MQSeries® Everyplace

Native Client Information

Version 1.0

GC34-5883-00

MQSeries® Everyplace

Native Client Information

Version 1.0

GC34-5883-00

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix. Notices" on page 171

Licence warning

MQSeries Everyplace Version 1.0 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 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.*

First Edition (June 2000)

This edition applies to MQSeries Everyplace Version 1 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/ts/mqseries/library/`.

Contents

About this book

This book is a programming guide for the MQSeries Everyplace product and contains information on how to use the MQSeries Everyplace C APIs. Guidance is provided to help with writing C programs to perform common messaging tasks, and 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:

1. Getting started information for specific devices
2. General programming guidance for the native client
3. Reference information for the native client

This book is intended to be used in conjunction with

- *MQSeries Everyplace Introduction* — a detailed description of the capabilities of MQSeries Everyplace,
- *MQSeries Everyplace Programming Guide* — guidance for programming with MQSeries Everyplace
- *MQSeries Everyplace 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 MQSeries library. This can be reached from the MQSeries Web site, URL address `http://www.ibm.com/software/ts/MQSeries/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 wanting to write C based MQSeries Everyplace programs to exchange secure messages between MQSeries Everyplace systems and other members of the MQSeries family of messaging and queueing products.

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

Prerequisite knowledge

It is assumed that the reader has a working knowledge of C programming techniques, and an understanding of MQSeries Everyplace as described in *MQSeries Everyplace 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 MQSeries books:

- *MQSeries An Introduction to Messaging and Queuing*

This book is available in softcopy form from Book section of the online MQSeries library. This can be reached from the MQSeries Web site, URL address `http://www.ibm.com/software/ts/MQSeries/library/`

Introduction

The MQSeries Everyplace C application programming interface (API) is an MQSeries 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 MQSeries family, extending the reach of MQSeries 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. Using 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 1.0 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 PalmOS device, such as Palm V, IBM Workpad C3, to an MQSeries Everyplace Java server.

Prerequisites

This information assumes the following environment:

- A PalmOS device with PalmOS version 2.0 or later
- A cradle for the device including a serial connection to a PC or laptop.
- A Microsoft Windows NT 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 command) or the MQSeries Everyplace Java server code .

Overview

The following sections explain how to:

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

Creating and compiling a basic Palm program that utilizes MQSeries messaging

Note: An example application and its source code is included in the support pack MAE1. This program can be used in place of a user written program.

To create a program, use the following procedure:

1. Start Metrowerks Codewarrior
2. Create a new project using the File menu.
3. A dialog box appears, prompting to 'Select Project Stationary' - click on the '+' next to Palm OS to expand the group. Choose 'Palm OS C App' and click OK.
4. In the dialog box that is displayed, choose an appropriate directory for the project folder and give the project a name, such as 'BasicApp' and click OK..

A new folder and a set of source files within that folder are created. The folder has the same name as the project, and the project file within the folder has a name extension of .mcp. There is also a 'src' folder that contains files named Starter.c (source code) and a Starter.rsrc (form resource).

5. In Codewarrior, a project window is opened. Expanding the AppSource and AppResources folders shows the Starter.c and Starter.rsrc files.

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

6. Edit the Starter.c file in the following way (the text in italics needs to be added):

```
#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, , );

    /* 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 starts a session to the local queue manager, creates an MQeFields message object, puts some data into the message object, and puts the message to a queue manager and queue with the names 'aQMgrName' and 'aQueueName' respectively. Change these names to those of the queue manager and queue that are to be used. (If a separate MQSeries Everyplace server is to be started, the queue manager name is 'ExampleQM' and the queue name is 'SYSTEM.DEFAULT.LOCAL.QUEUE'.) When the message put is complete, the message object is freed and the session is terminated

When the edit of the `Starter.c` file is complete, save it and close the edit window.

7. Link in the MQSeries Everyplace stub library as follows:
   a. Select the project window
   b. Choose the Project menu and select Add Files
   c. In the file dialog that appears change the 'Files of type' filter to 'Library files' and navigate to the `hmq.lib` file.
   d. Select `hmq.lib` and click Add.

   A project message appears confirming that an access path has been added.
8. Make sure that the compiler knows where to find the MQSeries Everyplace header file `hmq.h` as follows:
   a. Either click on the Settings button or choose the Edit menu and select Starter Settings. The Starter Settings dialog is displayed.
   b. Choose Access Paths from the Target group on the left of the dialog box.
   c. Choose the System Paths radio button and click Add. The 'Please Select An Access Path' dialog box is displayed.
   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. Click on the Save button of the Starter Settings dialog and then close the dialog.
9. Compile the program by either clicking on the Make button or choosing the Make item from the Project menu. The project builds a compiled program called `Starter.prc` in the project folder.

HotSyncing the program and MQSeries Everyplace files onto the PalmOS device

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.

   Both these files must be installed on the device.
2. **GUI program `hmqIni.prc`**

   This GUI program hooks into the Palm preference panel. It enables the user to manually configure the system parameters needed to run the MQSeries Everyplace system.

   This program must be installed on the device.
3. **Stub library `hmq.lib`**

   A small stub library that must be linked with an application program to use the MQSeries Everyplace system. This 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, programmers should be aware that using any of these helper functions increases the application code size by about 6KB.

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

   The first include file must be included in all applications, and the second one is needed if the helper functions are being used.

Installing the files on the Palm device

1. Start the Palm HotSync manager and the Palm Install Tool
2. Click the Add button and navigate to the project folder
3. Choose the `Starter.prc` file and click Open. The file is added to the list of files to install.

   If the example application (palmos-example) is to be used, add the `MQeExample.prc` file found in the palmos-example folder instead of `Starter.prc`.
4. Repeat the process 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. Press the HotSync button on the cradle (or use the HotSync program on the Palm).

   The files are installed on the PalmOS device.

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

On the PC:

1. Go to the Windows Control Panel and start the Network settings dialog.
2. Choose the Services tab and check to see if Remote Access Service (RAS) is already installed in the Network Services list.

   If it 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 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. Go to "Install RAS" on page 7.

If this modem is not installed,

1. Click on the Add button to start the 'Install New Modem' wizard.
2. Check the 'Don't Detect My Modem' checkbox and click on Next.
3. Choose the '(Standard Modem Types)' Manufacturer and the 'Dial-Up Networking Serial Cable between 2 PCs' Model, and click on Next. A Port selection dialog is displayed.

4. Choose the port that the Palm cradle is plugged into and click Next to install the modem.
5. Click Finish to close the wizard . The new modem is added to the list.
6. Click Close to close the Modems dialog.

Install RAS

To install RAS:

1. Start the Network dialog from the Control Panel.
2. Choose the Services tab and click on the Add button. In the dialog that is displayed, choose Remote Access Service and click OK. A Windows dialog asks for the location of some Windows NT files - these are probably 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 have installed, the Remote Access Setup dialog and the Add RAS Device dialog are displayed.
3. Choose the 'Dial-Up Networking Serial Cable Between 2 PCs' modem in this dialog and click OK.
4. On the Remote Access Setup dialog, click the Configure button to display a Configure Port Usage dialog
5. Select the 'Dial out and Receive calls' radio button and click OK.
6. Click Continue to start the RAS set up.
7. Click Close to close the Network dialog
8. Restart the PC when prompted to do so.

Create a new user

To create a new user 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, choose User Manager.
2. Add a new user by selecting the User menu and choosing New User.
3. Give the new user an appropriate username, such as 'palmuser', and give them a basic password, such as 'mqe'.
4. Uncheck the 'User Must Change Password at Next Logon' and check the 'User Cannot Change Password' and 'Password Never Expires' checkboxes.
5. Click the Dialin button
6. Check the 'Grant dialin permission to user' checkbox . 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. Go to the Administrative Tools folder of the Start Menu and start the Remote Access Admin application.
2. Make sure that the HotSync manager is no longer running (as this uses the same port as RAS) by checking for the blue and red HotSync icon in the system tray (on the taskbar) - if it is there, right click on it and choose Exit.
3. Select the Server menu in the Remote Access Admin application and choose '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 the PalmOS device

1. Start the Prefs application on the PalmOS device .
2. Choose the Network screen
3. Open the menu and choose New from the Service menu.
4. Name the service 'Windows RAS' or similar and put in the username and password that were set earlier on the PC.
5. Tap in the Phone area and put in the phone number 00 - this signifies that no number should be phoned.
6. Tap the Details button.
7. Set Connection type 'PPP', Idle time-out to 'Power Off' and check Query DNS and IP Address: Automatic.
8. Tap the Script button and enter the following script:

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

9. Tap on OK
10. With the device in its cradle and RAS running on the PC, tap the Connect button to connect to RAS.
11. When the connection is made, tap the Disconnect button to disconnect from RAS.
12. To configure MQSeries Everyplace on the Palm, open the preference panel.
13. Pull down a list of preferences and go to the menu item "IBM" to display the "IBM" preference panel. The text in this panel looks something like this

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

    This is the information needed to enable the MQSeries Everyplace queue manager on the Palm to make a connection to an MQSeries Everyplace server and perform the fundamental PutMsg and GetMsg() operations. The order of the above entries is not important.

    - The entry `QMgr.Name.Local=LocalQM` must be defined in order to use MQeInitialize() and start a session with the MQSeries Everyplace queue manager.

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

      Instead of the IP address, the host name, `abc.com` for example, can be used.
    - The third entry is the connection command definition. For the HTTP 1.0 protocol this command is inserted into every POST command. `?Channel` is the default command that is recognized by the MQSeries Everyplace HTTP server. This command with can be replaced with a servlet name to communicate with an 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 is being used.
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, check all the checkboxes.
5. In the View menu, select the System.Err command, so that all trace messages can be seen.

Run the Palm program on the PalmOS device

1. With RAS and the AWT MQSeries Everyplace server running, place the palm in its cradle.
2. From the Applications screen, tap on the Starter application.

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

Palm - getting started

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 APIs without exception 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 would test the Completion Code for an error or warning value `MQECC_ERROR` or `MQECC_WARNING` and take appropriate action (which would involve 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 the *fields* object. A fields object is a generic container of one or more fields, and each field is a name-value pair. There are also special classes of fields object that contain predefined fields. For example, the MQSeries Everyplace message object is a fields object that MQeQMgrPutMsg() accepts and that MQeQMgrGetMsg() and MQeQMgrBrowseMsgs() return. Each fields 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 fields object can be used to build and organize data in a hierarchical manner. A set of related name-value fields can be put into a fields object, that is then put into another fields object that is in turn put into a message object for sending.

A *filter* is a fields 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 the 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

Since a message object is a fields object, its construction is fundamentally the same. Both fields and message objects are constructed by calling the MQeFieldsAlloc API, but the *Type* parameter is used to specify whether a fields or message object is created. This 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 fields object that is no longer required should be destroyed to free resources back to the operating system. The MQeFieldsFree API is provided to destroy fields 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 fields 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 (such as a Session Handle or MQeFieldsHandle) and has subsequently been deleted (by MQeTerminate() for a Session Handle or by MQeFieldsFree() for a fields handle) so the handle is no longer valid.

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 Fields APIs” on page 27 section of this document.

Every piece of data put into the fields 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 fields object regardless of the format of the host system.

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

Table 1. Fields 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)
MQE_TYPE_SHORT	2 bytes (16 bits)
MQE_TYPE_INT	4 bytes (32 bits)

Table 1. Fields object data types (continued)

Type	Data Representation
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 a message 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 only 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 version 1.0.

The `ConfirmId` option is used to implement assured message delivery between the MQSeries Everyplace client and the server. When this option is specified with MQeQMgrPutMsg, the message is put onto the queue, but it is not made accessible until an MQeQMgrConfirmMsg API is called on the message object. The application issues an MQeQMgrConfirmMsg call only after the MQeQMgrPutMsg has successfully returned. Should the communication link fail during an MQeQMgrPutMsg call, when connection with the MQSeries Everyplace server is reestablished, the application first calls the MQeQMgrUndo API to remove the message that may or may not have been put on the queue in the first place. It then calls the 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);
```

putting messages onto a queue

```
/* 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 the MQeQMgrPutMsg(), each also supports the `ConfirmId` option. MQeQMgrBrowseMsgs also has a `Browse_Lock` option. One major difference between these two APIs is that the 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 out of a fields object can be given to MQeQMgrGetMsg(), so that the first message on the queue that matches the fields specified in the filter is returned.

MQeQMgrGetMsg() with ConfirmId

The message object is returned to the caller, but, unlike the previous case it is not deleted from the queue, however, it is made inaccessible to subsequent MQeQMgrBrowseMsgs and MQeQMgrGetMsg calls except an MQeQMgrGetMsg or an MQeQMgrDeleteMsgs() with input parameters containing the UID of the inaccessible message object. An MQeQMgrConfirmMsg() deletes the message from the queue, and an MQeQMgrUndo() makes the message object reappear on the queue again. 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 a fields object can be given to this function call, so that only the messages that match the fields specified in the filter are returned. Again, 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` . This lockID and the 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 also be specified with this option.

Messages locked on the queue are inaccessible to future MQeQMgrGetMsg() and MQeQMgrBrowseMsgs() operations, except an MQeQMgrGetMsg() with a filter that contains the lockID field, or an MQeQMgrDeleteMsgs() with the UID as its input parameter.

A filter can also be specified with this option.

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 also be specified with this option.

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 a message object

Use the following fields 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 Fields APIs" on page 27 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 the pointer to the memory which receives 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, NOT the same as the number of bytes. For example, with a field that has a single element of datatype MQE_TYPE_INT this call would return 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 receives the data. The data should be 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);
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", NULL, &buf, n, NULL, &compcode, &reason);
```

Chapter 7. Advanced Fields APIs

Three sets of advanced Fields APIs are provided for experienced programmers who want to put and get data in and out of the fields object more efficiently. 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 a fields object. Instead of doing individual MQeFieldsGet and MQeFieldsPut 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. So if used properly, this API improves the performance of the Fields 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 the application programmer to map a C data structure in the application program directly to a set of fields in the fields 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 a fields 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..7]=0xABCD1244-00000000" */
```

MQeFieldsRead() and MQeFieldsWrite()

These APIs enable the application to stream data in and out of a field in a fields 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 Fields 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.

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. This configuration can be achieved by putting a connection administration message (MQeConnectionAdminMsg) to the administration queue of the local (client) queue manager. 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. Consult the *MQSeries Everyplace 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 &lt;hmq.h&gt;
/* 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 = MQstrlen(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 = MQstrlen(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 = MQstrlen(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 = &amp;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 = &amp;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 = MQstrlen(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,&res,&reason);
    MQeFieldsFree(hSess,cam,&cc,&reason);
  }
  return res;
}
```

administration using administration messages

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 that can be used to create MQSeries Everyplace programs.

C language data types

In this section, the platform-independent primitive data types that are used throughout the C native APIs of MQSeries Everyplace are described. The descriptions include the size of the data type and its alignment requirement.

Primitive

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), and the opposite is true on a big-endian 68k based machine. The data on a transmission medium is always network-ordered, that is big endian.

MQSeries Everyplace Fields Data Types

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

Primitives

The fields 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 a fields object, and use MQeFieldsGet() to get a **copy** of a field from a fields object. Use MQeFieldsDelete() to remove a field from the field object.

General constraint

With MQSeries Everyplace Version 1.0 on a PalmOS device, the maximum number of fields object handles is limited to 13, and the maximum fields object (message object) size is 12 KB.

Array APIs

Two sets of APIs are defined to encode the arrays. One set of APIs starts with **MQeFieldsArrayOf*** and the other has **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/or dynamically adjust the array size.

The encoding for each element in the array to a field is accomplished by first creating a unique field name for each element by concatenating the field name of the array with an index, and then setting the data type of the field elements to the data type of the array.

•

Field name
Concatenation of the field name of the array; a separator character (which is a colon, ″:″), and the index of the elements. In other words, if the field name of the array is ″foo″, then the field name for the first, second elements and the nth elements are ″foo:0″, ″foo:1″ and ″foo:n-1″, respectively.

Field type
Same 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()	Allocates a new fields object and returns a handle to it.
MQeFieldsDelete()	Delete an existing field in the fields object.
MQeFieldsDump()	Serialize the internal name/value pair fields into a byte array for storage or communication.
MQeFieldsDumpLength()	Get the total number of byte needed to hold the serialized fields in the fields object.
MQeFieldsEquals()	Compares two fields object and determines if they are the same.
MQeFieldsFields()	Returns the number of fields in the fields object.

Table 2. MQeFields base API (continued)

API	Description
MQeFieldsFree()	Deallocates a fields object and recovers its resources.
MQeFieldsGet()	Given a field name, returns 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, returns the field at index.
MQeFieldsGetByStruct()	Given a data structure and its fields structure descriptor, populate the data structure with the fields.
MQeFieldsHide()	Exclude a field from field comparison API, MQeFieldsEquals()
MQeFieldsPut()	Put a field into the fields 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 the byte array into name/value pair fields and store them in the fields object.
MQeFieldsType()	Extract the object type of the fields 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 reference. These APIs are built on top of the APIs listed in Table 2 on page 42.

Table 3. MQeFields macros and helper APIs

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

MQeFields APIs

Table 3. MQeFields macros and helper APIs (continued)

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

Table 3. MQeFields macros and helper APIs (continued)

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

Pointer to the null terminated string name of the field. Application programs should follow 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 fd_namelen

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` for the field for operations that retrieve the name of a field. These sizes do not include a terminating NULL.

MQEBYTE fd_type
Data type of the field data.

MQEBYTE * fd_data
Pointer to data.

MQEINT32 fd_datalen
Number of data elements (not bytes) in `fd_data` . The input values of `fd_datalen` and `fd_datatype` specifies 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
Platform specific base pointer for data, should be NULL unless specifically being used.

MQeField structure descriptor

The field structure descriptor holds information about a field to be added to or retrieved from a fields 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
Pointer to the null terminated string name of the field. Application programs should follow 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
Length of the `sd_name` .

MQEBYTE sd_datatype
Data type of the field data.

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

MQEINT32 sd_dataoff
Offset of the element to get or put.

MQEBYTE * sd_datalen
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 pointer to length, not number of elements
MQSTRUCT_DATA	0x2	struct offset sd_dataoff is 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 fields 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 fields object and returns a handle to it. The handle represents a fields object. It must be specified on all subsequent fields accessing calls issued by the application. 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 fields object that is used for restore, MQeFieldsRestore.

"**com.ibm.mqe.MQeFields**"

The base fields object type

"**com.ibm.mqe.MQeMsgObject**"

A field object with two additional fields, 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**"

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 MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEHFIELDS `hFlds`

Handle to a fields object. If any error occurs during the allocation, then a MQEHANDLE_NULL is returned.

Implementation

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

Example

MQeFieldsAlloc

```
#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 fields object.

Given a field name, remove its associated field from the fields 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 a fields 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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_NOT_FOUND

The field was not found in the fields 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 a fields object into a byte array.

This functions supports partial dumps. Between partial dumps, the programmer should not add or delete any field in the fields 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 fields 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 a fields object.

MQEINT32 `srcOff` - input
Offset into internal byte array representation of the fields object at which the dump should start

MQEBYTE `pBuf[]` - output
Buffer to hold the dumped bytes

MQEINT32 `BufLen` - input
Number of bytes to dump

MQEVOID `*pbase` - input
base pointer for output buffer `pBuf`

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

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 network ordered, the big-endian order.

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 * textVal  = "The Owl and the Pussy Cat went to sea.";
static   MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44 };
#define  MAX_CHUNK_SIZE 64
MQEHSESS   hSess;
MQEHFIELDS hFlds;
MQECHAR *  pname;
MQEBYTE    datatype;
MQEINT32   n;
MQEBYTE *  pdata;
MQEINT32   compcode;
MQEINT32   reason;
MQEINT32   int32Val;
MQEINT32   i, offset;
MQEINT32   chunk, nchunks;
MQEBYTE ** buf_array;
MQEINT32   nbytes;
MQEINT32   rc;
MQEVOID *  base;

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

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

base    = (MQEVOID *)0;
offset  = 0;
i       = 0;

/*
* Calc number of chunks needed to hold the dump byte array
*/
nchunks = (nbytes + MAX_CHUNK_SIZE -1)/MAX_CHUNK_SIZE;
chunk = (nbytes + MAX_CHUNK_SIZE -1)%MAX_CHUNK_SIZE;

/*
* Allocate the buf array.
*/
buf_array = (MQEBYTE**) malloc(nchunks * sizeof(MQEBYTE*));

while (nchunks) {
   buf_array[i] = (MQEBYTE*) malloc(chunk);
   rc = MQeFieldsDump( hSess, hFlds, offset, &buf_array[i][0],
                       chunk, base,&compcode, &reason);
   offset  += chunk;
   nchunks--;
   i++;
          }

/* 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 fields object. From this number, an application programmer can 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 `hFlds` - input
Handle to a fields object.

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

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 fields objects and determine if they both have the same fields.

This API determines if two fields 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 fields objects are considered unequal, and the result is 0. If the two fields objects are not unequal, then the result depends on whether the second fields 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 fields objects; all that matters is that both fields objects contain the same fields. The types of the fields objects do not affect the result of the comparison, however both fields 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
First fields object handle.

MQEHFIELDS `hFlds2` - input
Second fields 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 (hFlds1 is a subset of hFlds2).
- **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.
- **–1** Error.

Example

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

```
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 a fields object.

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

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

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 * textVal  = "The Owl and the Pussy Cat went to sea.";
static  MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44 };
MQEHSESS   hSess;
MQEHFIELDS hFlds;
MQEBYTE    nbuf[48];
MQECHAR *  pname;
MQEBYTE    datatype;
MQEINT32   nameLen, n;
MQEBYTE    pdata[128];
MQEINT32   compcode;
MQEINT32   reason;
MQEINT32   int32Val;
MQEINT32   nFlds;
MQEINT32   rc,i,datalen;
MQEHFIELDS 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, textVal,
                   strlen(textVal), &compcode, &reason);
rc = MQeFieldsPut( hSess, hFlds, "b" , MQE_TYPE_BYTE,  textBuf,
                   sizeof(textBuf)/sizeof(textBuf[0]),
                   &compcode, &reason);

nFlds = MQeFieldsFields( hSess, hFlds, &compcode, &reason);
/* nFlds is 3 */
```

```
memset( fd, 0, sizeof(fd));
for (i=0; i<nFlds; i++) {
   MQECHAR * pname;
   MQEBYTE * pdata;

   /* Get each field by index */
   n = MQeFieldsGetByIndex( hSess, hFlds, i, &fd, NULL,
                            &compcode, &reason, );
   pname = (MQECHAR *) malloc(fd.fd_namelen+1);
   pdata = (MQEBYTE *) malloc(datalen * MQE_SIZEOF(fd.fd_datatype));
   n = MQeFieldsGetByIndex( hSess, hFlds, i, &fd, nFlds,
                            &compcode, &reason, );
   fd.fd_name[fd.fd_namelen] = '\0';
   printf("[%d] fieldname=\"%s\", datatype=0x%x, n=%d\n", i,
          fd.fd_name, datatype, n);
   free(pname);
   free(pdata);
}

/* Output of printf()s should look something like this */
/* [0] fieldname="x",  datatype=0xC6, n=1 */
/* [1] fieldname="nm", datatype=0xC2, n=23 */
/* [2] fieldname="b",  datatype=0xC4, n=4 */
```

See Also

MQeFieldsGetByIndex

MQeFieldsFree

Description

Deallocates a fields 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

Handle to a fields object.

MQEINT32 * pCompCode - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output

If MQECC_ERROR, then *pReason could be

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, retrieve 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
Handle to a fields object.

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

MQEBYTE * pDataType - input/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/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 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
Platform specific base pointer.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_NOT_FOUND

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

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, retrieve 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. Return the number of elements on success.

If an error occurs, return 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
Handle to a fields object.

MQECHAR * `pName` - input
Null terminated string name of the array. A null or a zero length string is invalid. Field names for each array element are constructed as described above.

MQEBYTE * `pDataType` - input/output
The data type for the buffer. The input value is used with `nElements` 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
Index of 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 number of elements of if the input value of `*pDataType` . If this parameter is less than or equal to 0, no data is copied.

MQEVOID * `pBase` - input
Platform specific base pointer.

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_NOT_FOUND

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_TYPE
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";
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 and datatype */
n = MQeFieldsGetArray( hSess, hFlds, "XYZ", &datatype, 0, NULL, 0,
                       NULL, &compcode, &reason);

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

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

See Also
MQeFieldsPut

MQeFieldsGetByArrayOfFd

Description

Get fields data and data lengths from a fields 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
Handle to a fields 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 MQECC_ERROR, then `*pReason` could be

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 extracted 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

MQeFieldsGetByArrayOfFd

```
#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

Copy information about some fields in a fields object into an array of descriptors. This call is used to discover information about the fields in a fields object without providing the field names. This is useful if the contents of the fields object are fully defined. If a fields 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 fields 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 fields 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 a fields 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
Handle to a fields object.

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

(MQEFIELD) `pFds` - input/output
Array of fields descriptors data structure. 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 the number of elements in the field (of the field's datatype, not the input datatype).

MQEINT32 nFlds - input
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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_INVALID_ARGUMENT
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 updated 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";
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;
MQEINT32   reason;
MQEHFIELDS hFlds;
MQEBYTE    byteVal;
MQEINT16   int16Val;
MQEINT32   int32Val;
MQEVOID *  ppData[4], ** ppData2;
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 =
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 the fields out by index*/
nFlds     = MQeFieldsFields( hSess, hFlds, &compcode, &reason);

fd = calloc(nFlds * sizeof(*fd));

/* get datatype, name and data lengths for all fields in object */
rc = MQeFieldsGetByIndex( hSess, hFlds, 0, fd, nFlds, &compcode, &reason);

nFlds = rc;
/* Get space to store field name and data */
for (i=0; i<nFlds; i++) {
/* Get space for the field name (plus NUL)*/
   fd[i].fd_name = (MQECHAR *) malloc(fd[i].fd_namelen+1);
/* Clear for the field name */
memset(pFds[i].fd_name, 0, fd[i].fd_namelen+1);
/* Get space for the field data */
 fd[i].fd_data = (MQEBYTE *) malloc(fd[i].fd_datalen*MQE_SIZEOF
                                                    (fd[i].fd_datatype));
}

/* get name and data for all fields */
rc = MQFieldsGetByIndex( hSess, hFlds, 0, fd, nFlds,, &compcode, &reason);
```

See Also

- MQeFieldsFields
- MQeFieldsGet

MQeFieldsGetByStruct

Description

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

Given a pointer to a user data structure and its corresponding struct descriptors, this API gets all the fields 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
Handle to a fields object.

PMQEVOID `pStruct` - output
Pointer to the data structure that is to be filled out.

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

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

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

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success returns the number of descriptors pdated 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.

Design Note

For the *get* operation, the structure field descriptor defines the elements in the data structure that need to be filled out by the fields data. The field descriptor associates a field name with an element in the data structure. All primitive data types should be copied out from the fields and into the data structure. Array elements can be either copied out into a destination location specified by the element in the data structure, or to a destination that is part of the data structure itself.

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 */

#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, 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, 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..7]=0xABCD1244-00000000" */
```

See Also

MQeFieldsPutByStruct

MQeFieldsHide

Description

Exclude 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 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 fields object. The value of the hide bit is exported when the fields object is serialized with MQeFieldsDump, so hidden fields remain hidden when fields 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
Handle to a fields object.

MQECHAR * `pName` - input
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 MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_NOT_FOUND
if the specified field is not present in the fields 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

Put a field into the fields 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
Handle to a fields object.

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

MQEBYTE DataType - input
DataType of the field data. This parameter cannot be a null and whose value must be one of the defined values. See Field Data Types.

MQEVOID * pData - input
Data buffer. If NULL, then an internal buffer is allocated whose size is specified by the nElements parameter. One can then use MQeFieldsWrite to put data into this pre-allocated buffer. And this internal buffer is initialized to zeros for all 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 and MQE_TYPE_DOUBLE. If DataType is MQE_TYPE_FIELDS, pData must not be null.

MQEINT32 nElements - input
Number of elements of type DataType in pData . This must be greater than 0. If DataType is MQE_TYPE_FIELDS or MQE_TYPE_ARRAY_ELEMENTS, then nElements must 1.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_INVALID_HANDLE
If any of 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 parameters combination

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
Handle to a fields object.

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

MQEBYTE DataType - input
DataType of the field data. See MQeFields Data Type. 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
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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_INVALID_HANDLE
If any of 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 fields 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 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
Handle to a fields object.

MQEFIELD * pFds - input
An array of **struct MQeField_st** field descriptors. Puts a field named `fd_name` into a fields 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 MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success returns the number of descriptors 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";
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 fields 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
Handle to a fields object.

PMQEVOID pStruct - input
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 fields object.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

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 */
```

```
#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;

/* Initialize data */
myData.x = 20;
myData.name = textVal;
myData.namelen = sizeof(textVal);
myData.fieldlen = sizeof(field)/sizeof(field[0]);
for (rc=0; rc<4; rc++) myData.buf[rc] = textVal[rc];
for (    ; 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, 3,
                           &compcode, &reason);
/* Add "field" whose length is in myData.fieldlen */
rc = MQeFieldsPut( hSess, hFlds, "field", MQE_TYPE_INT, &field, myData.fieldlen,
                   &compcode, &reason);
```

See Also

MQeFieldsGetByStruct

MQeFieldsRead

Description

Read a portion of a field's data block. Return 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
Handle to a fields object.

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

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

MQEBYTE * pDestBuf - output
Destination buffer for the read operation

MQEINT32 srcOff - input
Offset position into the field data to start the read.

MQEINT32 srcLen - input
Number of bytes to read

MQEVOID * pBase - input
Base pointer for destination buffer pDestBuf .

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

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 fields 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

A fields object can be restored from a logical byte array to a fields handle using a sequence of `MQeFieldsRestore` calls. Each individual call does a partial restore of the fields object, specifying the next subarray of the logical byte array. This allows a large fields 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 fields handle maintains some restore state in between partial restore calls. It returns the number of bytes consumed by this partial restore.

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

If an error occurs during one of the partial restores, the fields object's internal restore state enters an invalid state, and no further updates are made to the fields handle. The remaining calls should be made with valid arguments (except the contents of the data buffer are ignored), in order to return the fields 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 fields operations use the fields 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. A 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 fields object.

MQEINT32 `dumpLen` - input
The total dump length of the fields 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
Data byte array from which to perform a partial restore of the fields object.

MQEINT32 `dataLen` - input
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 MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_HANDLE
Fields object handle is invalid.

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_DATA
Byte array could be corrupted and the restore operation could not reconstruct the fields 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";
static   MQECHAR * textVal  = "The Owl and the Pussy Cat went to sea.";
static   MQECHAR   textBuf[] = { 0xAB, 0xCD, 0x12, 0x44 };
MQEHSESS   hSess;
MQEHFIELDS hFlds, hFlds2=MQEHANDLE_NULL;
MQEINT32   compcode;
MQEINT32   reason;
MQLONg     int32Val;
MQEBYTE *  buf;
MQEINT32   nbytes, rc;

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

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

nbytes = MQeFieldsDumpLength( hSess, hFlds, &compcode, &reason);
buf    = (MQEBYTE *)malloc(nbytes);
rc     = MQeFieldsDump( hSess, hFlds, 0, buf, 0, nbytes, &compcode, &reason);

/* Restore into another fields object in 2 operations */
/* Process 20 bytes first, assuming nbytes is > 20 */
hFlds2 = MQeFieldsAlloc( hSess, "", &compcode, &reason);
MQeFieldsRestore( hSess, hFlds2, nbytes, buf, 20, &compcode, &reason);
/* Process the remaining bytes */
MQeFieldsRestore( hSess, hFlds2, nbytes, buf+20, nbytes-20, &compcode, &reason);

MQeFieldsFree( hSess, hFlds2, &compcode, &reason);
```

See Also

- MQeFieldsDump
- MQeFieldsDumpLength

MQeFieldsType

Description

Determine the string name of a fields 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 a fields object.

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

MQEINT32 typeLen - input
Size of the pTypeName buffer in MQECHARs. If pTypeName is a NULL, then this parameter is ignored.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

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

Write into the data block of an existing field in a fields object. Return the number of elements written, or -1 if an error occurred.

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 a fields object.

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

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

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

MQEBYTE * pSrcBuf - input
Source buffer

MQEINT32 srcLen - input
Number of elements of type DataType to write.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_INVALID_ARGUMENT
Invalid inputs, for example, pSrcBuf is a NULL,

MQE_EXCEPT_TYPE
DataType does not match field's data type.

MQE_EXCEPT_NOT_FOUND
No field 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

MQeFieldsWrite

```
#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

Determine if the fields 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 a fields 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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

Return Value

MQEINT32

- **1** the fields 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 fields 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

Handle of the source fields object.

MQEHFIELDS `hDstFlds` - input

Handle of the destination fields object.

MQEINT32 `Option` - input

MQE_FIELDS_OPTION_NONE

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

MQFIELDS_OPTION_ALLFIELDS

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

MQFIELDS_OPTION_REPLACE

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

MQECHAR * `pName` - input

Null terminated string name of the field. If MQFIELDS_OPTION_ALLFIELDS 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 MQECC_ERROR, then `*pReason` could be

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, MQFIELDS_OPTION_ALLFIELDS, 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 a fields 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 MQECC_ERROR, then `*pReason` could be

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 a fields 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 MQECC_ERROR, then `*pReason` could be

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

Get number of elements in an encoded array. Return the number of elements, or -1 on error.

Syntax

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

Parameters

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

MQEHFIELDS `hFlds` - input
A handle to a fields 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 MQECC_ERROR, then `*pReason` could be

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;
MQEBYTE *  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", sizeof(data), data,
```

```
                                    &compcode, &reason);

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

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

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

See Also

MQeFieldsPutArrayLength

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

Description

Extract typed data from the fields 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 a fields object.

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

MQEBYTE * `pBoolean` - output
Returned boolean value.

MQEBYTE * `pByte` - output
Returned byte value.

MQEINT16 * `pShort` - output
Returned short value.

MQEINT32 * `pInt` - output
Returned 4 byte integer value.

MQEINT64 * `pLong` - output
Returned 8 byte integer value.

MQEFLOAT * pFloat - output
Returned double value.

MQEDOUBLE* pDouble - output
Returned float value.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_TYPE
Field type is incorrect.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

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

Extract a nested fields object from a fields 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 a fields 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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_TYPE

The field was not the correct type.

MQE_EXCEPT_NOT_FOUND

Field name not found.

MQE_EXCEPT_INVALID_HANDLE

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

Extract the data from the fields 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 a fields object.

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

MQEVOID * pBytes - output
Returned byte value.

MQEVOID * pShorts - output
Returned short value.

MQEINT32 * pInts - output
Returned 4 byte integer value.

MQEVOID * pLongs - output
Returned 8 byte integer value.

MQEVOID * pFloats - output
Returned double value.

MQEVOID * pDoubles - output
Returned float value.

MQEINT32 n - input
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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_DATA
srcOff out of range.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_TRUNCATED

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, sizeof(bytes),
                         [0], &compcode, &reason);
shorts[0]= 32000;
shorts[1]= 32020;
rc       = MQeFieldsPut( hSess, hFlds, "sh", MQE_TYPE_SHORT, sizeof(shorts),
                         [0], &compcode, &reason);
ints[0]  = 2000100000;
ints[1]  = 2000020000;
ints[2]  = 2000003000;
rc       = MQeFieldsPut( hSess, hFlds, "int", MQE_TYPE_INT, sizeof(ints),
                         [0], &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, sizeof(longs),
                         [0], &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"  , [0]  , 0, 5,
                          &compcode, &reason);
MQeFieldsGetArrayOfShort( hSess, hFlds, "sh" , [0] , 0, 2,
                          &compcode, &reason);
MQeFieldsGetArrayOfInt  ( hSess, hFlds, "int", [0]   , 0, 3,
                          &compcode, &reason);
MQeFieldsGetArrayOfLong ( hSess, hFlds, "lg" , [0]  , 0, 2,
                          &compcode, &reason);
```

See Also

- MQeFieldsGet
- MQeFieldsPutArrayOfByte
- MQeFieldsPutArrayOfShort
- MQeFieldsPutArrayOfInt
- MQeFieldsPutArrayOfLong
- MQeFieldsPutArrayOfFloat
- MQeFieldsPutArrayOfDouble,

MQeFieldsGetAscii, MQeFieldsGetUnicode, MQeFieldsGetObject

Description

Extract from a single field in the fields object an array of MQECHAR, MQESHORT, or MQEBYTE. 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 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 a fields object.

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

MQEBYTE * `pData` - output
Caller supplied destination buffer to receive the output data.

MQEINT16* `pData` - output
Caller supplied destination buffer to receive the output data.

MQEINT32 `DataLen` - input
Max. number of elements to copy

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

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

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 fields 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 a fields object.

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

MQEINT16 * pData - output
Returned short value.

MQEINT32 * pData - output
Returned 4 byte integer value.

MQEINT64 * pData - output
Returned 8 byte integer value.

MQEFLOAT * pData - output
Returned double value.

MQEDOUBLE* pData - output
Returned float value.

MQEINT32 srcOff - input
Starting index for source array.

MQEINT32 n - input
number of elements to get.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_NOT_FOUND
Field name not found.

MQE_EXCEPT_DATA
srcOff 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;

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

shorts[0]= 32000;
shorts[1]= 32020;
rc = MQeFieldsPutArray( hSess, hFlds, "sh", MQE_TYPE_SHORT,
                        &shorts[0],
                         sizeof(shorts)/sizeof(shorts[0],
                        &compcode, &reason);
ints[0]  = 2000100000;
ints[1]  = 2000020000;
ints[2]  = 2000003000;
rc = MQeFieldsPutArray( hSess, hFlds, "int", MQE_TYPE_INT, &ints[0],
                         sizeof(ints)/sizeof(ints[0]),
                        &compcode, &reason);
longs[0].hi = 265;
```

```
longs[0].lo = 2000000000;
longs[1].hi = 2000000000;
longs[1].lo = 255;
rc = MQeFieldsPutArray( hSess, hFlds, "lg", MQE_TYPE_LONG, &longs[0],
                        sizeof(longs)/sizeof(longs[0]),
                       &compcode, &reason);

/* Get the data */

length = MQeFieldsGetShortArray ( hSess, hFlds, "sh", NULL, 0, 0,
                                  &compcode, &reason);
gotShorts = malloc(length);
MQeFieldsGetShortArray ( hSess, hFlds, "sh", &shorts[0], 0,
                         length, &compcode, &reason);

length = MQeFieldsGetIntArray   ( hSess, hFlds, "int", NULL, 0, 0,
                                  &compcode, &reason);
gotInts = malloc(length);
MQeFieldsGetIntArray   ( hSess, hFlds, "int", &ints[0], 0,
                         length, &compcode, &reason);

length = MQeFieldsGetLongArray  ( hSess, hFlds, "lg", NULL, 0, NULL,
                                  &compcode, &reason);
gotLongs = malloc(length);
MQeFieldsGetLongArray  ( hSess, hFlds, "lg", &longs[0], 0,
                         length, &compcode, &reason);
```

See Also

- MQeFieldsGetArray
- MQeFieldsPutByteArray
- MQeFieldsPutShortArray
- MQeFieldsPutIntArray
- MQeFieldsPutLongArray
- MQeFieldsPutFloatArray
- MQeFieldsPutLongArray

MQeFieldsGetAsciiArray, MQeFieldsGetUnicodeArray, MQeFieldsGetByteArray

Description

Extract an encoded 2 dimensional array of MQECHAR, MQESHORT, or MQEBYTE. Starting at source index `srcOff` , extract at most `n` arrays, each at 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 a fields object.

MQECHAR * `pName` - input
Null terminated string 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/output
Array of `n` buffer lengths. The input values specify the length of the buffer in MQECHARs. The output values specify the length of the array element in the fields object in MQECHARs. If NULL, then all buffer lengths are considered to be 0.

MQEINT32 srcOff - input
Starting source index from which to copy the array elements.

MQEINT32 n - input
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 MQECC_ERROR, then *pReason could be

MQE_EXCEPT_NOT_FOUND
A field was not found.

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAILED

MQE_EXCEPT_TYPE
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 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

Put 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 a fields object.

MQECHAR * pName - input

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

MQEINT32 nElements - input

Number of array elements

MQEINT32 * pCompCode - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output

If MQECC_ERROR, then `*pReason` could be

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;
MQEINT32  n, data0, data1;
MQEBYTE *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       = MQeFieldsDataLen( hSess, hFlds, "foo", &compcode, &reason);
```

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

/* Get back the data */
rc      = MQeFieldsGetIntArray( hSess, hFlds, "ibm", pData, n,
                                &compcode, &reason );
data0 = *(MQEINT32*) pData;
data1 = *(MQEINT32*) (pData+4);
```

See Also

MQeFieldsGetArrayLength

MQeFieldsPutBoolean

Description

Put a boolean value into a fields 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 a fields object.

MQECHAR * pName - input
Null terminated string 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 MQECC_ERROR, then *pReason could be

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

Put a fields object into another fields object. The fields object that is being put into the other fields object becomes invalid after the API call. A fields 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 fields object that is receiving the other fields object.

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

MQEHFIELDS hFlds2 - input
The fields object that is being moved into the other fields object. This fields object becomes invalid after this API returns.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT
Invalid pName and/or hFlds1 equals 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

Put a float and a double into a field object. Put a 8, 16, 32, or 64 bit integer, float, or double into the fields 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 a fields object.

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

MQEBYTE* `aByte` - input
Pointer to a byte value.

MQEINT16 *`int16Val` - input
Pointer to a 16 bit short integer value.

MQEINT32*`anInt` - input
Pointer to a 32 bit integer value.

MQEINT64 * `pLong` - output
Pointer to a 64 bit integer value.

MQEFLOAT*`aFloat` - input
Pointer to a float value.

MQEDOUBLE* `aDouble` - input
Pointer to a double value.

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

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

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 fields 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 a fields object.

MQECHAR * `pName` - input
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
Max. number of MQECHAR, MQESHORT, or MQEBYTE to copy.

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

MQEINT32 * `pReason` - output
If MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32
Returns 0 on success or -1 on failure.

Example

MQeFieldsPutAscii

```
#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

Put an array of float and double into a single field in a field object. Put an array of 8, 16, 32, or 64 bit integers, floats or doubles into a single field in a fields 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 a fields object.

MQECHAR * `pName` - input
Null terminated string 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 integer.

MQEINT32 * `pInt` - input
An array of 4 byte integer.

MQEINT64 * `pLong` - output
An array of 8 byte integer.

MQEFLOAT * pFloat - input
An array of floats.

MQEDOUBLE* pDouble - input
An array of doubles.

MQEINT32 n - input
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 MQECC_ERROR, then *pReason could be

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

Put an array of MQEINT16s, MQEINT32s, MQEINT64s, MQEFLOATs, MQEDOUBLEs, or MQEHFIELDS as an encoded array into a fields 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 fields 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 a fields object.

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

MQEINT16 * pData - input
Input array.

MQEINT32 * pData - input
Input array.

MQEINT64 * pData - input
Input array.

MQEFLOAT * pData - input
Input array.

MQEDOUBLE* pData - input
Input array.

MQEHFIELDS* pData - input
Input array.

MQEINT32 n - input
number of elements to put.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

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

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_INVALID_ARGUMENT

MQE_EXCEPT_ALLOCATION_FAILED

Return Value

MQEINT32

- On success returns the number of put successfully.
- 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

Put a 2 dimensional array of MQECHAR, MQEINT16, or MQEBYTE as an encoded array into a fields object. First, the array elements are inserted in order as encoded fields, then the array length. Return the total number of fields added to the fields 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 a fields object.

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

MQEINT32 `n` - input
number of elements to put. If 0, the number of elements in the field is returned.

MQECHAR * `ppData[]` - input
Array of MQECHAR arrays.

MQEINT16 * `ppData[]` - input
Array of MQESHORT arrays.

MQEBYTE * `ppData[]` - input
Array of MQEBYTE.

MQEINT32 `pDataLen[]` - input
Array of lengths of each data element. correspond to each element of `ppData[]` .

MQEINT32 `srcOff` - input
Starting index from which to copy the array element.

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

MQEINT32 * pReason - output

If MQECC_ERROR, then *pReason could be

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 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);
```

See Also

- MQeFieldsGetAsciiArray
- MQeFieldsGetUnicodeArray
- MQeFieldsGetByteArray

System

The following APIs are used to interact with the MQSeries Everyplace queue manager.

MQeInitialize
Initiate a session with the MQSeries Everyplace client library.

MQeTerminate
Terminate a session with the MQSeries Everyplace client library. Write a trace string to default trace output.

MQeGetVersion
Get the version number of current MQSeries Everyplace software.

MQeConfigCreateQMgr
Initialize and create a queue manager presence on the system.

MQeConfigDeleteQMgr
Terminate and remove the presence of a queue manager in the system.

MQeTraceCmd
Trace enabling API.

MQeTrace

General constraints

Queue manager name must

- Not be NULL. An empty string "" defaults to the local queue manager.
- Conform to the ASCII character set, (characters with values that are greater than 31 but less than 128 and not include any of the characters {}[]#():;,'=)
- Be less than 48 characters long if required to interoperate with MQSeries.

Queue name must

- Be at least one character long.
- Conform to the ASCII character set, (characters with values that are greater than 31 but less than 128 and not include any of the characters {}[]#():;,'=)
- Be less than 48 characters long if required to interoperate with MQSeries.

MQeInitialize

Description

Initializes the MQSeries Everyplace for the application. If successful, 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. It 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. This name is used to identify the session; therefore, any previously opened 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 the recommendation is to keep it short, preferably less than 20 characters.

MQEINT32 * pCompCode - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output

If returned `*pCompCode` equals MQECC_WARNING, `*pReason` may have the following

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 MQECC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_ARGUMENT

Invalid session name, too short or too long.

MQE_EXCEPT_ALLOCATION_FAILED

MQSeries Everyplace library depleted its 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 a 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

This session handle returned by MQeInitialize

MQEINT32 * pCompCode - output

MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output

If MQECC_ERROR, then `*pReason` could be

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, MQCC_WARNING or MQCC_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);
```

See Also

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

Name of the local queue manager to be created.

MQEINT32 * pCompCode - output

MQECC_OK, MQCC_WARNING or MQCC_ERROR.

MQEINT32 * pReason - output

If MQECC_ERROR, then `*pReason` could be caused by

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, MQCC_WARNING or MQCC_ERROR.

MQEINT32 * pReason - output

If MQECC_ERROR, then `*pReason` could be caused by

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

MQeTraceCommand

Description

Start, stop and set 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 application trace string that starts with character

MQE_TRACE_OPTION_APP_INFO
Write out application trace string that starts with character 'I'

MQE_TRACE_OPTION_APP_WARNING
Write out application trace string that starts with character 'W'

MQE_TRACE_OPTION_APP_ERROR
Write out application trace string that starts with character 'E'

MQE_TRACE_OPTION_APP_DEBUG
Write out 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 system trace string that starts with character '_'

MQE_TRACE_OPTION_SYS_INFO
Write out system trace string that starts with character 'i'.

MQE_TRACE_OPTION_SYS_WARNING
Write out system trace string that starts with character 'w'

MQE_TRACE_OPTION_SYS_ERROR
Write out system trace string that starts with character 'e'

MQE_TRACE_OPTION_SYS_DEBUG
Write out system trace string that starts with character 'd'

MQE_TRACE_OPTION_SYS_ALL
Write out all system trace strings.

MQEINT32 * pCompCode - output
MQECC_OK, MQCC_WARNING or MQCC_ERROR.

MQEINT32 * pReason - output
If MQCC_ERROR, then `*pReason` could be

MQE_EXCEPT_INVALID_HANDLE

MQE_EXCEPT_ALLOCATION_FAIL
MQSeries Everyplace library depleted one of its resource or the system storage resources, e.g., an example of the latter could be not being able to open a file or database.

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

Write 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 a MQTS() macro.

Syntax

```
#include <hmq.h>
MQEVOID MQeTrace ( MQEHSESS hSess, MQETCHAR * pTStr);
```

Parameters

MQEHSESS hSess - input
This session handle returned by MQeInitialize.

MQETCHAR pTStr - input
Null terminated trace string.

MQEINT32 * pCompCode - output
MQECC_OK, MQCC_WARNING or MQCC_ERROR.

MQEINT32 * pReason - output
If MQCC_ERROR, then *pReason could be

MQE_EXCEPT_INVALID_HANDLE

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
Browse messages on a queue.

MQeQMgrConfirmMsg
Delete a message already retrieved from a queue or make a previously put message available.

MQeQMgrDeleteMsgs
Deletes an array of messages on the queue.

MQeQMgrGetMsg
Get a message from a queue.

MQeQMgrGetName
Get the name of the local queue manager.

MQeQMgrPutMsg
Put a message onto a queue.

MQeQMgrUndo
Undo one or more messages that were put, retrieved, or locked on a queue.

MQeQMgrUnlockMsgs
Unlock an array of messages on the queue that were locked by MQeQMgrBrowseMsgs().

General constraints

Queue manager name must

- Not be NULL. An empty string "" defaults to the local queue manager.
- Conform to the ASCII character set, (characters with values that are greater than 31 but less than 128 and not include any of the characters {}[]#():;,'=)
- Be less than 48 characters long if required to interoperate with MQSeries.

Queue name must

- Be at least one character long.
- Conform to the ASCII character set, (characters with values that are greater than 31 but less than 128 and not include any of the characters {}[]#():;,'=)
- Be less than 48 characters long if required to interoperate with MQSeries.

MQeQMgrBrowseMsgs

Description

Browse messages on a queue without removing the messages from the queue. The browse returns an array of message object handles. These message objects can then be interrogated by the application. Parameters can be specified to make the browse more specific by providing fields to match in the filter parameter, for example, message object fields (for example, MessageId and Priority), could be specified so that only messages that have matching fields are returned.

The size of the array into which the results are returned is specified by the application, giving the application programmer the ability to 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 1.0. This is especially 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 passed on the first call (pBrowseMsgOpts points to a MQEBMO type which maintains the context information for the browse). See example.

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
This session handle returned by MQeInitialize.

MQECHAR * pQMName - input
Null terminated ASCII string name of the queue manager. An empty string name "" defaults to local queue manager. A null is invalid input.

MQECHAR * pQName - input
Null terminated ASCII string name of the queue. A null or an empty string is invalid.

MQEVOID * pBrowseMsgOpts - input
This parameter is 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
Structure ID for the GetMsgOpts which is "BRWS" .

MQEINT32 Version - input
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 by the MQeQMgrDeleteMsgs or unlocked by the MQeQMgrUnlockMsgs function call, or if a *confirmID* is supplied, then MQeQMgrUndo can be used to unlock the messages on the queue.

When doing a synchronous browse operation on a remote queue, it is highly recommended that the 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 lock ID, the locked messages on the queue cannot be unlocked by the application and MQSeries Everyplace system administrative intervention would be required. However, with a confirm ID, 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 for this function call 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
Do the BrowseMsg operation with the confirm ID

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 tag message object is made inaccessible for MQeQMgrGetMsg() calls without the UID of the message and MQeQMgrBrowseMsgs(). These messages are made accessible again after MQeQMgrUndo() is called with this confirm ID.

This confirm ID 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 would affect the messages of another device with the same 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 isn't set, the call fails.

This confirm ID is intended to be used with the MQeQMgrUndo function, and should not be used with the MQeQMgrConfirmMsg function.

MQEHATTRB hAttrb - input

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. Default value is MQEHANDLE_NULL.

Release 1.0 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 `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() API 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/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. The application need not understand the meaning of this value except to pass it back on subsequent calls. 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 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 a MQEBMO data structure with the default values is used.

MQEHFIELDS hFilter - input
Handle to the MQeFields object that contains the matching criteria 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
Array to hold returned message object handles. If NULL, then zero is returned. If `pCookie` is not NULL, then its resources are released. Users are expected to call MQeFieldsFree() to release Fields handles held by this array.

MQEINT32 nMsgs - input
Number of message to browse for this call.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output
If MQECC_ERROR, then `*pReason` could be

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

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 assure 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
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string 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
A fields 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 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 MQECC_ERROR, then `*pReason` could be

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 confirm ID 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

Delete the messages on a queue identified by the unique identifier of each message. The unique identifier is a combination of 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
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string 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, these same 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 an 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, then 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 Fields handles stored in this array.

MQEINT32 nMsgs - input
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 MQECC_ERROR, then `*pReason` could be

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
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

- MQeQMgrBrowseMsgs
- MQeQMgrUnlockMsgs

MQeQMgrGetMsg

Description

Get the first message on a queue that matches the filter criteria. 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. Parameters can be specified, which are message object attributes (such as MessageId and Priority), in this case only messages which 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
This session handle, returned by MQeInitialize.

MQECHAR * pQMName - input
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string 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
Structure ID for the GetMsgOpts which is "GETM" .

MQEINT32 Version - input
Version number of this data structure. The current version number is 1.

MQEINT32 Options - input

- MQE_QMGR_OPTION_CONFIRMID - Do the GetMsg operation with the confirm ID. The retrieved message becomes inaccessible to subsequent MQeQMgrBrowseMsg() and MQeQMgrGetMsg() calls, and is not deleted from the queue until the MQeQMgrConfirmMsg is called with the UID of this message object or is made accessible again with MQeQMgrUndo call.

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 for future MQeQMgrGetMsg and MQeQMgrBrowseMsgs until MQeQMgrUndo is called with this confirm ID.

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 isn't set, the call fails.

MQEHATTRB hAttrb - input
Handle to the attribute object that is used to decode the message object on the queue before it is returned by this API. Default value is MQEHANDLE_NULL.

Note: Version 1.0 does not support message-level security so this parameter is ignored.

If NULL, then a MQEGMO data structure with the default values is used.

MQEHFIELDS hFilter - input
Handle to the Fields object 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 MQECC_ERROR, then `*pReason` could be

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
Handle to a message object (a fields 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
This session handle, returned by MQeInitialize.

MQECHAR * pQMgrName - output
Output into which the string name of the local queue manager is copied. If the buffer is NULL, then the length of the local queue manager name is returned.

MQEINT32 qmNameLen - input
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 MQECC_ERROR, then `*pReason` could be

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);
```

See Also

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 Release 1.0). 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; and 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 everytime this API is called. So an application can call this API with the same input message object and the UID of this message object is reset and then set again with different value every time. This resetting and setting mechanism guarantees that no message object with 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
This session handle, returned by MQeInitialize.

MQECHAR * pQMName - input
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string name of the queue.

MQEVOID * putMsgOpts - input/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
Structure ID for the GetMsgOpts that is `"PUTM"` .

MQEINT32 Version - input
Version number of this data structure. The current version number is 1.

MQEINT32 Options - input

- `MQE_QMGR_OPTION_CONFIRMID` - Do the PutMsg operation with the confirm ID. The put message is inaccessible to subsequent MQeQMgrBrowseMsg() and MQeQMgrGetMsg() calls until the

MQeQMgrConfirmMsg is called with the UID of the `hMsg` or is deleted from the queue with MQeQMgrUndo call.

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.

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 isn't set, the call fails.

MQEHATTRB hAttrb - input
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 1.0 does not support message-level security so this parameter is ignored.

If NULL, then a MQEPMO data structure with the default values is used.

MQEHFIELDS hMsg - input/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 MQECC_ERROR, then `*pReason` could be

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 object or a set of message objects on a queue that have the same confirm ID value. If the previous operation on the message objects is an MQeQMgrBrowseMsgs with lock, the messages objects are unlocked and made accessible again. If the previous operation on the message objects is a MQeQMgrGetMsg, then the message object is put back onto the queue. If the previous operation is a MQeQMgrPutMsg, then 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
This session handle, returned by MQeInitialize.

MQECHAR * pQMName - input
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string name of the queue.

MQEINT64 * pConfirmId - input
A 64 bit integer confirm ID that was used on previous operations on the message objects on the queue.

MQEINT32 * pCompCode - output
MQECC_OK, MQECC_WARNING or MQECC_ERROR.

MQEINT32 * pReason - output
If MQECC_ERROR, then `*pReason` could be

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 lock ID and the unique identifier (UID) of the message. The lock ID is returned from the earlier browse and lock operation, MQeQMgrBrowseMsgs with option MQE_QMGR_OPTION_BROWSE_LOCK. This API works in tandem with the MQeQMgrBrowseMsgs.

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
This session handle, returned by MQeInitialize.

MQECHAR * pQMName - input
Null terminated string name of the queue manager.

MQECHAR * pQName - input
Null terminated string name of the queue.

MQEINT64 * pLockID - input
The 8-bytes lock ID that was returned by the MQeQMgrBrowseMsgs() API call with MQE_QMGR_OPTION_BROWSE_LOCK option specified. This parameter must be specified for this API call.

MQEINT32 pMsgs[] - input
An array of message object handles to be unlocked. These messages object handles should be the same ones that were returned by the MQeQMgrBrowseMsgs() API 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 Fields handles stored in this array.

MQEINT32 nMsgs - input
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 MQECC_ERROR, then *pReason could be

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

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

exceptions and options

- 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
- 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 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

Part 4. Appendixes

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,

Winchester,
Hampshire
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business machines Corporation in the United States, or other countries, or both.

IBM
MQSeries

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

Windows 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 MQSeries.

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 MQSeries, 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 MQSeries client to a queue manager on a server system and transfers MQI calls and responses in a bidirectional manner.

MQSeries. MQSeries 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 MQSeries, 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 MQSeries 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 MQSeries 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 Introduction*, GC34-5843-00
- *MQSeries Everyplace Programming Reference*, SC34-5846-00
- *MQSeries Everyplace Programming Guide*, SC34-5845-00
- *MQSeries An Introduction to Messaging and Queuing*, GC33-0805-01
- *MQSeries for Windows NT V5R1 Quick Beginnings*, GC34-5389-00

Index

N

O

P

Q

R

S

T

U

W

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

GC34-5883-00