgrUndo
Undoes one or more messages that were put, retrieved, or locked on a queue.

MQeQMgrUnlockMsgs
Unlocks an array of messages on the queue that were locked by **MQeQMgrBrowseMsgs()**.

The general constraints listed in "General constraints" on page 135 apply to all the queue manager APIs.

MQeQMgrBrowseMsgs

Description

Browses messages on a queue without removing the messages from the queue. The browse returns an array of message object handles. The application can then interrogate the message objects. A filter can be used to make the browse more specific. For example, message object fields (for example, *MessageId* and *Priority*), could be specified so that only messages that have matching fields are returned.

The application specifies the size of the array into which the results are returned. This application programmer can therefore control the number of matched messages returned on a single browse call. The array size has a maximum limit in the MQSeries Everyplace system and is set at 13 concurrent handles in Version 1.2.7. This is important for devices that have limited resources and, therefore, may not be able to store all the matching messages. To retrieve the rest of the matched messages, the application can subsequently make repeated calls to this function passing the same *pBrowseMsgOpts* as on the first call. *pBrowseMsgOpts* points to an MQEBMO type which maintains the context information for the browse.

Once a browse operation has been initiated, all subsequent **MQeBrowseMsgs()** calls that use the same MQEBMO structure are directed to the queue manager and queue specified on the first call. Any changes to these parameters on subsequent calls are ignored. Once the resources assigned to an MQEBMO structure are released, the structure can be reused for a new browse operation to a different queue manager and queue.

The application is responsible for calling **MQeFieldsFree** to deallocate the returned message object handles.

Syntax

```
#include <hmq.h>
MQEINT32 MQeQMgrBrowseMsgs( MQEHSESS hSess, MQECHAR * pQMName,
                            MQECHAR * pQName, MQEVOID * pBrowseMsgOpts,
                            MQEHFIELDS hFilter, MQEHFIELDS pMsgs[ ],
                            MQEINT32 nMsgs, MQEINT32 * pCompCode,
                            MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated ASCII string containing the name of the queue manager. An empty string name "" defaults to the local queue manager. A null is invalid input.

MQECHAR * *pQName* - input
A null terminated ASCII string containing the name of the queue. A null or an empty string is invalid.

MQEVOID * *pBrowseMsgOpts* - input
A pointer to a data structure that contains the following elements:

```
typedef struct tagMQeBrowseMsgOpts{
   MQECHAR    StrucId[4];         /* Input  */
   MQEINT32   Version;            /* Input  */
   MQEINT32   Options;            /* Input  */
   MQEINT64   ConfirmId;          /* Input  */
```

```
   MQEHATTRB  hAttrb;            /* Input  */
   MQEINT64   LockId;            /* Output */
   MQEINT64   Cookie;            /* Output */
} MQEBMO;
```

MQECHAR *StrucId[4]* - input
The Structure ID for the **GetMsgOpts** which is BRWS .

MQEINT32 *Version* - input
The version number of this data structure. The current version number is 1.

MQEINT32 *Options* - input

MQE_QMGR_OPTION_BROWSE_LOCK
Browse the messages that match the *hFilter*. Lock all these messages on the queue to make them inaccessible to future **MQeQMgrBrowseMsgs()** or **MQeQMgrGetMsg()** operations. The locked messages can either be deleted with **MQeQMgrDeleteMsgs** or unlocked with **MQeQMgrUnlockMsgs**. If a *confirmID* is supplied, **MQeQMgrUndo** can be used to unlock the messages on the queue.

If you are browsing a remote queue synchronously, it is highly recommended that your application also sets the MQE_QMGR_OPTION_CONFIRMID option when using the MQE_QMGR_OPTION_BROWSE_LOCK option. This is because a network communication error can cause the returned data packet that contains the *LockID* field to be lost, and without this *LockID*, the locked messages on the queue cannot be unlocked by the application. In this case, MQSeries Everyplace system administrative intervention would be required. However, with a *ConfirmID*, the application can recover from this error condition by calling the **MQeQMgrUndo** function to unlock the messages on the remote queue and make these messages available to the application again.

MQE_QMGR_OPTION_BROWSE_JUST_UID
Browse the messages that match the *hFilter* and return message objects that contain only the unique IDs

MQE_QMGR_OPTION_CONFIRMID
Include the *confirmID* in the **BrowseMsg** operation.

The above three options can be used together in any combination.

MQEINT64 *ConfirmId*
A 64 bit integer that the application supplies to tag the returned message object on the queue. The tagged message object is made inaccessible for subsequent**MQeQMgrBrowseMsgs()** calls and for**MQeQMgrGetMsg()** calls without the ***UID*** of the message. These messages are made accessible again after **MQeQMgrUndo()** is called with this *ConfirmID*.

This *ConfirmID* value must be different for different devices, so that no two devices can **put**, **get** or **browse** locked messages on the same queue with the same *ConfirmID*. Otherwise an undo operation issued by one device could affect the messages of another device with the same *confirmID*.

The default value is "0".

If MQE_QMGR_OPTION_CONFIRMID is set and *ConfirmID* is "0", or if *ConfirmID* is nonzero and MQE_QMGR_OPTION_CONFIRMID is not set, the call fails.

This *ConfirmID* is intended to be used with the **MQeQMgrUndo** function, and should not be used with the **MQeQMgrConfirmMsg** function.

MQEHATTRB *hAttrb* - input

The handle to the MQeAttribute object that is used to decode the message objects on the queue before it is returned by this function. This parameter is used for message-level security. The default value is MQEHANDLE_NULL.

Version 1.2.7 Note: Message-level security is not supported, so this parameter is ignored.

MQEINT64 *LockId* - output

A 64 bit integer returned by the queue manager when the MQE_QMGR_OPTION_BROWSE_LOCK option is set. If this option is not set, the return value of this parameter is undefined. This value is associated with the message object handles that are copied into the *pMsgs[]* array. The value returned in this parameter may be different for each call to this function.

The returned *LockID* is used by **MQeQMgrUnlockMsgs** to unlock the locked message.

A locked message remains locked until one of the following occurs:

- It is unlocked by the **MQeQMgrUnlockMsgs** using the *LockId* or
- It is deleted by **MQeQMgrDeleteMsgs()**
- It is retrieved with an **MQeQMgrGetMsg()** call using a filter containing the *LockId*
- The message expires on the queue

Otherwise locked messages can only be unlocked by the MQSeries Everyplace system utility.

MQEINT64 *Cookie* - output and input

A queue manager generated number that the application must pass back to this function on subsequent calls to retrieve the next set of message handles. This number serves as a bookmark that the queue manager uses to find the starting point in the queue to start the browse operation. The application need not understand the meaning of this value except to pass it back on subsequent calls. The first time this function is called, *Cookie* must be

zero. To browse the remaining messages, the same input parameters <*pQMName*,*pQName*,*hFilter*,*hAttrb*> must be supplied on subsequent calls. If any of these four parameters differs from the original ones that the queue manager used to generate the *Cookie* , then the "book mark" is still used as the starting point to return the message. If *Cookie* is zero, then a new browse operation is initiated.

The implementation of this cookie may hold resource. These resources are released when

- The last message that satisfies the *hFilter* is browsed.
- *pMsgs[]* is a NULL.

  If an application has completed the required browse operation before the last message is browsed, it can release any resource held by the cookie by setting *pMsgs[]* to NULL in the subsequent browse call.
- **MQeTerminate** is called.

The default value is zero.

If *pBrowseMsgOpts* is a NULL, then an MQEBMO data structure with the default values is used.

MQEHFIELDS *hFilter* - input
A handle to the filter that contains the matching fields for the messages on the queue. If no filter is provided, then all currently unlocked messages up to *nMsgs* on the queue are returned. If the MQE_QMGR_OPTION_BROWSE_LOCK option is set, at least *nMsgs* matching messages, and possibly all the matching messages on the queue are locked. The number of messages locked depends on the implementation.

Default value is MQEHANDLE_NULL.

MQEHFIELDS *pMsgs[]* - output
An array to hold returned message object handles. If this is NULL, then zero is returned. If *pCookie* is not NULL, then its resources are released. Users are expected to call **MQeFieldsFree()** to release MQeFields handles held by this array.

MQEINT32 *nMsgs* - input
The number of message to browse for this call.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_Q_NO_MSG_AVAILABLE

MQE_EXCEPT_Q_NO_MATCHING_MSG

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEINT32

The number of message object handles returned in the *pMsgs[]* array. This number is less than or equal to *nMsgs*.

Example

```
#include <hmq.h>
MQEHSESS   hSess;
MQEHFIELDS hFilter = MQEHANDLE_NULL;
MQEINT32   i, n, nMsgs;
MQEINT32   compcode;
MQEINT32   reason;
MQEBMO     bmo = MQEBMO_DEFAULT;
MQEHFIELDS pMsgs[2];
MQECHAR   *qm, *q;

qm = "MyQM";
q  = "QQ";
hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
nMsgs  = 2;

/*--------------------------------------*/
/* Browse with no locking or confirm ID */
/*--------------------------------------*/
n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                       pMsgs, nMsgs, &compcode, &reason );

/* Now set the browse option for lock and confirm */
bmo.Option = MQE_QMGR_BROWSE_LOCK | MQE_QMGR_CONFIRMID;
/* Set the confirm ID */
bmo.ConfirmId.hi = bmo.ConfirmId.lo = 0x12345678;

/*--------------------------------------*/
/* Browse and undo                      */
/*--------------------------------------*/
n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                     pMsgs, nMsgs, &compcode, &reason );

MQeQMgrUndo(hSess, qm, q, bmo.ConfirmId, &compcode, &reason, );

/*--------------------------------------*/
/* Browse and delete                    */
/*--------------------------------------*/
/* Browse nMsgs at a time until no messages are left */
while (1) { /* do forever */
   /* Browse the nMsgs matching messages */
   n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                       pMsgs, nMsgs, &compcode, &reason );

   if (n==0) {
      /* Any resources held by the cookie has been released already */
      break;
   }

   for(i=0; i<n; i++) {
      /****************************************/
      /* Process the message objects in pMsgs[] */
      /****************************************/
   }
```

```
   /* Delete the n locked messages in pMsgs[] */
   MQeQMgrDeleteMsgs( hSess, qm, q, pMsgs, n, &compcode, &reason );

   /* free pMsgs[] handle resources */
   for(i=0; i<n; i++) {
      MQeFieldsFree(hSess, pMsgs[i], &compcode, &reason);
   }
};

MQeTerminate(hSess, &compcode, &reason);
```

See Also

- **MQeQMgrDeleteMsgs**
- **MQeQMgrUnlockMsgs**
- **MQeQMgrUndo**

MQeQMgrConfirmMsg

Description

This function is used to support the assured message delivery mechanism of MQSeries Everyplace. This API call tells the queue manager to commit the previous **MQeQMgrGetMsg** or **MQeQMgrPutMsg** operation. The application must have supplied a *ConfirmID* with these previous calls. The input parameter *hMsg* must contain the unique identifier *UID* of the message object that is to be confirmed. The unique identifier of a message object is a 64 bit integer value and the string name of the origin queue manager.

This function confirms only a single **MQeQMgrGetMsg** or **MQeQMgrPutMsg** operation and not a set of them, therefore this API is not a unit-of-work function.

Syntax

```
#include <hmq.h>
MQEVOID MQeQMgrConfirmMsg( MQEHSESS hSess, MQECHAR * pQMName,
                           MQECHAR * pQName, MQEINT32 Option,
                           MQEHFIELDS hMsg, MQEINT32 * pCompCode,
                           MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
This session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string containing the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEINT32 *Option* - input

MQE_QMGR_OPTION_CONFIRM_GETMSG
Confirms an earlier **MQeQMgrGetMsg** operation

MQE_QMGR_OPTION_CONFIRM_PUTMSG
Confirms an earlier **MQeQMgrPutMsg** operation.

If both options are set, then MQE_QMGR_OPTION_CONFIRM_GETMSG takes precedent.

MQEHFIELDS *hMsg* - input
An MQeFields object that contains the unique identifier of the message object to be confirmed. This could be the same messages object handle that was used earlier with the **MQeQMgrGetMsg** and **MQeQMgrPutMsg** function call with the MQE_QMGR_OPTION_CONFIRMID option set. The function extracts the unique identifier of the message object handle and uses it to confirm the message on the queue. All other fields in the *hMsg* are ignored.

The application has to call **MQeFieldsFree()** to free the message object handle.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

hMsg does not contain the *UID* fields of a message object.

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_NOT_FOUND

No *confirmID* is associated with the *UID* supplied in the *hMsg* .

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEVOID

Example

```
#include <hmq.h>
MQEHSESS   hSess;
MQCHAR *   qm = "myQM";
MQCHAR *   q  = "QQ";
MQEHFIELDS hFilter = MQEHANDLE_NULL;
MQEINT32   i, n, nMsgs;
MQEINT32   compcode;
MQEINT32   reason;
MQEGMO     gmo = MQEGMO_DEFAULT;

hSess = MQeInitialize("MyAppsName", &compcode, &reason);

/* Set up the GMO for confirm msg operation */
gmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;
gmo.ConfirmId.hi = 0;
gmo.ConfirmId.lo = 0x55aa;

hMsg  = MQeQMgrGetMsg( hSess, qm, q, &gmo, hFilter, &compcode, &reason);

/* Process the message */

/* Confirms the message */
MQeQMgrConfirmMsg( hSess, qm, q, MQE_QMGR_OPTION_CONFIRM_GETMSG, hMsg,
                   &compcode, &reason);
MQeTerminate(hSess, &compcode, &reason);
```

See Also

- **MQeQMgrBrowseMsgs**
- **MQeQMgrGetMsg**
- **MQeQMgrPutMsg**
- **MQeQMgrUndo**

MQeQMgrDeleteMsgs

Description

Deletes the messages on a queue identified by the unique identifier of each message. The unique identifier is a combination of an 8 bytes integer unique ID and the origin queue manager name of the messages. The application is responsible for calling the **MQeFieldsFree()** to free the message object handles in the input array.

Syntax

```
#include <hmq.h>
MQEINT32 MQeQMgrDeleteMsgs( MQEHSESS hSess, MQECHAR * pQMName,
                            MQECHAR * pQName,  MQEHFIELDS pMsgs[],
                            MQEINT32 nMsgs, MQEINT32 * pCompCode,
                            MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
This session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string containing the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEINT32 *pMsgs[]* - input
An array of message object handles to be deleted. To delete messages that are returned by a browse-and-lock function call, the messages object handles should be the input in this array. The queue manager extracts the unique identifier (*UID*) of each message object handle and sends it to the queue manager. The unique identifier of a message object is a 64 bit unique value and the string name of the origin queue manager. The rest of the fields in the message object are ignored.

If an entry in the *pMsgs[]* is a NULL, this NULL entry is skipped and the delete operation continues on to the next entry in the array. The delete operation stops when it encounters an exception, and any remaining message object handles not processed are left as-is and remain on the queue.

MQeFieldsFree() is used to release MQeFields handles stored in this array.

MQEINT32 *nMsgs* - input
The number of array elements in the *pMsgs[]* array, including elements that are NULL.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_Q_NO_MSG_AVAILABLE

MQE_EXCEPT_Q_NO_MATCHING_MSG
Could not find the message on the queue, and therefore, no message is deleted.

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEINT32
The number of the array entries successfully processed, including the NULL entries.

Example

```
#include <hmq.h>
MQEHSESS   hSess;
MQCHAR   * qm, *q;
MQEHFIELDS hFilter = MQEHANDLE_NULL;
MQEINT32   i, n, nMsgs;
MQEINT32   compcode;
MQEINT32   reason;
MQEBMO     bmo = MQEBMO_DEFAULT;
MQEHFIELDS pMsgs[2];

qm = "aQM";
q  = "QQ";
hSess = MQeInitialize("MyAppsName", &compcode, &reason);

/* Max. number of messages to get at a time for this run */
nMsgs  = 2;
bmo.cookie.hi = bmo.cookie.lo = 0;
bmo.lockId.hi = bmo.lockId.lo = 0;
bmo.option   |= MQE_QMGR_OPTION_BROWSE_LOCK;

/* Browse nMsgs at a time until no messages are left */
while (1) { /* do forever */
   /* Browse the nMsgs matching messages */
   n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                                  pMsgs, nMsgs, &cookie, &compcode, &reason);

   if (n==0) {
      /* Any resources held by the cookie has been released already */
      break;
   }

   for(i=0; i<n; i++) {
      /* Process the message objects in pMsgs[] */
   }

   /* Delete the n locked messages in pMsgs[] */
   MQeQMgrDeleteMsgs( hSess, qm, q, pMsgs, n, &compcode, &reason);

   /* free pMsgs[] handle resources */
   for(i=0; i<n; i++) {
      MQeFieldsFree(hSess, pMsgs[i], &compcode, &reason);
   }
};

MQeTerminate(hSess, &compcode, &reason);
```

See Also

MQeQMgrDeleteMsgs

- **MQeQMgrBrowseMsgs**
- **MQeQMgrUnlockMsgs**

MQeQMgrGetMsg

Description

Get the first message on a queue that matches the filter. This API returns a fields object handle whose object type is MQeMsgObject_Type, on a specified queue manager and queue. The returned message is deleted from the queue. The queue may belong to a different MQSeries Everyplace queue manager from the one to which the call was made. A filter can be specified, so that only messages that have matching attributes are returned.

The application programmer is responsible for calling **MQeFieldsFree** to deallocate the returned message handle.

Syntax

```
#include <hmq.h>
MQEHFIELDS MQeQMgrGetMsg( MQEHSESS hSess, MQECHAR * pQMName, MQECHAR * pQName,
                          MQEVOID * pGetMsgOpts, MQEHFIELDS hFilter,
                          MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string that contains the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEVOID * *pGetMsgOpts* - input
This parameter is a pointer to a data structure that contains the following elements:

```
typedef struct tagMQeGetMsgOpts{
   MQECHAR   StrucId[4];         /* Input  */
   MQEINT32  Version;            /* Input  */
   MQEINT32  Options;            /* Input  */
   MQEINT64  ConfirmId;          /* Input  */
   MQEHATTRB hAttrb;             /* Input  */
} MQEGMO;
```

MQECHAR *StrucId[4]* - input
The structure ID for the **GetMsgOpts** which is GETM .

MQEINT32 *Version* - input
The version number of this data structure. The current version number is "1".

MQEINT32 *Options* - input

MQE_QMGR_OPTION_CONFIRMID
Include the *ConfirmID*with the **GetMsg** operation. The retrieved message becomes inaccessible to subsequent **MQeQMgrBrowseMsg()** and **MQeQMgrGetMsg()** calls. It is not deleted from the queue until the **MQeQMgrConfirmMsg** is called with the *UID* of this message object or the message is made accessible again with **MQeQMgrUndo** call.

The default value is MQE_QMGR_OPTION_NONE.

MQEINT64 *ConfirmId* - input
A 64 bit integer that the application programmer supplies to mark the returned message object on the queue. The marked message object is made inaccessible to subsequent **MQeQMgrBrowseMsg()** and **MQeQMgrGetMsg()** calls until **MQeQMgrUndo** is called with this *ConfirmId*.

Default value is "0". If MQE_QMGR_OPTION_CONFIRMID is set and *ConfirmId* is "0", or if *ConfirmId* is nonzero and MQE_QMGR_OPTION_CONFIRMID is not set, the call fails.

MQEHATTRB *hAttrb* - input
The handle to the attribute object that is used to decode the message object on the queue before it is returned by this API. The default value is MQEHANDLE_NULL.

Note: Version 1.2.7 does not support message-level security so this parameter is ignored.

If this parameter is NULL, then an MQEGMO data structure with the default values is used.

MQEHFIELDS *hFilter* - input
A handle to the filter that has the matching criteria for the messages on the queue.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_Q_NO_MSG_AVAILABLE

MQE_EXCEPT_Q_NO_MATCHING_MSG

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEHFIELDS *hMsgObj*
The handle to a message object (an MQefields object with type MQE_OBJECT_TYPE_MQE_MSGOBJECT).

Example

```
#include <hmq.h>
MQEHSESS   hSess;
MQEHFIELDS hMsg, hFilter;
MQEINT32   compcode;
MQEINT32   reason;
MQEGMO     gmo = MQEGMO_DEFAULT;
MQECHAR  * aKey = "aKey", * qm, *q;

qm = "aQM";
```

```
q  = "QQ";

hSess = MQeInitialize("MyAppsName", &compcode, &reason);

/* Get msg with filter and confirmID*/

gmo.ConfirmId.hi = 0x2222;
gmo.ConfirmId.lo = 0x1111;
gmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

hFilter = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_FIELDS,
                          &compcode, &reason);
MQeFieldsPut( hSess, hFilter, "FindThis", MQE_TYPE_ASCII, aKey, strlen(aKey),
              &compcode, &reason);

/* Get a message that contains the field-name "FindThis", */
/*field-type of ASCII, and  a field-value of "aKey". */
hMsg  = MQeQMgrGetMsg( hSess, qm, q, &gmo, hFilter,
                       &compcode, &reason);

if (compcode==MQECC_OK) {
   /* Do something with the message. */

   /* Confirms the message, i.e., delete it off the queue. */
   MQeQMgrConfirmMsg( hSess, qm, q, MQE_QMGR_OPTION_CONFIRM_GETMSG, hMsg,
                      &compcode, &reason);

   /* Free the message handle */
   MQeFieldsFree( hSess, hMsg, &compcode, &reason);
}

MQeFieldsFree( hSess, hFilter, &compcode, &reason);
MQeTerminate( hSess, &compcode, &reason);
```

See Also

- **MQeQMgrConfirmMsg**
- **MQeQMgrPutMsg**
- **MQeQMgrUndo**

MQeQMgrGetName

Description

Get the string name of the local queue manager.

Syntax

```
#include <hmq.h>
MQEINT32 MQeQMgrGetName( MQEHSESS hSess, MQECHAR * pQMgrName,
                         MQEINT32 qmNameLen,  MQEINT32 * pCompCode,
                         MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQECHAR * *pQMgrName* - output
The output field into which the string name of the local queue manager is copied. If the buffer is NULL, the length of the local queue manager name is returned.

MQEINT32 *qmNameLen* - input
The buffer size of *pQMName*. If *pQMgrName* is NULL, then this parameter is ignored.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32 *qmnLen*
The length of the queue manager name.

Example

```
#include <hmq.h>
MQEHSESS  hSess;
MQEINT32  qmLen;
MQECHAR * qm;
MQEINT32  rc,len;
MQEINT32  compcode;
MQEINT32  reason;

hSess  = MQeInitialize("MyAppsName", &compcode, &reason);

len    = MQeQMgrGetName( hSess, NULL, 0, &compcode, &reason);
qm     = (MQECHAR *) malloc(len+1);
rc     = MQeQMgrGetName( hSess, qm, len, &compcode, &reason);
qm[len] = '\0';
printf("The Queue Manager Name is \"%s\"\n", qm);
MQeTerminate( hSess, &compcode, &reason);
```

MQeQMgrPutMsg

Description

Put a message on a queue. If the destination queue manager name is the same as the local queue manager name, then the message is put on a local queue (With the exception of AdminQ, local queue is not supported on the Palm in Version 1.2.7). If the destination queue manager name is a remote queue manager, then for synchronous messaging, a communication connection is made to the remote queue manager and the message is transmitted to that queue manager. This call is blocked until the message is transmitted to the remote queue manager.

When this API call returns, the unique identifier (*UID*) is set in the input message object, and it is set every time this API is called. So an application can call this API with the same input message object and the *UID* is set with a different value every time. This resetting mechanism guarantees that no message object with a duplicate *UID* enters the MQSeries Everyplace network.

The application must call **MQeFieldsFree** to deallocate the message handle *hMsg* .

Syntax

```
#include <hmq.h>
MQEVOID MQeQMgrPutMsg( MQEHSESS hSess, MQECHAR * pQMName, MQECHAR * pQName,
                       MQEVOID * pPutMsgOpts, MQEHFIELDS hMsg,
                       MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string containing the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEVOID * *putMsgOpts* - input and output
This parameter is a pointer to a data structure that contains the following elements:

```
typedef struct tagMQePutMsgOpts{
   MQECHAR   StrucId[4];         /* Input  */
   MQEINT32  Version;            /* Input  */
   MQEINT32  Options;            /* Input  */
   MQEINT64  ConfirmId;          /* Input  */
   MQEHATTRB hAttrb;             /* Input  */
} MQEPMO;
```

MQECHAR *StrucId[4]* - input
The structure ID for the **GetMsgOpts** that is PUTM .

MQEINT32 *Version* - input
The version number of this data structure. The current version number is "1".

MQEINT32 *Options* - input

MQE_QMGR_OPTION_CONFIRMID
Include the *ConfirmID*with the **PutMsg** operation. The put message is inaccessible to subsequent

MQeQMgrBrowseMsg() and **MQeQMgrGetMsg()** calls until **MQeQMgrConfirmMsg** is called with the *UID* of the *hMsg* or the message is deleted from the queue with **MQeQMgrUndo**.

The default value is MQE_QMGR_OPTION_NONE.

MQEINT64 *ConfirmId* - input
A 64 bit integer that the application programmer supplies to tag the returned message object on the queue.

The default value is "0". If MQE_QMGR_OPTION_CONFIRMID is set and *ConfirmId* is "0", or if *ConfirmId* is nonzero and MQE_QMGR_OPTION_CONFIRMID is not set, the call fails.

MQEHATTRB *hAttrb* - input
The handle to the attribute object that is use to decode the message object on the queue before it is returned by this API. Default value is MQEHANDLE_NULL.

Note: Version 1.2.7 does not support message-level security so this parameter is ignored.

If this parameter is NULL, then an MQEPMO data structure with the default values is used.

MQEHFIELDS *hMsg* - input and output
The message object to put on the queue. If this message object is one of the following types, which may have a request-reply messaging type, then for synchronous MQSeries Everyplace client that do not have a local AdminReplyQ queue, the reply message is returned in this parameter.

- com.ibm.mqe.MQeAdminMsg
- com.ibm.mqe.MQeQueueAdminMsg
- com.ibm.mqe.MQeQueueManagerAdminMsg

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value
None

Example

```
#include <hmq.h>
static const MQECHAR pHello[] = "Hello world.";
MQEHSESS   hSess;
MQEHFIELDS hMsg;
MQEINT32   rc;
MQEINT32   compcode;
MQEINT32   reason;
MQEPMO     pmo = MQEPMO_DEFAULT;
MQECHAR    * qm, *q;

qm = "aQM";
q  = "QQ";

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hMsg  = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_MSGOBJECT,
                        &compcode, &reason);
MQeFieldsPut(hSess, hMsg, "hi", MQE_TYPE_ASCII, pHello, sizeof(pHello),
             &compcode, &reason);

/* Put msg with confirmID*/

pmo.ConfirmId.hi = 0x2222;
pmo.ConfirmId.lo = 0x1111;
pmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

MQeQMgrPutMsg( hSess, qm, q, &pmo, hMsg, &compcode, &reason);

/* Confirms the message, i.e., delete it off the queue. */
MQeQMgrConfirmMsg( hSess, qm, q, MQE_QMGR_OPTION_CONFIRM_PUTMSG, hMsg,
                   &compcode, &reason);

/* Free the message handle */
MQeFieldsFree( hSess, hMsg, &compcode, &reason);
MQeTerminate( hSess, &compcode, &reason);
```

See Also

- **MQeQMgrConfirmMsg**
- **MQeQMgrGetMsg**
- **MQeQMgrUndo**

MQeQMgrUndo

Description

Undo the previous **MQeQMgrBrowseMsgs()**, or **MQeQMgrGetMsg()** or **MQeQMgrPutMsg()**, or combination of these operations on a message, or a set of messages that have the same *ConfirmID* value. If the previous operation on the message objects was **MQeQMgrBrowseMsgs()** with lock, the messages objects are unlocked and made accessible again. If the previous operation on the message objects was **MQeQMgrGetMsg()**, the message object is put back onto the queue. If the previous operation was **MQeQMgrPutMsg()**, the message object is deleted from the queue.

Syntax

```
#include <hmq.h>
MQEVOID MQeQMgrUndo( MQEHSESS hSess, MQECHAR * pQMName,
                     MQECHAR * pQName,  MQEINT64 * pConfirmId,
                     MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string containing the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEINT64 * *pConfirmId* - input
A 64 bit integer *ConfirmID* that was used on previous operations on the message objects.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output
If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_Q_NO_MSG_AVAILABLE

MQE_EXCEPT_NOT_FOUND

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEVOID

Example

```
#include <hmq.h>
static const MQECHAR pHello[] = "Hello world.";
MQEHSESS   hSess;
```

```
MQEHFIELDS hMsg;
MQEINT32   rc;
MQEINT32   compcode;
MQEINT32   reason;
MQEPMO     pmo = MQEPMO_DEFAULT;
MQECHAR    * qm, *q;

qm = "aQM";
q  = "QQ";

hSess = MQeInitialize("MyAppsName", &compcode, &reason);
hMsg  = MQeFieldsAlloc( hSess, MQE_OBJECT_TYPE_MQE_MSGOBJECT,
                        &compcode, &reason);
MQeFieldsPut(hSess, hMsg, "hi", MQE_TYPE_ASCII, pHello, sizeof(pHello),
            &compcode, &reason);

/* Put msg with confirmID, the Undo*/

pmo.ConfirmId.hi = 0x2222;
pmo.ConfirmId.lo = 0x1111;
pmo.Options     |= MQE_QMGR_OPTION_CONFIRMID;

/* Put 200 messages onto the queue. */
for (i=0; i<200; i++) {
   MQeQMgrPutMsg( hSess, qm, q, &pmo, hMsg, &compcode, &reason);
}

/* Undo the 200 putmsg operations. */
MQeQMgrUndo( hSess, qm, q, pmo.ConfirmId, &compcode, &reason);

/* Free the message handle */
MQeFieldsFree( hSess, hMsg, &compcode, &reason);
MQeTerminate( hSess, &compcode, &reason);
```

See Also

- **MQeQMgrBrowseMsgs**
- **MQeQMgrConfirmMsg**
- **MQeQMgrGetMsg**
- **MQeQMgrPutMsg**

MQeQMgrUnlockMsgs

Description

Unlock the messages on a queue identified by the *LockID* and the unique identifier (*UID*) of the message. The *LockID* is returned from an earlier , **MQeQMgrBrowseMsgs** with option MQE_QMGR_OPTION_BROWSE_LOCK.

The application programmer is responsible for calling **MQeFieldsFree** to deallocate the message handles.

Syntax

```
#include <hmq.h>
MQEINT32 MQeQMgrUnlockMsgs( MQEHSESS hSess, MQECHAR * pQMName,
                            MQECHAR * pQName,  MQEINT64 * pLockID,
                            MQEHFIELDS pMsgs[], MQEINT32 nMsgs,
                            MQEINT32 * pCompCode, MQEINT32 * pReason)
```

Parameters

MQEHSESS *hSess* - input
The session handle, returned by **MQeInitialize**.

MQECHAR * *pQMName* - input
A null terminated string containing the name of the queue manager.

MQECHAR * *pQName* - input
A null terminated string containing the name of the queue.

MQEINT64 * *pLockID* - input
The 8-bytes *LockID* that was returned by the **MQeQMgrBrowseMsgs()** call with the MQE_QMGR_OPTION_BROWSE_LOCK option specified.

This parameter must be specified.

MQEINT32 *pMsgs[]* - input
An array of message object handles to be unlocked. These message object handles should be the same ones that were returned by the **MQeQMgrBrowseMsgs()** call. The queue manager extracts the unique identifier of each message object handle and uses it with the **pLockID* value to unlock the locked message on the queue. The unique identifier of a message object is an 8-byte unique value and the string name of the origin queue manager. All other fields are ignored as they are not needed for the deletion operation.

If an entry in the *pMsgs[]* is a NULL, then this NULL entry is skipped and the unlock operation continues on to the next entry in the array. The unlock operation stops when it encounters an exception, and any remaining message object handles not processed are left as-is and remain locked on the queue.

Use the **MQeFieldsFree()** call to release MQeFields handles stored in this array.

MQEINT32 *nMsgs* - input
The number of array elements in the *pMsgs[]* array, including elements that are NULL.

MQEINT32 * *pCompCode* - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * *pReason* - output

If the returned **pCompCode* equals MQECC_ERROR, **pReason* may have any of the following values:

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

- *pLockID* is a NULL.

MQE_EXCEPT_QMGR_INVALID_QMGR_NAME

MQE_EXCEPT_QMGR_INVALID_Q_NAME

MQE_EXCEPT_QMGR_UNKNOWN_QMGR

MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST

MQE_EXCEPT_Q_NO_MSG_AVAILABLE

MQE_EXCEPT_Q_NO_MATCHING_MSG

Could not find the message on the queue, and therefore, no message is deleted.

MQE_EXCEPT_NETWORK_ERROR_OPEN|READ|WRITE

Return Value

MQEINT32

The number of the array entries successfully processed, including the NULL entries.

Example

```
#include <hmq.h>
MQEHSESS   hSess;
MQEHFIELDS hFilter = MQEHANDLE_NULL;
MQEINT32   i, n, nMsgs;
MQEINT32   compcode;
MQEINT32   reason;
MQEBMO     bmo = MQEBMO_DEFAULT;
MQEHFIELDS pMsgs[2];
MQECHAR   *qm, *q;

qm = "MyQM";
q  = "QQ";
hSess  = MQeInitialize("MyAppsName", &compcode, &reason);
nMsgs  = 2;

/* Set the browse option for lock and confirm */
bmo.Option = MQE_QMGR_BROWSE_LOCK | MQE_QMGR_CONFIRMID;
/* Set the confirm ID */
bmo.ConfirmId.hi = bmo.ConfirmId.lo = 0x12345678;

/*--------------------------------------*/
/* Browse and Unlock                    */
/*--------------------------------------*/
/* Browse nMsgs at a time until no messages are left */
while (1) { /* do forever */
   /* Browse the nMsgs matching messages */
   n = MQeQMgrBrowseMsgs( hSess, qm, q, &bmo, hFilter,
                          pMsgs, nMsgs, &compcode, &reason);

   if (n==0) {
      /* Any resources held by the cookie has been released already */
      break;
   }

   for(i=0; i<n; i++) {
```

```
            /****************************************/
            /* Process the message objects in pMsgs[] */
            /****************************************/
         }

         /* Delete the n locked messages in pMsgs[] */
         MQeQMgrUnlockMsgs( hSess, qm, q, bmo.LockId, pMsgs, n, &compcode, &reason);

         /* free pMsgs[] handle resources */
         for(i=0; i<n; i++) {
            MQeFieldsFree(hSess, pMsgs[i], &compcode, &reason);
         }
      };

      MQeTerminate(hSess, &compcode, &reason);
```

See Also

- **MQeQMgrBrowseMsgs**
- **MQeQMgrDeleteMsgs**

Chapter 11. MQExceptions and Options

MQExceptions

Completion codes

0 - MQECC_OK

-1 - MQECC_WARNING

-2 - MQECC_ERROR

Reason Codes

Sorted by Error Code

- 000 - MQE_EXCEPT_UNCODED
- 001 - MQE_EXCEPT_DEBUG
- 002 - MQE_EXCEPT_NOT_SUPPORTED
- 003 - MQE_EXCEPT_SYNTAX
- 004 - MQE_EXCEPT_TYPE
- 005 - MQE_EXCEPT_COMMAND
- 006 - MQE_EXCEPT_NOT_FOUND
- 007 - MQE_EXCEPT_DATA
- 008 - MQE_EXCEPT_BAD_REQUEST
- 009 - MQE_EXCEPT_STOPPED
- 010 - MQE_EXCEPT_CLOSED
- 011 - MQE_EXCEPT_DUPLICATE
- 012 - MQE_EXCEPT_NOT_ALLOWED
- 013 - MQE_EXCEPT_RULE
- 014 - MQE_EXCEPT_TIMEOUT
- 015 - MQE_EXCEPT_BUFFER_OVERFLOW
- 016 - MQE_EXCEPT_INVALID_HANDLE
- 017 - MQE_EXCEPT_INVALID_ARGUMENT
- 018 - MQE_EXCEPT_ALLOCATION_FAILED
- 019 - MQE_EXCEPT_FAILURE
- 020 - MQE_EXCEPT_CHNL_ATTRIBUTES
- 022 - MQE_EXCEPT_CHNL_DESTINATION
- 023 - MQE_EXCEPT_CHNL_LIMIT
- 024 - MQE_EXCEPT_CHNL_ID
- 025 - MQE_EXCEPT_CHNL_OVERRUN
- 028 - MQE_EXCEPT_CHNL_OPEN
- 040 - MQE_EXCEPT_TRNSPORT_QMGR
- 041 - MQE_EXCEPT_TRNSPORT_REQUEST
- 100 - MQE_EXCEPT_QMGR_NOT_ACTIVE
- 101 - MQE_EXCEPT_QMGR_INVALID_QMGR_NAME
- 102 - MQE_EXCEPT_QMGR_ACTIVATED

- 103 - MQE_EXCEPT_QMGR_ALREADY_EXISTS
- 104 - MQE_EXCEPT_QMGR_INVALID_Q_NAME
- 105 - MQE_EXCEPT_QMGR_Q_EXISTS
- 106 - MQE_EXCEPT_QMGR_UNKNOWN_QMGR
- 107 - MQE_EXCEPT_QMGR_Q_NOT_EMPTY
- 108 - MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST
- 109 - MQE_EXCEPT_QMGR_Q_IN_USE
- 110 - MQE_EXCEPT_QMGR_WRONG_QTYPE
- 111 - MQE_EXCEPT_QMGR_INVALID_CHANNEL
- 112 - MQE_EXCEPT_QMGR_SECURE_MSG_DECODE_FAILED
- 120 - MQE_EXCEPT_Q_NO_MSG_AVAILABLE
- 121 - MQE_EXCEPT_Q_NO_MATCHING_MSG
- 122 - MQE_EXCEPT_Q_NO_MATCHING_MSG_LISTENER
- 124 - MQE_EXCEPT_Q_INVALID_PRIORITY
- 125 - MQE_EXCEPT_Q_FULL
- 126 - MQE_EXCEPT_Q_MSG_TOO_LARGE
- 127 - MQE_EXCEPT_Q_NOT_ACTIVE
- 128 - MQE_EXCEPT_Q_ACTIVE
- 129 - MQE_EXCEPT_Q_INVALID_NAME
- 201 - MQE_EXCEPT_RAS_DIAL_FAILED
- 202 - MQE_EXCEPT_RAS_GET_PROJECTION_INFO_FAILED
- 203 - MQE_EXCEPT_RAS_HANGUP_FAILED
- 210 - MQE_EXCEPT_CONNECT_ADAPTER_NOT_ACTIVE
- 211 - MQE_EXCEPT_CONNECT_INVALID_DEFINITION
- 301 - MQE_EXCEPT_REG_NOT_FOUND
- 302 - MQE_EXCEPT_REG_NULL_NAME
- 303 - MQE_EXCEPT_REG_ALREADY_EXISTS
- 304 - MQE_EXCEPT_REG_DOES_NOT_EXIST
- 305 - MQE_EXCEPT_REG_NOT_ACTIVATED
- 306 - MQE_EXCEPT_REG_OPEN_FAILED
- 307 - MQE_EXCEPT_REG_INVALID_SESSION
- 308 - MQE_EXCEPT_REG_NOT_DEFINED
- 309 - MQE_EXCEPT_REG_INVALID_NAME
- 310 - MQE_EXCEPT_REG_LOWER_CASE
- 311 - MQE_EXCEPT_REG_ADD_FAILED
- 312 - MQE_EXCEPT_REG_DELETE_FAILED
- 313 - MQE_EXCEPT_REG_READ_FAILED
- 314 - MQE_EXCEPT_REG_UPDATE_FAILED
- 315 -MQE_EXCEPT_REG_LIST_FAILED
- 316 - MQE_EXCEPT_REG_SEARCH_FAILED
- 350 - MQE_EXCEPT_PRIVATE_REG_BAD_PIN
- 351 - MQE_EXCEPT_PRIVATE_REG_ACTIVATE_FAILED
- 352 - MQE_EXCEPT_PRIVATE_REG_NOT_OPEN
- 360 - MQE_EXCEPT_MINI_CERTREG_BAD_PIN
- 361 - MQE_EXCEPT_MINI_CERTREG_ACTIVATE_FAILED

- 362 - MQE_EXCEPT_MINI_CERTREG_NOT_OPEN
- 370 - MQE_EXCEPT_PUBLIC_REG_ACTIVATE_FAILED
- 371 - MQE_EXCEPT_PUBLIC_REG_INVALID_REQUEST
- 400 - MQE_EXCEPT_ADMIN_NOT_ADMIN_MSG
- 500 - MQE_EXCEPT_AUTHENTICATE
- 501 - MQE_EXCEPT_S_CIPHER
- 502 - MQE_EXCEPT_S_INVALID_SIGNATURE
- 503 - MQE_EXCEPT_S_CERTIFICATE_EXPIRED
- 504 - MQE_EXCEPT_S_INVALID_ATTRIBUTE
- 505 - MQE_EXCEPT_S_MINICERT_NOT_AVAILABLE
- 506 - MQE_EXCEPT_S_REGISTRY_NOT_AVAILABLE
- 507 - MQE_EXCEPT_S_BAD_INTEGRITY
- 508 - MQE_EXCEPT_S_NO_PRESET_KEY_AVAILABLE
- 509 - MQE_EXCEPT_S_MISSING_SECTION
- 600 - MQE_EXCEPT_NETWORK_ERROR
- 601 - MQE_EXCEPT_NETWORK_ERROR_OPEN
- 602 - MQE_EXCEPT_NETWORK_ERROR_READ
- 603 - MQE_EXCEPT_NETWORK_ERROR_WRITE
- 700 - MQE_EXCEPT_EOF
- 701 - MQE_EXCEPT_NON_MQE_SYSTEM_EXCEPTION
- 2000 - MQE_EXCEPT_PLATFORM_LIB_LOAD_FAILED
- 2001 - MQE_EXCEPT_PLATFORM_LIB_STILL_OPEN
- 2500 - MQE_WARN_PLATFORM_LIB_ALREADYOPEN
- 2501 - MQE_WARN_SESSION_DELETED
- 2502 - MQE_WARN_FIELDS_DATA_TRUNCATED
- 3001 - MQE_EXCEPT_ADMIN_UNKNOWN_CHARACTERISTIC
- 3002 - MQE_EXCEPT_ADMIN_UNDEFINED_ACTION

Sorted by Exception

- MQE_EXCEPT_ADAPTER_HTTP_ERROR - 4000
- MQE_EXCEPT_ADMIN_NOT_ADMIN_MSG - 400
- MQE_EXCEPT_ADMIN_UNKNOWN_CHARACTERISTIC - 3001
- MQE_EXCEPT_ADMIN_UNDEFINED_ACTION - 3002
- MQE_EXCEPT_ALLOCATION_FAILED - 003
- MQE_EXCEPT_AUTHENTICATE - 500
- MQE_EXCEPT_BAD_REQUEST - 008
- MQE_EXCEPT_BUFFER_OVERFLOW - 015
- MQE_EXCEPT_CHNL_ATTRIBUTES - 020
- MQE_EXCEPT_CHNL_DESTINATION - 022
- MQE_EXCEPT_CHNL_ID - 023
- MQE_EXCEPT_CHNL_LIMIT - 024
- MQE_EXCEPT_CHNL_OPEN - 028
- MQE_EXCEPT_CHNL_OVERRUN - 025
- MQE_EXCEPT_CLOSED - 010
- MQE_EXCEPT_COMMAND - 005

- MQE_EXCEPT_CONNECT_ADAPTER_NOT_ACTIVE - 210
- MQE_EXCEPT_CONNECT_INVALID_DEFINITION - 211
- MQE_EXCEPT_DATA - 007
- MQE_EXCEPT_DEBUG - 001
- MQE_EXCEPT_DUPLICATE - 011
- MQE_EXCEPT_EOF - 700
- MQE_EXCEPT_FAILURE - 019
- MQE_EXCEPT_INVALID_ARGUMENT - 017
- MQE_EXCEPT_INVALID_HANDLE - 017
- MQE_EXCEPT_MINI_CERTREG_ACTIVATE_FAILED - 361
- MQE_EXCEPT_MINI_CERTREG_BAD_PIN - 360
- MQE_EXCEPT_MINI_CERTREG_NOT_OPEN - 362
- MQE_EXCEPT_NETWORK_ERROR - 600
- MQE_EXCEPT_NETWORK_ERROR_OPEN - 601
- MQE_EXCEPT_NETWORK_ERROR_READ - 602
- MQE_EXCEPT_NETWORK_ERROR_WRITE - 603
- MQE_EXCEPT_NON_MQE_SYSTEM_EXCEPTION - 701
- MQE_EXCEPT_NOT_ALLOWED - 012
- MQE_EXCEPT_NOT_FOUND - 006
- MQE_EXCEPT_NOT_SUPPORTED - 002
- MQE_EXCEPT_PLATFORM_LIB_LOAD_FAILED - 2000
- MQE_EXCEPT_PLATFORM_LIB_STILL_OPEN - 2001
- MQE_EXCEPT_PRIVATE_REG_ACTIVATE_FAILED - 351
- MQE_EXCEPT_PRIVATE_REG_BAD_PIN - 350
- MQE_EXCEPT_PRIVATE_REG_NOT_OPEN - 352
- MQE_EXCEPT_PUBLIC_REG_ACTIVATE_FAILED - 370
- MQE_EXCEPT_PUBLIC_REG_INVALID_REQUEST - 371
- MQE_EXCEPT_QMGR_ACTIVATED - 102
- MQE_EXCEPT_QMGR_ALREADY_EXISTS - 103
- MQE_EXCEPT_QMGR_INVALID_CHANNEL - 111
- MQE_EXCEPT_QMGR_INVALID_QMGR_NAME - 101
- MQE_EXCEPT_QMGR_INVALID_Q_NAME - 104
- MQE_EXCEPT_QMGR_NOT_ACTIVE - 100
- MQE_EXCEPT_QMGR_Q_DOES_NOT_EXIST - 108
- MQE_EXCEPT_QMGR_Q_EXISTS - 105
- MQE_EXCEPT_QMGR_Q_IN_USE - 109
- MQE_EXCEPT_QMGR_Q_NOT_EMPTY - 107
- MQE_EXCEPT_QMGR_SECURE_MSG_DECODE_FAILED - 112
- MQE_EXCEPT_QMGR_UNKNOWN_QMGR - 106
- MQE_EXCEPT_QMGR_WRONG_QTYPE - 110
- MQE_EXCEPT_Q_ACTIVE - 128
- MQE_EXCEPT_Q_FULL - 125
- MQE_EXCEPT_Q_INVALID_NAME - 129
- MQE_EXCEPT_Q_INVALID_PRIORITY - 124
- MQE_EXCEPT_Q_MSG_TOO_LARGE - 126

- MQE_EXCEPT_Q_NOT_ACTIVE - 127
- MQE_EXCEPT_Q_NO_MATCHING_MSG - 121
- MQE_EXCEPT_Q_NO_MATCHING_MSG_LISTENER - 122
- MQE_EXCEPT_Q_NO_MSG_AVAILABLE - 120
- MQE_EXCEPT_RAS_DIAL_FAILED - 201
- MQE_EXCEPT_RAS_GET_PROJECTION_INFO_FAILED - 202
- MQE_EXCEPT_RAS_HANGUP_FAILED - 203
- MQE_EXCEPT_REG_ADD_FAILED - 311
- MQE_EXCEPT_REG_ALREADY_EXISTS - 303
- MQE_EXCEPT_REG_DELETE_FAILED - 312
- MQE_EXCEPT_REG_DOES_NOT_EXIST - 304
- MQE_EXCEPT_REG_INVALID_NAME - 309
- MQE_EXCEPT_REG_INVALID_SESSION - 307
- MQE_EXCEPT_REG_LIST_FAILED - 315
- MQE_EXCEPT_REG_LOWER_CASE - 310
- MQE_EXCEPT_REG_NOT_ACTIVATED - 305
- MQE_EXCEPT_REG_NOT_DEFINED - 308
- MQE_EXCEPT_REG_NOT_FOUND - 301
- MQE_EXCEPT_REG_NULL_NAME - 302
- MQE_EXCEPT_REG_OPEN_FAILED - 306
- MQE_EXCEPT_REG_READ_FAILED - 313
- MQE_EXCEPT_REG_SEARCH_FAILED - 316
- MQE_EXCEPT_REG_UPDATE_FAILED - 314
- MQE_EXCEPT_RULE - 013
- MQE_EXCEPT_STOPPED - 009
- MQE_EXCEPT_SYNTAX - 003
- MQE_EXCEPT_S_BAD_INTEGRITY - 507
- MQE_EXCEPT_S_CERTIFICATE_EXPIRED - 503
- MQE_EXCEPT_S_CIPHER - 501
- MQE_EXCEPT_S_INVALID_ATTRIBUTE - 504
- MQE_EXCEPT_S_INVALID_SIGNATURE - 502
- MQE_EXCEPT_S_MINICERT_NOT_AVAILABLE - 505
- MQE_EXCEPT_S_MISSING_SECTION - 509
- MQE_EXCEPT_S_NO_PRESET_KEY_AVAILABLE - 508
- MQE_EXCEPT_S_REGISTRY_NOT_AVAILABLE - 506
- MQE_EXCEPT_TIMEOUT - 014
- MQE_EXCEPT_TRNSPORT_QMGR - 040
- MQE_EXCEPT_TRNSPORT_REQUEST - 041
- MQE_EXCEPT_TYPE - 004
- MQE_EXCEPT_UNCODED - 000
- MQE_WARN_PLATFORM_LIB_ALREADYOPEN - 2500
- MQE_WARN_SESSION_DELETED - 2501
- MQE_WARN_FIELDS_DATA_TRUNCATED - 2502

MQe options

MQeFields options

- 0 - MQE_FIELDS_OPTION_NONE
- 1 - MQE_FIELDS_OPTION_ALL_FIELDS
- 2 - MQE_FIELDS_OPTION_REPLACE

MQeQMgr options

- 0x00000000 - MQE_QMGR_OPTION_NONE
- 0x00000000 - MQE_QMGR_OPTION_PUT_DEFAULT
- 0x0000000F - MQE_QMGR_OPTION_PUT_MASK
- 0x00000001 - MQE_QMGR_OPTION_PUT_ASYNCHRONOUS
- 0x00000002 - MQE_QMGR_OPTION_PUT_SYNCHRONOUS
- 0x00000010 - MQE_QMGR_OPTION_BROWSE_LOCK
- 0x00000020 - MQE_QMGR_OPTION_BROWSE_JUST_UID
- 0x00000100 - MQE_QMGR_OPTION_CONFIRMID
- 0x00000300 - MQE_QMGR_OPTION_CONFIRM_GETMSG
- 0x00000500 - MQE_QMGR_OPTION_CONFIRM_PUTMSG

MQeTrace options

MQeTrace Commands

- 1 - MQE_TRACE_CMD_START
- 2 - MQE_TRACE_CMD_STOP
- 3 - MQE_TRACE_CMD_SET_MASK
- 4 - MQE_TRACE_CMD_SET_HANDLER

MQeTrace Options

- 0x0001 - MQE_TRACE_OPTION_APP_MSG
- 0x0002 - MQE_TRACE_OPTION_APP_INFO
- 0x0004 - MQE_TRACE_OPTION_APP_WARNING
- 0x0008 - MQE_TRACE_OPTION_APP_ERROR
- 0x0010 - MQE_TRACE_OPTION_APP_DEBUG
- 0x001F - MQE_TRACE_OPTION_APP_ALL
- 0x0100 - MQE_TRACE_OPTION_SYS_MSG
- 0x0200 - MQE_TRACE_OPTION_SYS_INFO
- 0x0400 - MQE_TRACE_OPTION_SYS_WARNING
- 0x0800 - MQE_TRACE_OPTION_SYS_ERROR
- 0x1000 - MQE_TRACE_OPTION_SYS_DEBUG
- 0x1F00 - MQE_TRACE_OPTION_SYS_ALL

Appendix A. Trap numbers for functions in shared libraries

This section lists the trap numbers for the published functions built into the `hmqLib.prc` shared library. These traps expect the library number as the first function parameter, followed by the parameters as published in the API interfaces. It is also important to note that helper functions published in `hmqHelper.h` are not in this shared library.

Table 1 lists the trap numbers for functions in the `hmqLib.prc` and Table 2 lists the trap numbers for functions in the `hmqFields.prc`.

Table 5. Trap numbers for hmqLib.prc shared library functions

Function	Trap number
MQeQMgrBrowseMsgs	11
MQeQMgrDeleteMsgs	12
MQeQMgrGetMsg	12
MQeQMgrGetName	14
MQeQMgrGetQueueList	15
MQeQMgrPutMsg	16
MQeQMgrUnlockMsgs	18
MQeQMgrConfirmMsg	20
MQeQMgrUndo	21
MQeTrace	23
MQeTraceCmd	24

Table 6. Trap numbers in hmqFieldsLib.prc shared library functions

Function	Trap number
MQeFieldsAlloc	25
MQeFieldsDelete	26
MQeFieldsDump	27
MQeFieldsDumpLength	28
MQeFieldsEquals	29
MQeFieldsFields	30
MQeFieldsFree	31
MQeFieldsGet	32
MQeFieldsGetArray	33
MQeFieldsGetByArrayOfFd	34
MQeFieldsGetByIndex	35
MQeFieldsGetByStruct	36
MQeFieldsHide	37
MQeFieldsPut	38
MQeFieldsPutArray	39
MQeFieldsPutByArrayOfFd	40

Trap numbers for functions in shared libraries

Table 6. Trap numbers in hmqFieldsLib.prc shared library functions (continued)

Function	Trap number
MQeFieldsPutByStruct	41
MQeFieldsRead	42
MQeFieldsRestore	43
MQeFieldsWrite	45
MQeFieldsType	46

Appendix B. 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.

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

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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.

trademarks

Windows and Windows NT are registered trademark of Microsoft Corporation in the United States and other countries.

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

Glossary

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is used in this book.

If you do not find the term you are looking for, see the index or the *IBM Dictionary of Computing*, New York:. McGraw-Hill, 1994.

Application Programming Interface (API). An Application Programming Interface consists of the functions and variables that programmers are allowed to use in their applications.

asynchronous messaging. A method of communicating between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

authenticator. A program that checks that verifies the senders and receivers of messages.

bridge. An MQSeries Everyplace object that allows messages to flow between MQSeries Everyplace and other messaging systems, including WebSphere MQ.

channel. See *dynamic channel* and *MQI channell*.

channel manager. An MQSeries Everyplace object that supports logical multiple concurrent communication pipes between end points.

class. A class is an encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

client. In WebSphere MQ, a client is a run-time component that provides access to queuing services on a server for local user applications.

compressor. A program that compacts a message to reduce the volume of data to be transmitted.

cryptor. A program that encrypts a message to provide security during transmission.

dynamic channel. A dynamic channel connects MQSeries Everyplace devices and transfers synchronous and asynchronous messages and responses in a bidirectional manner.

encapsulation. Encapsulation is an object-oriented programming technique that makes an object's data private or protected and allows programmers to access and manipulate the data only through method calls.

gateway. An MQSeries Everyplace gateway (or server) is a computer running the MQSeries Everyplace code including a channel manager.

Hypertext Markup Language (HTML). A language used to define information that is to be displayed on the World Wide Web.

instance. An instance is an object. When a class is instantiated to produce an object, we say that the object is an instance of the class.

interface. An interface is a class that contains only abstract methods and no instance variables. An interface provides a common set of methods that can be implemented by subclasses of a number of different classes.

Internet. The Internet is a cooperative public network of shared information. Physically, the Internet uses a subset of the total resources of all the currently existing public telecommunication networks. Technically, what distinguishes the Internet as a cooperative public network is its use of a set of protocols called TCP/IP (Transport Control Protocol/Internet Protocol).

Java Developers Kit (JDK). A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Naming and Directory Service (JNDI). An API specified in the Java programming language. It provides naming and directory functions to applications written in the Java programming language.

Lightweight Directory Access Protocol (LDAP). LDAP is a client-server protocol for accessing a directory service.

message. In message queuing applications, a message is a communication sent between programs.

message queue. See queue

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

method. Method is the object-oriented programming term for a function or procedure.

MQI channel. An MQI channel connects an WebSphere MQ client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

WebSphere MQ. WebSphere MQ is a family of IBM licensed programs that provide message queuing services.

object. (1) In Java, an object is an instance of a class. A class models a group of things; an object models a particular member of that group. (2) In WebSphere MQ, an object is a queue manager, a queue, or a channel.

package. A package in Java is a way of giving a piece of Java code access to a specific set of classes. Java code that is part of a particular package has access to all the classes in the package and to all non-private methods and fields in the classes.

personal digital addistant (PDA). A pocket sized personal computer.

private. A private field is not visible outside its own class.

protected. A protected field is visible only within its own class, within a subclass, or within packages of which the class is a part

public. A public class or interface is visible everywhere. A public method or variable is visible everywhere that its class is visible

queue. A queue is an WebSphere MQ object. Message queueing applications can put messages on, and get messages from, a queue

queue manager. A queue manager is a system program the provides message queuing services to applications.

server. (1) An MQSeries Everyplace server is a device that has an MQSeries Everyplace channel manager configured. (2) An WebSphere MQ server is a queue manager that provides message queuing services to client applications running on a remote workstation. (3) More generally, a server is a program that responds to requests for information in the particular two-program information flow model of client/server. (3) The computer on which a server program runs.

servlet. A Java program which is designed to run only on a web server.

subclass. A subclass is a class that extends another. The subclass inherits the public and protected methods and variables of its superclass.

superclass. A superclass is a class that is extended by some other class. The superclass's public and protected methods and variables are available to the subclass.

synchronous messaging. A method of communicating between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing . Contrast with *asynchronous messaging*.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Web. See World Wide Web.

Web browser. A program that formats and displays information that is distributed on the World Wide Web.

World Wide Web (Web). The World Wide Web is an Internet service, based on a common set of protocols, which allows a particularly configured server computer to distribute documents across the Internet in a standard way.

Bibliography

Related publications:

- *MQSeries Everyplace for Multiplatforms, Introduction*, SC34-6183-00
- *MQSeries Everyplace for Multiplatforms, Programming Reference*, SC34-6185-00
- *MQSeries Everyplace for Multiplatforms, Programming Guide*, SC34-6184-00
- *WebSphere MQ An Introduction to Messaging and Queuing*, GC33-0805-01
- *WebSphere MQ for Windows NT V5R1 Quick Beginnings*, GC34-5389-00

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

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 request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should 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 44–1962–870229
  - From within the U.K., use 01962–870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in U.S.A.

SC34-6187-00