



Sun Java™ System

Message Queue 3 Developer's Guide for C Clients

2005Q1

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819-0067-10

Copyright © 2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms. This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, Sun[tm] ONE, JDK, Java Naming and Directory Interface, JavaMail, JavaHelp and Javadoc are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

Copyright © 2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

L'utilisation est soumise aux termes de la Licence.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, Sun[tm] ONE, JDK, Java Naming and Directory Interface, JavaMail, JavaHelp et Javadoc sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Ce produit est soumis à la législation américaine en matière de contrôle des exportations et peut être soumis à la réglementation en vigueur dans d'autres pays dans le domaine des exportations et importations. Les utilisations, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers les pays sous embargo américain, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exhaustive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécialement désignés, sont rigoureusement interdites.

Contents

List of Tables	7
List of Procedures	9
Preface	11
Who Should Use This Book	11
Before You Read This Book	12
How This Book Is Organized	12
Conventions Used in This Book	13
Text Conventions	13
Directory Variable Conventions	13
Related Documentation	14
The Message Queue Documentation Set	14
Online Help	15
Example Client Applications	15
The Java Message Service (JMS) Specification	16
Related Third-Party Web Site References	16
Sun Welcomes Your Comments	16
Chapter 1 Introduction	17
Message Queue for the C Developer	18
Building and Running C Clients	19
Building C Clients	19
Header Files and Shared Libraries	19
Pre-Processor Definitions	20
C++ Runtime Library Support	21
Providing Runtime Support	21

Working With the Sample C-Client Programs	22
Building the Sample Programs	22
To Compile and Link on Solaris	22
To Compile and Link on Linux	22
To Compile on Windows	22
To Link on Windows	23
Running the Sample Programs	23
Client Application Deployment Considerations	24
Chapter 2 Using the C API	25
Message Queue C Client Setup Operations	26
To Set Up a Message Queue C Client to Produce Messages	26
To Set Up a Message Queue C Client to Consume Messages Synchronously	27
To Set Up a Message Queue C Client to Consume Messages Asynchronously	27
Working With Properties	28
Setting Connection and Message Properties	28
To Set Properties for a Connection	29
To Set Message Properties	30
Getting Message Properties	30
To Iterate Through a Properties Handle	31
Working With Connections	32
Defining Connection Properties	33
Connection Handling	34
Reliability	34
Flow Control	34
Working With Secure Connections	35
Configuring the Client for Secure Communication	36
Verification Using Fingerprints	37
To Set Up Fingerprint Certification, Do the Following:	37
Coordinating NSS Initialization	37
To Coordinate NSS Initialization	38
Shutting Down Connections	38
Working With Sessions and Destinations	39
Creating a Session	40
Transacted Sessions	40
Message Acknowledgement	40
Receive Mode	41
Managing a Session	42
Creating Destinations	42
Programming Domains	43
Auto-Created Destinations	43
Temporary Destinations	43
Getting Information About Destinations	44

Working With Messages	44
Composing Messages	44
Message Header	45
Message Body Types	46
Composing the Message	46
Sending a Message	47
Receiving Messages	49
Working With Consumers	50
Receiving a Message Synchronously	50
Receiving a Message Asynchronously	51
Processing a Message	52
Error Handling	53
To Handle Errors in Your Code	53
Memory Management	54
Logging	55
Chapter 3 Client Design Issues	57
Producers and Consumers	57
Using Selectors Efficiently	58
Determining Message Order and Priority	59
Managing Threads	60
Message Queue C Runtime Thread Model	60
Concurrent Use of Handles	60
Single-Threaded Session Control	61
Connection Exceptions	62
Managing Physical Destination Limits	62
Managing the Dead Message Queue	63
Factors Affecting Performance	67
Delivery Mode (Persistent/Non-persistent)	68
Use of Transactions	69
Acknowledgement Mode	69
Durable vs. Non-Durable Subscriptions	70
Use of Selectors (Message Filtering)	71
Message Size	71
Message Body Type	72
Chapter 4 Reference	73
Data Types	74
Connection Properties	76
Acknowledge Modes	80
Callback Type for Asynchronous Messaging	81
Callback Type for Connection Exception Handling	83

Function Reference	84
Setting a Client Identifier	102
Handling Connection Exceptions	103
Header Files	183
Appendix A Message Queue C API Error Codes	185
Error Codes	186
Index	193

List of Tables

Table 1	Book Contents	12
Table 2	Document Conventions	13
Table 3	Message Queue Directory Variable Used by C Clients	14
Table 4	Message Queue Documentation Set	14
Table 1-1	Locations of C-API Libraries and Header Files	20
Table 1-2	Preprocessor Definitions for Supporting Fixed-Size Integer Types	20
Table 1-3	Location of C Sample Programs	22
Table 1-4	Checklist for the Message Queue Administrator	24
Table 2-1	Functions Used to Set Properties	28
Table 2-2	Functions Used to Get Message Properties	31
Table 2-3	Functions Used to Work with Connections	32
Table 2-4	Functions Used to Work with Sessions	39
Table 2-5	Functions Used to Work with Destinations	42
Table 2-6	Functions Used to Construct Messages	44
Table 2-7	JMS-defined Message Header	45
Table 2-8	C-API Message Body Types	46
Table 2-9	Functions for Sending Messages	47
Table 2-10	Functions Used to Receive Messages	49
Table 2-11	Functions Used to Process Messages	52
Table 2-12	Functions Used in Handling Errors	53
Table 2-13	Functions Used to Free Memory	54
Table 3-1	Thread Model for NSPR GLOBAL Threads	60
Table 3-2	Handles and Concurrency	61
Table 3-3	Message Properties Relating to Dead Message Queue	64
Table 3-4	Dead Message Properties	65
Table 3-5	Comparison of High Reliability and High Performance Scenarios	67
Table 4-1	Message Queue C-API Data Type Summary	74

Table 4-2	Connection Properties	77
Table 4-3	acknowledgeMode Values	81
Table 4-4	Message Queue C-API Function Summary	84
Table 4-5	Message Header Properties	134
Table 4-6	Message Header Properties	173
Table 4-7	Message Queue C-API Header Files	183
Table A-1	Message Queue C Client Error Codes	186

List of Procedures

- To Compile and Link on Solaris 22
- To Compile and Link on Linux 22
- To Compile on Windows 22
- To Link on Windows 23
- To Set Up a Message Queue C Client to Produce Messages 26
- To Set Up a Message Queue C Client to Consume Messages Synchronously 27
- To Set Up a Message Queue C Client to Consume Messages Asynchronously 27
- To Set Properties for a Connection 29
- To Set Message Properties 30
- To Iterate Through a Properties Handle 31
- To Set Up Fingerprint Certification, Do the Following: 37
- To Coordinate NSS Initialization 38
- To Handle Errors in Your Code 53

Preface

This book provides programming and reference information for developers working with Sun Java™ System Message Queue (formerly Sun™ ONE Message Queue) 3 2005Q1, who want to use the C language binding to the Message Queue Service to send, receive, and process Message Queue messages.

This preface contains the following sections:

- [“Who Should Use This Book” on page 11](#)
- [“How This Book Is Organized” on page 12](#)
- [“Conventions Used in This Book” on page 13](#)
- [“Related Documentation” on page 14](#)

Who Should Use This Book

This guide is for developers who want to use the C-API in order to write C or C++ messaging programs that can interact with the Message Queue broker to send and receive JMS messages.

This book assumes that readers are experienced C or C++ programmers and that they are familiar with the Java Message Service specification.

Before You Read This Book

You must read the *Message Queue Technical Overview* to become familiar with Message Queue's implementation of the Java Message Service specification, with the components of the Message Queue service, and with the basic process of developing, deploying, and administering a Message Queue application.

How This Book Is Organized

This guide is designed to be read from beginning to end. The following table briefly describes the contents of each chapter:

Table 1 Book Contents

Chapter	Description
Chapter 1, "Introduction"	Introduces the C-API, provides quick start instructions on compiling and building Message Queue C clients. Introduces the Message Queue C-Client sample applications that are shipped with Message Queue, and explains how you set up your environment to run these examples. Provides a deployment worksheet.
Chapter 2, "Using the C API"	Explains how you use the C-API to construct, to send, to receive, and to process messages. This chapter also covers error handling, memory management, and logging.
Chapter 3, "Client Design Issues"	Explains the major considerations that you need to keep in mind when designing a Message Queue C client.
Chapter 4, "Reference"	Provides complete reference information for the Message Queue C-API: data structures and functions. It also lists and describes the contents of the C-API header files.
Appendix A, "Message Queue C API Error Codes"	Lists the code and descriptive string returned for errors that are returned by C library functions.

Conventions Used in This Book

This section provides information about the conventions used in this document.

Text Conventions

Table 2 Document Conventions

Format	Description
<i>italics</i>	Italicized text represents a placeholder. Substitute an appropriate clause or value where you see italic text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced.
monospace	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method or function names (including all elements in the signature), package names, reserved words, and URLs.
[]	Square brackets to indicate optional values in a command line syntax statement.
ALL CAPS	Text in all capitals represents file system types (GIF, TXT, HTML and so forth), environment variables (IMQ_HOME), or acronyms (Message Queue, JSP).
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Directory Variable Conventions

Message Queue makes use of three directory variables, one of which is relevant to C clients. [Table 3](#) describes this variable and explains how it is used on the Solaris, Windows, and Linux platforms.

Table 3 Message Queue Directory Variable Used by C Clients

Variable	Description
<code>IMQ_HOME</code>	<p>This is generally used in Message Queue documentation to refer to the Message Queue base directory (root installation directory):</p> <ul style="list-style-type: none"> On Solaris, there is no root Message Queue installation directory. Therefore, <code>IMQ_HOME</code> is not used in Message Queue documentation to refer to file locations on Solaris. On Windows, the root Message Queue installation directory is set by the Message Queue installer (by default, as <code>C:\Program Files\Sun\MessageQueue3</code>). On Linux, there is no root Message Queue installation directory. Therefore, <code>IMQ_HOME</code> is not used in Message Queue documentation to refer to file locations on Linux. For Sun Java System Application Server on Windows, Solaris, and Linux, the root Message Queue installation directory is <code>/imq</code>, under the Application Server base directory.

In this guide, `IMQ_HOME` is shown *without* platform-specific environment variable notation or syntax (for example, `$IMQ_HOME` on UNIX). Path names generally use UNIX directory separator notation (`/`).

Related Documentation

In addition to this guide, Message Queue provides additional documentation resources.

The Message Queue Documentation Set

The documents that comprise the Message Queue documentation set are listed in [Table 4](#) in the order in which you would normally use them.

Table 4 Message Queue Documentation Set

Document	Audience	Description
<i>Message Queue Installation Guide</i>	Developers and administrators	Explains how to install Message Queue software on Solaris, Linux, and Windows platforms.

Table 4 Message Queue Documentation Set (*Continued*)

Document	Audience	Description
<i>Message Queue Release Notes</i>	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
<i>Message Queue Technical Overview</i>	Developers and administrators	Introduces the basic concepts of JMS client design and describes Message Queue services and tools.
<i>Message Queue Developer's Guide for Java Clients</i>	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using JMS and SAAJ and Message Queue software.
<i>Message Queue Administration Guide</i>	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using Message Queue administration tools.
<i>Message Queue Developer's Guide for C Clients</i>	Developers	Provides programming and reference documentation for developers of Message Queue C and C++ client programs that use the Message Queue software.

Online Help

Message Queue 3 2005Q1 includes command-line utilities for performing Message Queue message service administration tasks. To access the online help for these utilities, see the *Message Queue Administration Guide*.

Message Queue 3 2005Q1 also includes a graphical user interface (GUI) administration tool, the Administration Console (`imqadmin`). Context sensitive online help is included in the Administration Console.

Example Client Applications

Sample applications that illustrate the C-API are listed and described in [Chapter 1, "Introduction"](#).

The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

<http://java.sun.com/products/jms/docs.html>

The specification includes sample JMS Java client code.

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

NOTE Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions.

To share your comments, go to <http://docs.sun.com> and click Send Comments. In the online form, provide the document title and part number. The part number is a seven-digit or nine-digit number that can be found on the title page of the book or at the top of the document.

Introduction

This chapter summarizes the differences between the C API and the Java API to Message Queue and provides a quick start to compiling and running Message Queue C clients. It covers the following topics:

- [“Message Queue for the C Developer” on page 18](#)
- [“Building and Running C Clients” on page 19](#)
- [“Working With the Sample C-Client Programs” on page 22](#)
- [“Client Application Deployment Considerations” on page 24](#)

You should be familiar with the concepts presented in the *Message Queue Technical Overview* before you read this chapter.

Depending on your needs, after you read this chapter, you can proceed either to [Chapter 3, “Client Design Issues” on page 57](#), which describes the major issues governing C client design, or to [Chapter 2, “Using the C API” on page 25](#), which explains how you use C data types and functions to obtain the messaging behavior that interests you.

The term “C developer” is used generically throughout this book and includes the C++ developer as well.

Message Queue for the C Developer

The Message Queue product is an enterprise messaging system that implements the Java™ Message Specification (JMS) standard as a JMS provider. Message Queue developers can use two programming interfaces to establish a connection to the broker, and send or receive messages:

- C clients use the API described in this manual to send messages to and retrieve messages from a Message Queue broker.
- Java clients use the Java API, described in the *Message Queue Developer's Guide for Java Clients*, to send messages to and receive messages from a Message Queue broker.

Message Queue provides a C API to its messaging services to enable legacy C applications and C++ applications to participate in JMS-based messaging. It is important to understand however that the Java Message Service specification is a standard for *Java* clients only; thus the C API described in this book is specific to the Message Queue provider and cannot be used with other JMS providers. A messaging application that includes a C client cannot be handled by another JMS provider.

The C interface, compared to the Java interface, does not support the following features:

- The use of administered objects
- Map, stream, or object message types
- Consumer-based flow control
- Queue browsers
- JMS application server facilities (ConnectionConsumer, distributed transactions)
- Receiving or sending SOAP messages
- Receiving or sending compressed JMS messages
- Auto-reconnect or failover, which allows the client runtime to automatically reconnect to a broker if a connection fails
- The `NO_ACKNOWLEDGE` mode

Like the Java interface, the C interface does support the following:

- Publish/subscribe and point-to-point connections
- Synchronous and asynchronous receives
- CLIENT, AUTO, and DUPS_OK acknowledgement modes
- Local transactions
- Session recover
- Temporary topics and queues
- Message selectors

The JMS programming model is the foundation for the design of a Message Queue C client. [Chapter 2, “Using the C API” on page 25](#) explains how this model is implemented by the C data types and functions used by a Message Queue C client for delivery of messages.

The next section provides a quick introduction to building and running Message Queue clients.

Building and Running C Clients

Message Queue provides several sample Message Queue C-client applications that illustrate how to send and receive messages. Before you run these applications, read through the next two sections to make sure that you understand the general procedure and requirements for building and running Message Queue C-Client programs.

Building C Clients

This section explains how you build Message Queue programs from C source files. You should already be familiar with writing and compiling C applications.

Header Files and Shared Libraries

The Message Queue C client includes the header files (`mqcrt.h`), the C client runtime shared library `mqcrt`, and its direct dependency libraries. When writing a Message Queue C client application, you should include the header files and link to the runtime library `mqcrt`.

For each platform, [Table 1-1](#) lists the installed location of the header files and the supporting runtime library.

Table 1-1 Locations of C-API Libraries and Header Files

Platform	Library	Header File
Solaris	/opt/SUNWimq/lib	/opt/SUNWimq/include
Solaris/SPARC	/opt/SUNWimq/lib/sparcv9 (64-bit support)	/opt/SUNWimq/include
Linux	/opt/sun/mq/lib	/opt/sun/mq/include
Windows	IMQ_HOME\lib	IMQ_HOME\include

Pre-Processor Definitions

Use the appropriate compiler for your platform, as described in the *Message Queue Release Notes*.

When compiling a Message Queue C client application, you need to specify the preprocess or definition shown for each platform in [Table 1-2](#). This definition is used to support Message Queue fixed-size integer types.

Table 1-2 Preprocessor Definitions for Supporting Fixed-Size Integer Types

Platform	Definition
Solaris	SOLARIS
Linux	LINUX
Windows	WIN32

C++ Runtime Library Support

When building a Message Queue C client application, you should be aware that the Message Queue C runtime library is a multi-threaded library and requires C++ runtime library support:

- **On Solaris**, this support is provided by the Sun WorkShop 6 `libCrun C++` runtime library.
- **On Linux**, this support is provided by the `gcc/g++ libstdc++` runtime library.
- **On Windows**, this support is provided by Microsoft Windows Visual C++ runtime library `msvcr.t`.

Providing Runtime Support

To run a Message Queue C-client application, you need to make sure that the application can find the `mqcrt` shared library. Please consult the documentation for your compiler to determine the best way to do this.

You also need to make sure that the appropriate C++ runtime support library, as described in [“C++ Runtime Library Support” on page 21](#) is available.

On Windows you also need to make sure that your application can find the dependent libraries NSPR and NSS that are shipped with Message Queue. These may be different from the NSPR and NSS libraries that are installed on your system to support the Netscape browser and the Sun Java System Application Server. The `mqcrt` shared library depends directly on the NSPR and NSS versions installed with Message Queue. If a different version of the libraries is loaded at runtime, you may get a runtime error specifying that the libraries being used are incompatible.

Working With the Sample C-Client Programs

This section describes the sample C-Client programs that are installed with Message Queue and explains how you should build them and run them. [Table 1-3](#) lists the location of the sample programs on each platform.

Table 1-3 Location of C Sample Programs

Platform	Directory
Solaris	/opt/SUNWimq/demo/C
Linux	/opt/sun/mq/examples/C
Windows	IMQ_HOME\demo\C

Building the Sample Programs

The following commands illustrate the process of building and linking the sample application `Producer.c` on the Solaris, Linux, and Windows platforms. The commands include the preprocessor definitions needed to support fixed-size integer types. For options used to support multithreading, please consult documentation for your compiler.

► To Compile and Link on Solaris

```
CC -compat=5 -mt -DSOLARIS -I/opt/SUNWimq/include -o Producer \
-L/opt/SUNWimq/lib -lmqcrtd Producer.c
```

For Solaris/SPARC, if you need 64-bit support, you need to specify the following compiler options: `-xarch=v9` and `-L/opt/SUNWimq/lib/sparcv9`. For example, to compile and link the example application, you would use the following command:

```
CC -compat=5 -mt -xarch=v9 -DSOLARIS -I/opt/SUNWimq/include -o Producer \
L/opt/SUNWimq/lib/sparcv9 -lmqcrtd Producer.c
```

► To Compile and Link on Linux

```
g++ -DLINUX -D_REENTRANT -I/opt/sun/mq/include -o Producer \
-L/opt/sun/mq/lib -lmqcrtd Producer.c
```

► To Compile on Windows

```
cl /c /MD -DWIN32 -I%IMQ_HOME%\include Producer.c
```

► To Link on Windows

```
link Producer.obj /NODEFAULTLIB msvcrt.lib \  
    /LIBPATH:%IMQ_HOME%\lib mqcr.lib
```

Running the Sample Programs

The sample C client program files include the following:

- `Producer.c` and `Consumer.c`, which illustrate how you send a message and receive it synchronously.
- `ProducerAsyncConsumer.c`, which illustrates how you send a message and receive it asynchronously.
- `RequestReply.c`, which illustrates how you send and respond to a message that specifies a reply-to destination.

Before you run any sample programs, you should start the broker. You can display output describing the command-line options for each program by starting the program with the `-help` option.

The directory that contains the sample programs also includes a `README` file that explains how you should run these samples. For example, the following command, runs the program `Producer`. It specifies that the program should connect to the broker running on the host `MyHost` and port `8585`, and that it should send a message to the destination `My Topic`:

```
C: Producer -h MyHost -p 8585 -d MyTopic
```

Client Application Deployment Considerations

When you are ready to deploy your client application, you should make sure the administrator knows your application's needs. The checklist in [Table 1-4](#) shows the basic information required. Consult with your administrator to determine the exact information needed. In some cases, it might be useful to provide a range of values rather than a specific value. Refer to the *Message Queue Administration Guide* for additional information about attribute names and default values.

Table 1-4 Checklist for the Message Queue Administrator

Configuring physical destinations:

- Type:
- Name:
- Attributes:
- Maximum number of messages expected:
- Maximum size of messages expected:
- Maximum message bytes expected:

Configuring Dead Message Queue

- Place dead messages on Dead Message Queue
 - Log the placement of messages on the Dead Message Queue
 - Discard the body of messages placed on the Dead Message Queue
-

Using the C API

This chapter describes how to use C functions to accomplish specific tasks and provides brief code samples to illustrate some of these tasks. (For clarity, the code examples shown in the following sections omit a function call status check.)

Following a brief discussion of overall design and a summary of client tasks, the topics covered include the following:

- [“Message Queue C Client Setup Operations” on page 26](#)
- [“Working With Properties” on page 28](#)
- [“Working With Connections” on page 32](#)
- [“Working With Sessions and Destinations” on page 39](#)
- [“Working With Messages” on page 44](#)
- [“Error Handling” on page 53](#)
- [“Memory Management” on page 54](#)
- [“Logging” on page 55](#)

This chapter does not provide exhaustive information about each function. For detailed function information, please see the description of that function in [Chapter 4, “Reference” on page 73](#).

For information on building Message Queue C programs, see [Chapter 3, “Client Design Issues” on page 57](#).

Message Queue C Client Setup Operations

The general procedures for producing and consuming messages are introduced below. The procedures have a number of common steps which need not be duplicated if a client is both producing and consuming messages.

► To Set Up a Message Queue C Client to Produce Messages

1. Call the `MQCreateProperties` function to get a handle to a properties object.
2. Use one or more of the `MQSet...Property` functions to set connection properties that specify the name of the broker, its port number, and its behavior.
3. Use the `MQCreateConnection` function to create a connection.
4. Use the `MQCreateSession` function to create a session and to specify its acknowledge mode and its receive mode. If the session will be used only for producing messages, use the receive mode `MQ_SESSION_SYNC_RECEIVE` to avoid creating a thread for asynchronous message delivery.
5. Use the `MQCreateDestination` function to specify a physical destination on the broker. The destination name you specify must be the same as the name of the physical destination.
6. Use the `MQCreateMessageProducer` function or the `MQCreateMessageProducerForDestination` function to create a message producer. (If you plan to send a lot of messages to the same destination, you should use the `MQCreateMessageProducerForDestination` function.)
7. Use the `MQCreateBytesMessage` function or the `MQCreateTextMessage` function to get a newly created message handle.
8. Call the `MQCreateProperties` function to get a handle to a properties object that will describe the message header properties. This is only required if you want to set a message header property.
9. Use one or more of the `MQSet...Property` functions to set properties that specify the value of the message header properties you want to set.
10. Use the `MQSetMessageHeaders` function, passing a handle to the properties object you created in [Step 8](#) and [Step 9](#).
11. Repeat [Step 8](#) if you want to define custom message properties, and then use the `MQSetMessageProperties` function to set these properties for your message.

12. Use the `MQSetMessageReplyTo` function if you want to specify a destination where replies to the message are to be sent.
13. Use one of the `MQSendMessage...` functions to send the message.

➤ **To Set Up a Message Queue C Client to Consume Messages Synchronously**

1. Call the `MQCreateProperties` function to get a handle to a properties object.
2. Use one or more of the `MQSet...Property` functions to set connection properties that specify the name of the broker, its port number, and its behavior.
3. Use the `MQCreateConnection` function to create a connection.
4. Use the `MQCreateSession` function to create a session and to specify its receive mode. Specify `MQ_SESSION_SYNC_RECEIVE` for a synchronous session.
5. Use the `MQCreateDestination` function to specify a destination on the broker from which the consumer is to receive messages. The destination name you specify must be the same as the name of the physical destination.
6. Use the `MQCreateMessageConsumer` function or the `MQCreateDurableMessageConsumer` function to create a consumer.
7. Use the `MQStartConnection` function to start the connection.
8. Use one of the `MQReceiveMessage...` functions to start message delivery.

➤ **To Set Up a Message Queue C Client to Consume Messages Asynchronously**

1. Call the `MQCreateProperties` function to get a handle to a properties object.
2. Use one or more of the `MQSet...Property` functions to set connection properties that specify the name of the broker, its port number, and its behavior.
3. Use the `MQCreateConnection` function to create a connection.
4. Use the `MQCreateSession` function to create a session and to specify its acknowledge mode and its receive mode. Specify `MQ_SESSION_ASYNC_RECEIVE` for asynchronous message delivery.
5. Use the `MQCreateDestination` function to specify a destination on the broker from which the consumer is to receive messages. The logical destination name you specify must be the same as the name of the physical destination.

6. Write a callback function of type `MQMessageListenerFunc` that will be called when the broker starts message delivery. In the body of this callback function, use the functions listed in [Table 2-11 on page 52](#), to process the contents of the incoming message.
7. Use the `MQCreateAsyncMessageConsumer` function or the `MQCreateAsyncDurableMessageConsumer` function to create a consumer.
8. Use the `MQStartConnection` function to start the connection and message delivery.

Working With Properties

When you create a connection, set message header properties, or set user-defined message properties, you must pass a handle to a properties object. You use the `MQCreateProperties` function to create this object and to obtain a handle to it. When you receive a message, you can use specific `MQGet...Property` functions to obtain the type and value of each message property.

This section describes the functions you use to set and get properties. A *property* is defined as a key-value pair.

Setting Connection and Message Properties

You use the functions listed in [Table 2-1](#) to create a handle to a properties object, and to set properties. You can use these functions to create and define properties for connections or for individual messages.

Table 2-1 Functions Used to Set Properties

Function	Description
MQCreateProperties	Creates a properties object and passes back a handle to it.
MQSetBoolProperty	Sets an <code>MQBool</code> property.
MQSetStringProperty	Sets an <code>MQString</code> property.
MQSetInt8Property	Sets an <code>MQInt8</code> property.
MQSetInt16Property	Sets an <code>MQInt16</code> property.
MQSetInt32Property	Sets an <code>MQInt32</code> property.
MQSetInt64Property	Sets an <code>MQInt64</code> property.

Table 2-1 Functions Used to Set Properties (*Continued*)

Function	Description
<code>MQSetFloat32Property</code>	Sets an <code>MQFloat32</code> property.
<code>MQSetFloat64Property</code>	Sets an <code>MQFloat64</code> property.

► To Set Properties for a Connection

1. Call the `MQCreateProperties` function to get a handle to a newly created properties object.
2. Call one of the `MQSet...Property` functions to set one of the connection properties listed in [Table 4-2 on page 77](#). At a minimum, you must specify the name of the host of the broker to which you want to connect and its port number.

Which function you call depends on the type of the property you want to set; for example, to set an `MQString` property, you call the `MQSetStringProperty` function; to set an `MQBool` property, you call the `MQSetBoolProperty` function; and so on. Each function that sets a property requires that you pass a key name and value; these are listed and described in [Table 4-2](#).

3. When you have set all the properties you want to define for the connection, you can then create the connection, by calling the `MQCreateConnection` function.

Once the connection is created with the properties you specify, you cannot change its properties. If you need to change connection properties after you have created a connection, you will need to destroy the old connection and its associated objects and create a new one with the desired properties. It is a good idea to think through the desired behavior before you create a connection.

[Code Example 2-1](#) illustrates how you create a properties handle and how you use it for setting connection properties.

Code Example 2-1 Setting Connection Properties

```
MQStatus status;
MQPropertiesHandle propertiesHandle = MQ_INVALID_HANDLE;

status = (MQCreateProperties(&propertiesHandle);

status = (MQSetStringProperty(propertiesHandle,
                               MQ_BROKER_HOST_PROPERTY, "localhost"));
```

Code Example 2-1 Setting Connection Properties (*Continued*)

```

status = (MQSetInt32Property(propertiesHandle,
    MQ_BROKER_PORT_PROPERTY, 7676));

status = MQSetStringProperty(propertiesHandle,
    MQ_CONNECTION_TYPE_PROPERTY, "TCP");

```

The Message Queue C client runtime sets the connection properties that specify the name and version of the Message Queue product; you can retrieve these using the [MQGetMetaData](#) function. These properties are described at the end of [Table 4-2](#), starting with `MQ_NAME_PROPERTY`.

► To Set Message Properties

Set message properties and message header properties using the same procedure you used to set connection properties. You can set the following message header properties for sending a message:

- `MQ_CORRELATION_ID_HEADER_PROPERTY`
- `MQ_MESSAGE_TYPE_HEADER_PROPERTY`

For more information, see the description of the [MQSetMessageProperties](#) function.

Getting Message Properties

When you receive a message, if you are interested in the message properties, you need to obtain a handle to the properties object associated with that message:

- Use the `MQGetMessageProperties` function to obtain a handle to the properties object for user-defined properties.
- If you are interested in any message header properties, use the `MQGetMessageHeaderProperties` function to obtain a handle to the header properties. See [Table 4-5 on page 134](#).

Having obtained the handle, you can iterate through the properties and then use the appropriate `MQGet...Property` function to determine the type and value of each property.

[Table 2-2](#) lists the functions you use to iterate through a properties handle and to obtain the type and value of each property.

Table 2-2 Functions Used to Get Message Properties

Function	Description
<code>MQPropertiesKeyIterationStart</code>	Starts the iteration process through the specified properties handle
<code>MQPropertiesKeyIterationHasNext</code>	Returns <code>MQ_TRUE</code> if there are additional property keys left in the iteration.
<code>MQPropertiesKeyIterationGetNext</code>	Passes back the address of the next property key in the referenced property handle.
<code>MQGetPropertyType</code>	Gets the type of the specified property.
<code>MQGetBoolProperty</code>	Gets the value of the specified <code>MQBool</code> type property.
<code>MQGetStringProperty</code>	Gets the value of the specified <code>MQString</code> type property.
<code>MQGetInt8Property</code>	Gets the value of the specified <code>MQInt8</code> type property.
<code>MQGetInt16Property</code>	Gets the value of the specified <code>MQInt16</code> type property.
<code>MQGetInt32Property</code>	Gets the value of the specified <code>MQInt32</code> type property.
<code>MQGetInt64Property</code>	Gets the value of the specified <code>MQInt64</code> type property.
<code>MQGetFloat32Property</code>	Gets the value of the specified <code>MQFloat32</code> type property.
<code>MQGetFloat64Property</code>	Gets the value of the specified <code>MQFloat64</code> type property.

► To Iterate Through a Properties Handle

1. Start the process by calling the `MQPropertiesKeyIterationStart` function.
2. Loop using the `MQPropertiesKeyIterationHasNext` function.
3. Extract the name of each property key by calling the `MQPropertiesKeyIterationGetNext` function.
4. Determine the type of the property value for a given key by calling the `MQGetPropertyType` function.
5. Use the appropriate `MQGet...Property` function to find the value of the specified property key and type.

If you know the property key, you can just use the appropriate `MQGet...Property` function to get its value. [Code Example 2-2](#) illustrates how you implement these steps.

Code Example 2-2 Getting Property Values for a Message Header

```

MQStatus status;

MQPropertiesHandle headersHandle = MQ_INVALID_HANDLE;

MQBool redelivered;

ConstMQString my_msgtype;

status = (MQGetMessageHeaders(messageHandle, &headersHandle));

status = (MQGetBoolProperty(headersHandle,
    MQ_REDELIVERED_HEADER_PROPERTY, &redelivered));

status = MQGetStringProperty(headersHandle,
    MQ_MESSAGE_TYPE_HEADER_TYPE_PROPERTY, &my_msgtype);

```

Working With Connections

All messaging occurs within the context of a connection: the behavior of the connection is defined by the properties set for that connection. You use the functions listed in [Table 2-3](#) to create, start, stop, and close a connection.

Table 2-3 Functions Used to Work with Connections

Function	Description
MQInitializeSSL	Initializes the SSL library. You must call this function before you create any connection that uses SSL.
MQCreateConnection	Creates a connection and passes back a handle to it.
MQStartConnection	Starts the specified connection and starts or resumes delivery of messages.
MQStopConnection	Stops the specified connection.
MQGetMetaData	Returns a handle to name and version information for the Message Queue product.
MQCloseConnection	Closes the specified connection.

Before you create a connection, you must do the following:

- Define the connection properties. See [“Setting Connection and Message Properties” on page 28](#) for more information.
- Specify a user name and password for the connection. See the *Message Queue Administration Guide* for information on how to set up users.
- Write a connection exception listener function. You will need to pass a reference to this listener when you create the connection. This function will be called synchronously when a connection exception occurs for this connection. For more information, see [“Callback Type for Connection Exception Handling” on page 83](#).
- If you want a secure connection, call the `MQInitializeSSL` function to initialize the SSL library. See [“Working With Secure Connections” on page 35](#) for more information.

When you have completed these steps, you are ready to call `MQCreateConnection` to create a connection. After you create the connection, you can create a session as described in [“Working With Sessions and Destinations” on page 39](#).

When you send a message, you do not need to start the connection explicitly by calling `MQStartConnection`. You *do* need to call `MQStartConnection` before the broker can deliver messages to a consumer.

If you need to halt delivery in the course of processing messages, you can call the `MQStopConnection` function.

Defining Connection Properties

Connection properties specify the following information:

- The host name and port of the broker to which you want to connect
- The transport protocol of the connection service used by the client
- How broker and client acknowledgements are handled to support messaging reliability
- How message flow is to be managed
- How secure messaging should be implemented

The following sections examine the effect of properties used to manage connection handling, reliability, message flow, and security.

[Table 4-2 on page 77](#) lists and describes all properties of a connection. For information on how to set and change connection properties, see [“Working With Properties” on page 28](#).

Connection Handling

Connections to a message server are specified by a broker host name and port number.

- Set `MQ_BROKER_NAME_PROPERTY` to specify the broker name.
- Set `MQ_BROKER_PORT_PROPERTY` to specify the broker port.
- Set the connection property `MQ_CONNECTION_TYPE_PROPERTY` to specify the underlying transport protocol. Possible values are TCP or SSL.

The `MQ_PING_INTERVAL_PROPERTY` also affects connection handling. This property is set to the interval (in seconds) that the connection can be idle before the C client runtime pings the broker to test whether the connection is still alive. This property is useful for either producers who use the connection infrequently or for clients who are exclusive consumers, passively waiting for messages to arrive. The default value is 30 seconds. Setting an interval that is too low may result in some performance loss. The minimum permitted value is 1 second to prevent this from happening.

Currently, the C-API does not support auto-reconnect or failover, which allows the client runtime to automatically reconnect to a broker if a connection fails.

Reliability

Two connection properties enable the acknowledgement of messages sent to the broker and of messages received from the broker. These are described in [“Message Acknowledgement” on page 40](#). In addition to setting these properties, you can also set `MQ_ACK_TIMEOUT_PROPERTY`, which determines the maximum time that the client runtime will wait for any broker acknowledgement before throwing an exception.

Flow Control

A number of connection properties determine the use of Message Queue control messages by the client runtime. Messages sent and received by Message Queue clients and Message Queue control messages pass over the same client-broker connection. Because of this, delays may occur in the delivery of control messages, such as broker acknowledgements, if these are held up by the delivery of JMS messages. To prevent this type of congestion, Message Queue meters the flow of JMS messages across a connection.

- Set `MQ_CONNECTION_FLOW_COUNT_PROPERTY` to specify the number of Message Queue messages in a metered batch. When this number of messages is delivered to the client runtime, delivery is temporarily suspended, allowing any control messages that had been held up to be delivered. Message delivery is resumed upon notification by the client runtime, and continues until the count is again reached.
- `MQ_CONNECTION_FLOW_LIMIT_PROPERTY` specifies the maximum number of unconsumed messages that can be delivered to a client runtime. When the number of messages reaches this limit, delivery stops and resumes only when the number of unconsumed messages drops below the specified limit. This helps a consuming client that is taking a long time to process messages from being overwhelmed with pending messages that might cause it to run out of memory.
- `MQ_CONNECTION_FLOW_LIMIT_ENABLED_PROPERTY` specifies whether the value `MQ_CONNECTION_FLOW_LIMIT_PROPERTY` is used to control message flow.

You should keep the value of `MQ_CONNECTION_FLOW_COUNT_PROPERTY` low if the client is doing operations that require many responses from the broker; for example, the client is using the `CLIENT_ACKNOWLEDGE` or `AUTO_ACKNOWLEDGE` modes, persistent messages, transactions, or if the client is adding or removing consumers. You can increase the value of `MQ_CONNECTION_FLOW_COUNT_PROPERTY` without compromising performance if the client has only simple consumers on a connection using `DUPS_OK` mode.

The C API does not currently support consumer-based flow control.

Working With Secure Connections

Establishing a secure connection between the client and the broker requires both the administrator and the developer to do some additional work. The administrator's work is described in the *Message Queue Administration Guide*. In brief, it requires that the administrator do the following:

- Generate certificates (self-signed or signed by a certificate authority) and add those certificates to the broker's keystore
- Enable the `ssljms` connection service in the broker
- Provide the password to the certificate keystore when starting the broker

The developer must also do some work to configure the client for secure messaging. The work required depends on whether the broker is trusted (the default setting) and on whether the developer wants to provide an additional means of verification if the broker is not trusted and the initial attempt to create a secure connection fails.

The MessageQueue C-API library uses NSS to support the SSL transport protocol between the Message Queue C client and the Message Queue broker. The developer must take care if the client application using secure Message Queue connections uses NSS (for other purposes) directly as well and does NSS initialization. For additional information, see [“Coordinating NSS Initialization” on page 37](#).

Configuring the Client for Secure Communication

By default the `MQ_SSL_BROKER_IS_TRUSTED` property is set to `true`, and this means that the Message Queue client runtime will accept any certificate that is presented to it.

To establish a secure connection, a client must do the following:

1. Set the `MQ_CONNECTION_TYPE_PROPERTY` to `SSL`.
2. If you want the runtime to check the broker’s certificate, set the `MQ_SSL_BROKER_IS_TRUSTED` property to `false`. Otherwise, you can leave it to its default (`true`) value.
3. Generate the NSS files `CertN.db`, `keyN.db`, and `secmod.db` using the certificate database tool `certutil`. You can find this tool at the following locations:
 - o Solaris: `/usr/sfw/bin`
 - o Linux: `/opt/sun/private/bin`
 - o Windows: `IMQ_HOME\bin`

For directions and an example of using this tool, see

<http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html>

4. Note the path name of the directory that contains the NSS files you generated in [Step 3](#).
5. If you have set the `MQ_SSL_BROKER_IS_TRUSTED` property to `false`, use the `certutil` tool to import the root certificate of the authority certifying the broker into the database files you generated in [Step 3](#).

Make sure that the `MQ_BROKER_HOST_PROPERTY` value is set to the same value as the (CN) common name in the broker’s certificate.

6. If you have set the `MQ_SSL_BROKER_IS_TRUSTED` property to `false`, you have the option of enabling broker fingerprint-based verification in case authorization fails. For details, see [“Verification Using Fingerprints” on page 37](#).
7. Call the function `MQInitializeSSL` once (and only once) before creating the connection, and pass the name of the directory that contains the NSS files you generated in [Step 3](#). If the broker is trusted, these files can be empty.

You must call this function before you create *any* connection to the broker, including connections that do not use SSL.

Verification Using Fingerprints

If certificate authorization fails when the broker is using a certificate authority, it is possible to give the client runtime another means of establishing a secure connection by comparing broker certificate fingerprints. If the fingerprints match, the connection is granted; if they do not match, the attempt to create the connection will fail.

► To Set Up Fingerprint Certification, Do the Following:

1. Set the broker connection property `MQ_SSL_CHECK_BROKER_FINGERPRINT` to `true`.
2. Retrieve the broker’s certificate fingerprint by using the `java keytool -list` option on the broker’s keystore file:

You will use the output of this command as the value for the connection property `MQ_SSL_BROKER_CERT_FINGERPRINT` in [Step 3](#). For example, if the output contains a value like the following:

```
Certificate fingerprint (MD5):
F6:A5:C1:F2:E6:63:40:73:97:64:39:6C:1B:35:0F:8E
```

You would specify this value for `MQ_SSL_BROKER_CERT_FINGERPRINT`.

3. Set the connection property `MQ_SSL_BROKER_CERT_FINGERPRINT` to the value obtained in [Step 2](#).

Coordinating NSS Initialization

If your application uses NSS directly, other than to support Message Queue secure communication, you need to coordinate NSS initialization with the Message Queue C-API library. There are two cases to consider:

- Your application does not use secure Message Queue connections.

In this case, you should do your application’s NSS initialization before calling `MQCreateConnection` to create any connection to the Message Queue broker.

- Your application does use secure Message Queue connections.

In this case, you should follow the procedure outlined below before calling `MQCreateConnection` to create any Message Queue connection.

► To Coordinate NSS Initialization

1. Call the function `MQInitializeSSL`. (You must specify the path to the directory containing the NSS files as the `certdbpath` parameter to this function.)

Your application's use of NSS must specify the same `certdbpath` value for the location of its NSS files. (That is, the certificates needed by your application must be located in the same directory as the certificates needed by Message Queue.)

Internally, the function `MQInitializeSSL` does the following:

- Calls the function `NSS_Init(certdbpath)`.
 - Sets `DOMESTIC` cipher policy using the function `NSS_SetDomesticPolicy()`.
 - Enables all cipher suites, including `RSA_NULL_MD5` by calling the function `SSL_CipherPrefSetDefault(SSL_RSA_WITH_NULL_MD5, PR_TRUE)`.
 - Calls the function `SSL_ClearSessionCache()`.
2. If your application needs different cipher suite settings, after you call the `MQInitializeSSL()` function, you can modify the cipher suites by calling the function `SSL_CipherPrefSetDefault`. However, note that these changes will affect your secure connection to the Message Queue broker as well.

Shutting Down Connections

In order to do an orderly shutdown, you need to close the connection by calling `MQCloseConnection` and then to free the memory associated with the connection by calling the `MQFreeConnection` function.

- Closing the connection closes all sessions, producers, and consumers created from this connection. This also forces all threads associated with this connection that are blocking in the library to return.

- After all the application threads associated with this connection and its descendant sessions, producers, consumers, etc. have returned, the application can call the `MQFreeConnection` function to release all resources associated with the connection.

To get information about a connection, call the `MQGetMetaData` function. This returns name and version information for the Message Queue product.

Working With Sessions and Destinations

A session is a single-threaded context for producing and consuming messages. You can create multiple producers and consumers for a session, but you are restricted to using them serially. In effect, only a single logical thread of control can use them. A session supports reliable delivery through acknowledgment options or by using transactions.

Table 2-4 describes the functions you use to create and manage sessions.

Table 2-4 Functions Used to Work with Sessions

Function	Description
<code>MQCreateSession</code>	Creates the specified session and passes back a handle to it.
<code>MQGetAcknowledgeMode</code>	Passes back the acknowledgement mode of the specified session.
<code>MQRecoverSession</code>	Stops message delivery and restarts message delivery with the oldest unacknowledged message. (For non-transacted sessions.)
<code>MQRollBackSession</code>	Rolls back a transaction associated with the specified session.
<code>MQCommitSession</code>	Commits a transaction associated with the specified session.
<code>MQCloseSession</code>	Closes the specified session.

Creating a Session

The `MQCreateSession` function creates a new session and initializes a handle to it in the `sessionHandle` parameter. The number of sessions you can create for a single connection is limited only by system resources. You can create a session after you have created a connection.

When you create a session, you specify whether it is transacted, the acknowledge mode, and the receive mode. After you create a session, you can create the producers, consumers, and destinations that use the session context to do their work.

Transacted Sessions

If you specify that a session be transacted, the acknowledge mode is ignored. Within a transacted session, the broker tracks sends and receives, completing these operations only when the client issues a call to commit the transaction. If a send or receive operation fails, an exception is raised. Your application can handle the exception by ignoring it, retrying it, or rolling back the entire transaction. When a transaction is committed, all the successful operations are completed. When a transaction is rolled back, all successful operations are cancelled.

The scope of a transaction is always a single session. That is, one or more producer or consumer operations performed in the context of a single session can be grouped into a single local transaction.

Since transactions span only a single session, you cannot have an end-to-end transaction encompassing both the production and consumption of a message. (In other words, the delivery of a message to a destination and the subsequent delivery of the message to a client cannot be placed in a single transaction.)

Message Queue does not support distributed transactions for C clients.

Message Acknowledgement

Both messages that are sent and messages that are received can be acknowledged.

In the case of message producers, if you want the broker to acknowledge its having received a non-persistent message (to its physical destination), you must set the connection's `MQ_ACK_ON_PRODUCE_PROPERTY` to `MQ_TRUE`. If you do so, the sending function will return only after the broker has acknowledged receipt of the message. By default, the broker acknowledges receipt of persistent messages.

Acknowledgements on the consuming side means that the client runtime acknowledges delivery and consumption of all messages from a physical destination before the message service deletes the message from that destination. You can specify one of the following acknowledge modes for the consuming session when you create that session.

- `MQ_AUTO_ACKNOWLEDGE` specifies that the session automatically acknowledge each message consumed by the client.
- `MQ_CLIENT_ACKNOWLEDGE` specifies that the client must explicitly acknowledge messages by calling `MQAcknowledgeMessages`. In this case, all messages are acknowledged that have been consumed up to the point where the acknowledge function is called. (This could include messages consumed asynchronously by many different message listeners in that session, independent of the order in which they were consumed.)
- `MQ_DUPS_OK_ACKNOWLEDGE` specifies that the session acknowledges receipt of messages after each ten messages are consumed. It does not guarantee that messages are delivered and consumed only once.

(The setting of the connection property `MQ_ACK_ON_ACKNOWLEDGE_PROPERTY` also determines the effect of some of these acknowledge modes. For more information, see [Table 4-2 on page 77](#).)

NOTE In the `DUPS_OK_ACKNOWLEDGE` mode, the session does not wait for broker acknowledgements. This option can be used in Message Queue C clients for which duplicate messages are not a problem. Also, you can call the `MQRecoverSession` function to explicitly request redelivery of messages that have been received but not yet acknowledged by the client. When redelivering such messages, the broker will set the header field `MQ_REDDELIVERED_HEADER_PROPERTY`.

Receive Mode

You can specify a session's receive mode as either `MQ_SESSION_SYNC_RECEIVE` or `MQ_SESSION_ASYNC_RECEIVE`. If the session you create will be used for sending messages only, you should specify `MQ_SESSION_SYNC_RECEIVE` for its receive mode for optimization because the asynchronous receive mode automatically allocates an additional thread for the delivery of messages it expects to receive.

Managing a Session

Managing a session involves using threads appropriately for the type of session (synchronous or asynchronous) and managing message delivery for both transacted and nontransacted sessions. For more information about thread management, see “[Managing Threads](#)” on page 60.

- For a session that is not transacted, use the [MQRecoverSession](#) function to restart message delivery with the last unacknowledged message.
- For a session that is transacted, use the [MQRollBackSession](#) function to roll back any messages that were delivered within this transaction. Use the [MQCommitSession](#) function to commit all messages associated with this transaction.
- Use the [MQCloseSession](#) function to close a session and all its associated producers and consumers. This function also frees memory allocated for the session.

You can get information about a session’s acknowledgment mode by calling the [MQGetAcknowledgeMode](#) function.

Creating Destinations

After creating a session, you can create destinations or temporary destinations for the messages you want to send. [Table 2-5](#) lists the functions you use to create and to get information about destinations.

Table 2-5 Functions Used to Work with Destinations

Functions	Description
MQCreateDestination	Creates a destination and initializes a handle to it.
MQCreateTemporaryDestination	Creates a temporary destination and initializes a handle to it.
MQGetDestinationType	Returns the type (queue or topic) of the specified destination.

A *destination* refers to where a message is destined to go. A *physical destination* is a JMS message service entity (a location on the broker) to which producers send messages and from which consumers receive messages. The message service provides the routing and delivery for messages sent to a physical destination.

When a Message Queue C client creates a destination programmatically using the `MQCreateDestination` function, a destination name must be specified. The function initializes a handle to a destination data type that holds the identity (name) of the destination. The important thing to remember is that this function does *not* create the physical destination on the broker; this must be done by the administrator. The destination that is created programmatically however *must* have the exact same name and type as the physical destination created on the broker. For example, if you use the `MQCreateDestination` function to create a queue destination called `myMailQDest`, the administrator has to create a physical destination on the broker named `myMailQDest`.

Destination names starting with “mq” are reserved and should not be used by client programs.

Programming Domains

When you create a destination, you must also specify its type: `MQ_QUEUE_DESTINATION` or `MQ_TOPIC_DESTINATION`. See the *Message Queue Technical Overview* for a discussion of these two types of destinations and how to choose the type that suits your needs.

Auto-Created Destinations

By default, the `imq.autocreate.topic` and `imq.autocreate.queue` broker properties are turned on. In this case, which is more convenient in a development environment, the broker automatically creates a physical destination whenever a message consumer or message producer attempts to access a non-existent destination. The auto-created physical destination will have the same name as that of the destination you created using the `MQCreateDestination` function.

Temporary Destinations

You use the `MQCreateTemporaryDestination` to create a temporary destination. You can use such a destination to implement a simple request/reply mechanism. When you pass the handle of a temporary destination to the `MQSetMessageReplyTo` function, the consumer of the message can use that handle as the destination to which it sends a reply.

Temporary destinations are explicitly created by client applications and are automatically deleted when the connection is closed. They are maintained (and named) by the broker only for the duration of the connection for which they are created. Temporary destinations are system-generated uniquely for their connection and only their own connection is allowed to create message consumers for them.

Getting Information About Destinations

Use the `MQGetDestinationType` function to determine the type of a destination: queue or topic. There may be times when you do not know the type of the destination to which you are replying; for example, when you get a handle from the `MQGetMessageReplyTo` function. Because the semantics of queue and topic destinations differ, you need to determine the type of a destination in order to reply appropriately.

Working With Messages

This section describes how you use the C-API to complete the following tasks:

- Compose a message
- Send a message
- Receive a message
- Process a message

Composing Messages

You can create either a text message or a bytes message. A message, whether text or bytes, is composed of a header, properties, and a body.

[Table 2-6](#) lists the functions you use to construct messages.

Table 2-6 Functions Used to Construct Messages

Function	Description
<code>MQCreateBytesMessage</code>	Creates an <code>MQ_BYTES_MESSAGE</code> message.
<code>MQCreateTextMessage</code>	Creates an <code>MQ_TEXT_MESSAGE</code> message.
<code>MQSetMessageHeaders</code>	Sets message header properties. (Optional)
<code>MQSetMessageProperties</code>	Sets user-defined message properties.
<code>MQSetStringProperty</code>	Sets the body of an <code>MQ_TEXT_MESSAGE</code> message.
<code>MQSetBytesMessageBytes</code>	Sets the body of an <code>MQ_BYTES_MESSAGE</code> message.
<code>MQSetMessageReplyTo</code>	Specifies the destination where replies to this message should be sent.

Message Header

A header is required of every message. Header fields contain values used for routing and identifying messages.

Some header field values are set automatically by Message Queue during the process of producing and delivering a message, some depend on settings specified when message producers send a message, and others are set on a message-by-message basis by the client using the `MQSetMessageHeader` function.

[Table 2-7](#) lists the header fields defined (and required) by JMS and their corresponding names, as defined by the C-API.

Table 2-7 JMS-defined Message Header

JMS Message Header Field	C-API Message Header Property Name
JMSDestination	Defined implicitly when a producer sends a message to a destination, or when a consumer receives a message from a destination.
JMSDeliveryMode	MQ_PERSISTENT_HEADER_PROPERTY
JMSExpiration	MQ_EXPIRATION_HEADER_PROPERTY
JMSPriority	MQ_PRIORITY_HEADER_PROPERTY
JMSMessageID	MQ_MESSAGE_ID_HEADER_PROPERTY
JMSTimestamp	MQ_TIMESTAMP_HEADER_PROPERTY
JMSRedelivered	MQ_REDELIVERED_HEADER_PROPERTY
JMSCorrelationID	MQ_CORRELATION_ID_HEADER_PROPERTY
JMSReplyTo	Set by the <code>MQSetMessageReplyTo</code> function, and obtained by the <code>MQGetMessageReplyTo</code> function.
JMSType	MQ_MESSAGE_TYPE_HEADER_PROPERTY

For additional information about each property type and who sets it, see [Table 4-6 on page 173](#).

Message Body Types

JMS specifies six classes (or types) of messages. The C-API supports only two of these types, as described in [Table 2-8](#). If a Message Queue C client expects to receive messages from a Message Queue Java client, it will be unable to process messages whose body types are other than those described in [Table 2-8](#). It will also be unable to process messages that are compressed by the Message Queue Java client runtime.

Table 2-8 C-API Message Body Types

Type	Description
TextMessage	A message whose body contains an MQString string, for example an XML message.
BytesMessage	A message whose body contains a stream of uninterpreted bytes.

Composing the Message

Create a message using either the [MQCreateBytesMessage](#) function or the [MQCreateTextMessage](#) function. Either of these functions returns a message handle that you can then pass to the functions you use to set the message body, header, and properties (listed in [Table 2-6](#)).

- Use the [MQSetTextMessageText](#) function to define the body of a text message; use the [MQSetBytesMessageBytes](#) function to define the body of a bytes message.
- Use the [MQSetMessageHeaders](#) to set any message header properties.

The message header can specify up to eight properties; most of these are set by the client runtime when sending the message or are set by the broker. The client can set `MQ_CORRELATION_ID_HEADER_PROPERTY` and `MQ_MESSAGE_TYPE_HEADER_PROPERTY` for sending a message.

- Use the [MQSetMessageProperties](#) function to set any user-defined properties for this message.

When you set message header properties or when you set additional user-defined properties, you must pass a handle to a properties object that you have created using the [MQCreateProperties](#) function. For more information, see [“Working With Properties” on page 28](#).

You can use the `MQSetMessageReplyTo` function to associate a message with a destination that recipients can use for replies. To do this, you must first create a destination that will serve as your reply-to destination. Then, pass a handle to that destination when you call the `MQSetMessageReplyTo` function. The receiver of a message can use the `MQGetMessageReplyTo` function to determine whether a sender has set up a destination where replies are to be sent.

Sending a Message

Messages are sent by a message producer within the context of a connection and a session. Once you have obtained a connection, created a session, and composed your message, you can use the functions listed in [Table 2-9](#) to create a message producer and to send the message.

Which function you choose to send a message depends on the following factors:

- Whether you want the send function to override certain message header properties
 - Send functions whose names end in `Ext` allow you to override default values for priority, time-to-live, and delivery mode header properties.
- Whether you want to send the message to the destination associated with the message producer

If you created a message producer with no specified destination, you must use one of the `...ToDestination` send functions. If you created a message producer with a specified destination, you must use one of the other send functions.

Table 2-9 Functions for Sending Messages

Function	Action
<code>MQCreateMessageProducer</code>	Creates a message producer with no specified destination.
<code>MQCreateMessageProducerForDestination</code>	Creates a message producer with a specified destination.
<code>MQSendMessage</code>	Sends a message for the specified producer.
<code>MQSendMessageExt</code>	Sends a message for the specified producer and allows you to set priority, time-to-live, and delivery mode.
<code>MQSendMessageToDestination</code>	Sends a message to the specified destination.

Table 2-9 Functions for Sending Messages

<code>MQSendMessageToDestinationExt</code>	Sends a message to the specified destination and allows you to set priority, time-to-live, and delivery mode.
--	---

If you send a message using one of the functions that does not allow you to override header properties, the following message header fields are set to default values by the send function.

- `MQ_PERSISTENT_HEADER_PROPERTY` will be set to `MQ_PERSISTENT_DELIVERY`.
- `MQ_PRIORITY_HEADER_PROPERTY` will be set to 4.
- `MQ_EXPIRATION_HEADER_PROPERTY` will be set to 0, which means that the message will never expire.

To override these values, use one of the extended send functions. For a complete list of message header properties, see [Table 4-5 on page 134](#).

Message headers also contain fields that can be set by the sending client; in addition, you can set user-defined message properties as well. For more information, see [“Composing Messages” on page 44](#).

You can set the connection property `MQ_ACK_ON_PRODUCE_PROPERTY` when you create the connection to make sure that the message has reached its destination on the broker:

- By default, the broker acknowledges receiving persistent messages only.
- If you set the property to `MQ_TRUE`, the broker acknowledges receipt of all messages (persistent and non-persistent) from the producing client.
- If you set the property to `MQ_FALSE`, the broker does not acknowledge receipt of any message (persistent or non-persistent) from the producing client.

Note that “acknowledgement” in this case is not programmatic but internally implemented. That is, the client thread is blocked and does not return until the broker acknowledges messages it receives.

An administrator can set a broker limit, `REJECT_NEWEST`, which allows the broker to avert memory problems by rejecting the newest incoming message. If the incoming message is persistent, then an error is returned which the sending client should handle, perhaps by retrying the send a bit later. If the incoming message is not persistent, the client has no way of knowing that the broker rejected it. The broker might also reject a message if it exceeds a specified limit.

Receiving Messages

Messages are received by a message consumer in the context of a connection and a session. In order to receive messages, you must explicitly start the connection by calling the `MQStartConnection` function.

[Table 2-10](#) lists the functions you use to create message consumers and to receive messages.

Table 2-10 Functions Used to Receive Messages

Function	Description
MQCreateMessageConsumer	Creates the specified synchronous consumer and passes back a handle to it.
MQCreateDurableMessageConsumer	Creates a durable synchronous message consumer for the specified destination.
MQCreateAsyncMessageConsumer	Creates an asynchronous message consumer for the specified destination.
MQCreateAsyncDurableMessageConsumer	Creates a durable asynchronous message consumer for the specified destination.
MQUnsubscribeDurableMessageConsumer	Unsubscribes the specified durable message consumer.
MQReceiveMessageNowait	Passes a handle back to a message delivered to the specified consumer if a message is available; otherwise it returns an error.
MQReceiveMessageWait	Passes a handle back to a message delivered to the specified consumer if a message is available; otherwise it blocks until a message becomes available.
MQReceiveMessageWithTimeout	Passes a handle back to a message delivered to the specified consumer if a message is available within the specified amount of time.
MQAcknowledgeMessages	Acknowledges the specified message and all messages received before it on the same session
MQCloseMessageConsumer	Closes the specified consumer.

Working With Consumers

When you create a consumer, you need to make several decisions:

- Do you want to receive messages synchronously or asynchronously?
If you create a synchronous consumer, you can call one of three kinds of receive functions to receive your messages. If you create an asynchronous consumer, you must specify the name of a callback function that the client runtime can call when a message is delivered to the destination for that consumer. For information about the callback function signature, see [“Callback Type for Asynchronous Messaging” on page 81](#).
- If you are consuming messages from a topic, do you want to use a durable or a nondurable consumer?

A durable consumer receives all the messages published to a topic, including the ones published while the subscriber is inactive. A nondurable consumer only receives messages while the subscriber is active.

The broker retains a record of this durable subscription and makes sure that all messages from the publishers to this topic are retained until they are either acknowledged by this durable subscriber or until they have expired. Sessions with durable subscribers must always provide the same client identifier. In addition, each consumer must specify a durable name using the `durableName` parameter, which uniquely identifies (for each client identifier) each durable subscription it creates.

A session’s consumers are automatically closed when you close the session or connection to which they belong. However, messages will be routed to the durable subscriber while it is inactive and delivered when a new durable consumer is recreated. To close a consumer without closing the session or connection to which it belongs, use the [MQCloseMessageConsumer](#) function. If you want to close a durable consumer permanently, you should call the function [MQUnsubscribeDurableMessageConsumer](#) after closing it, to delete state information maintained by the broker on behalf of the durable consumer.

Receiving a Message Synchronously

If you have created a synchronous consumer, you can use one of three receive functions: `MQReceiveMessageNoWait`, `MQReceiveMessageWait`, or `MQReceiveMessageWithTimeOut`. In order to use any of these functions, you must have specified `MQ_SESSION_SYNC_RECEIVE` for the receive mode when you created the session.

When you create a session you must specify one of several acknowledge modes for that session. If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge messages that you have received. If the session is transacted, the acknowledge mode parameter is ignored.

When the receiving function returns, it gives you a handle to the delivered message. You can pass that handle to the functions described in [“Processing a Message” on page 52](#), in order to read message properties and information stored in the header and body of the message.

It is possible that a message can be lost for synchronous consumers in a session using `AUTO_ACKNOWLEDGE` mode if the provider fails. To prevent this possibility, you should either use a transacted session or a session in `CLIENT_ACKNOWLEDGE` mode.

Because distributed applications involve greater processing time, such an application might not behave as expected if it were run locally. For example, calling the `MQReceiveMessageNoWait` function might return `MQ_NO_MESSAGE` even when there is a message available to be retrieved on the broker. See the usage notes provided in the section [“MQReceiveMessageNoWait” on page 149](#) for more information.

Receiving a Message Asynchronously

To receive a message asynchronously, you must create an asynchronous message consumer and pass the name of an `MQMessageListenerFunc` type callback function. (Therefore, you must set up the callback function before you create the asynchronous consumer that will use it.) You should start the connection only after creating an asynchronous consumer. If the connection is already started, you should stop the connection before creating an asynchronous consumer.

You are also responsible for writing the message listener function. Mainly, the function needs to process the incoming message by examining its header, body, and properties, or it needs to pass control to a function that can do this processing. The client is also responsible for freeing the message handle (either from within the listener or from outside of the listener) by calling the `MQFreeMessage` function.

When you create a session you must specify one of several acknowledge modes for that session. If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge messages that you have received.

For more information about the signature and content of a call back function, see [“Callback Type for Asynchronous Messaging” on page 81](#).

When the callback function is called by the session delivery of a message, it gives you a handle to the delivered message. You can pass that handle to the functions described in [“Processing a Message” on page 52](#), in order to read message properties and information stored in the header and body of the message.

Processing a Message

When a message is delivered to you, you can examine the message’s properties, type, headers, and body. The functions used to process a message are described in [Table 2-11](#).

Table 2-11 Functions Used to Process Messages

Function	Description
MQGetMessageHeaders	Gets message header properties.
MQGetMessageProperties	Gets user-defined message properties.
MQGetMessageType	Gets the message type: <code>MQ_TEXT_MESSAGE</code> or <code>MQ_BYTES_MESSAGE</code>
MQGetTextMessageText	Gets the body of an <code>MQ_TEXT_MESSAGE</code> message.
MQGetBytesMessageBytes	Gets the body of an <code>MQ_BYTES_MESSAGE</code> message.
MQGetMessageReplyTo	Gets the destination where replies to this message should be sent.

If you are interested in a message’s header information, you need to call the `MQGetMessageHeaders` function. If you need to read or check any user-defined properties, you need to call the `MQGetMessageProperties` function. Each of these functions passes back a properties handle. For information on how you can read property values, see [“Getting Message Properties” on page 30](#).

Before you can examine the message body, you can call the `MQGetMessageType` function to determine whether the message is a text or bytes message. You can then call the `MQGetTextMessageText`, or the `MQGetBytesMessageBytes` function to get the contents of the message.

Some message senders specify a reply destination for their message. Use the `MQGetMessageReplyTo` function to determine that destination.

Error Handling

Nearly all Message Queue C functions return an `MQStatus` result. You can use this return value to determine whether the function returned successfully and, if not, to determine the cause of the error.

[Table 2-12](#) lists the functions you use to get error information.

Table 2-12 Functions Used in Handling Errors

Function	Description
<code>MQStatusIsError</code>	Returns an <code>MQ_TRUE</code> if the specified <code>MQStatus</code> is an error.
<code>MQGetStatusCode</code>	Returns the error code for the specified <code>MQStatus</code> .
<code>MQGetStatusString</code>	Returns a descriptive string for the specified <code>MQStatus</code> .
<code>MQGetErrorTrace</code>	Returns the calling thread's current error trace or <code>NULL</code> if no error trace is available.

► To Handle Errors in Your Code

1. Call `MQStatusIsError`, passing it an `MQStatus` result for the function whose result you want to test.
2. If the `MQStatusIsError` function returns `MQ_TRUE`, call `MQGetStatusCode` or `MQGetStatusString` to identify the error.
3. If the status code and string information is not sufficient to identify the cause of the error, you can get additional diagnostic information by calling `MQGetErrorTrace` to obtain the calling thread's current error trace if this information is available.

[Chapter 4, "Reference" on page 73](#), lists common errors returned for each function. In addition to these errors, the following error codes may be returned by any Message Queue C function:

- `MQ_STATUS_INVALID_HANDLE`
- `MQ_OUT_OF_MEMORY`
- `MQ_NULL_PTR_ARG`

In addition, the `MQ_TIMEOUT_EXPIRED` can return from any Message Queue C function that communicates with the Message Queue broker if the connection `MQ_ACK_TIMEOUT_PROPERTY` is set to a non-zero value.

Memory Management

[Table 2-13](#) lists the functions you use to free or deallocate memory allocated by the Message Queue-C client library on behalf of the user. Such deallocation is part of normal memory management and will prevent memory leaks.

The functions `MQCloseConnection`, `MQCloseSession`, `MQCloseMessageProducer`, and `MQCloseMessageConsumer` are used to free resources associated with connections, sessions, producers, and consumers.

Table 2-13 Functions Used to Free Memory

Function	Description
MQFreeConnection	Frees memory allocated to the specified connection.
MQFreeDestination	Frees memory allocated to the specified destination.
MQFreeMessage	Frees memory allocated to the specified message.
MQFreeProperties	Frees memory allocated to the specified properties handle.
MQFreeString	Frees memory allocated to the specified <code>MQString</code> .

You should free a connection only after you have closed the connection with the [MQCloseConnection](#) function and after all of the application threads associated with this connection and its dependent sessions, producers, and consumers have returned.

You should not free a connection while an application thread is active in a library function associated with this connection or one of its dependent sessions, producers, consumers, and destinations.

Freeing a connection does not release resources held by a message associated with this connection. You must free memory allocated for this message by explicitly calling the `MQFreeMessage` function.

You should not free a properties handle if the properties handle passed to a function becomes invalid on its return. If you do, you will get an error.

Logging

The Message Queue C-API library uses two environment variables to control execution-time logging:

- `MQ_LOG_FILE` specifies the file to which log messages are directed. If you do not specify a file name for this variable, `stderr` is used. If `MQ_LOG_FILE` is a directory name, it should include a trailing directory separator.

By default, `.n` (where `n` is 0, 1, 2,...) is appended to the actual log file name. This is used as a rotation index, and the indices are used sequentially when the maximum log file size is reached. You can use `%g` to specify a rotation index replacement in `MQ_LOG_FILE` after the last directory separator. Only the last `%g` is used if multiple `%g`'s are specified. the `%g` replacement can be escaped with `%`. The maximum rotation index is 9, and the maximum log file size is 1 MB. These limits are not configurable.

- `MQ_LOG_LEVEL` specifies a numeric level that indicates the detail of logging information needed. A value of -1 specifies that nothing be logged. By default the level is set to 3.

Logging

Client Design Issues

This chapter describes a number of messaging issues that impact Message Queue C client design. It covers the following topics:

- [“Producers and Consumers” on page 57](#)
- [“Using Selectors Efficiently” on page 58](#)
- [“Determining Message Order and Priority” on page 59](#)
- [“Managing Threads” on page 60](#)
- [“Managing Physical Destination Limits” on page 62](#)
- [“Managing the Dead Message Queue” on page 63](#)
- [“Factors Affecting Performance” on page 67](#)

This chapter does not discuss the particulars of the C-API and how to use the data types and functions it defines to create messaging clients. For this information, see [Chapter 2, “Using the C API” on page 25](#).

Producers and Consumers

Aside from the reliability your client requires, the design decisions that relate to producers and consumers include the following:

- Do you want to use a point-to-point or a publish/subscribe domain?

There are some interesting permutations here. There are times when you would want to use publish/subscribe even when you have only one subscriber. Performance considerations might make the point-to-point model more efficient than the publish/subscribe model, when the work of sorting messages between subscribers is too costly. Sometimes these decisions cannot be made in the abstract, but different prototypes must be developed and tested.

- Are you using an asynchronous message consumer that does not get called often or a producer that is seldom used?

You might need to adjust the `MQ_PING_INTERVAL_PROPERTY` when you create your connection, so that your client gets an exception if the connection should fail. For more information see [“Connection Handling” on page 34](#).

- Are you using a synchronous consumer in a distributed application?

You might need to allow a small time interval between connecting and calling the `MQReceiveMessageNoWait` function in order not to miss a pending message. For more information, see usage information in the section [“MQReceiveMessageNoWait” on page 149](#).

Using Selectors Efficiently

The use of selectors can have a significant impact on the performance of your application. It's difficult to put an exact cost on the expense of using selectors since it varies with the complexity of the selector expression, but the more you can do to eliminate or simplify selectors the better.

One way to eliminate (or simplify) selectors is to use multiple destinations to sort messages. This has the additional benefit of spreading the message load over more than one producer, which can improve the scalability of your application. For those cases when it is not possible to do that, here are some techniques that you can use to improve the performance of your application when using selectors:

- Have consumers share selectors. As of version 3.5 of Message Queue, message consumers with identical selectors “share” that selector in the broker, which can significantly improve performance. So if there is a way to structure your application to have some selector sharing, consider doing so.
- Use `IN` instead of multiple string comparisons. For example, the following expression:

```
color IN ('red', 'green', 'white')
```

is much more efficient than this expression

```
color = 'red' OR color = 'green' OR color = 'white'
```

especially if the above expression usually evaluates to false.

- Use `BETWEEN` instead of multiple integer comparisons. For example:

```
size BETWEEN 6 AND 10
```

is generally more efficient than

```
size >= 6 AND size <= 10
```

especially if the above expression usually evaluates to true.

- Order the selector expression so that MQ can short circuit the evaluation. The short circuiting of selector evaluation was added in MQ 3.5 and can easily double or triple performance when using selectors depending on the complexity of the expression.
 - If you have two expressions joined by an `OR`, put the expression that is most likely to evaluate to `TRUE` first.
 - If you have two expressions joined by an `AND`, put the expression that is most likely to evaluate to `FALSE` first.

For example, if `size` is usually greater than 6, but `color` is rarely `red` you would want the order of an `OR` expression to be:

```
size > 6 OR color = 'red'
```

If you are using `AND`:

```
color = 'red' AND size > 6
```

Determining Message Order and Priority

In general, all messages sent to a destination by a single session are guaranteed to be delivered to a consumer in the order they were sent. However, if they are assigned different priorities, a messaging system will attempt to deliver higher priority messages first.

Beyond this, the ordering of messages consumed by a client can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to a number of destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see the *Message Queue Administration Guide*), and message service availability.

Managing Threads

This section addresses a number of thread management issues that you should be aware of in designing and programming a Message Queue C client. It covers the following topics:

- “Message Queue C Runtime Thread Model” on page 60
- “Concurrent Use of Handles” on page 60
- “Single-Threaded Session Control” on page 61
- “Connection Exceptions” on page 62

Message Queue C Runtime Thread Model

The Message Queue C-API library creates the thread(s) needed to provide runtime support for a Message Queue C client. It uses NSPR (Netscape Portable Runtime) GLOBAL threads. NSPR GLOBAL threads are fully compatible with native threads on each supported platform. [Table 3-1](#) shows the thread model that the NSPR GLOBAL threads map to on each platform. For more information on NSPR, please see

<http://www.mozilla.org/projects/nspr/>

Table 3-1 Thread Model for NSPR GLOBAL Threads

Platform	Thread Model
Solaris	pthreads
Linux	pthreads
Windows	Win32 threads (from Microsoft Visual C++ runtime library <code>msvcrt</code>)

Concurrent Use of Handles

[Table 3-2](#) lists the handles (objects) used in a C client program and specifies which of these may be used concurrently and which can only be used by one logical thread at a time.

Table 3-2 Handles and Concurrency

Handle	Supports Concurrent Use
MQDestinationHandle	YES
MQConnectionHandle	YES
MQSessionHandle	NO
MQProducerHandle	NO
MQConsumerHandle	NO
MQMessageHandle	NO
MQPropertiesHandle	NO

Single-Threaded Session Control

A session is a single-threaded context for producing and consuming messages. Multiple threads should not use the same session concurrently nor use the objects it creates concurrently. The only exception to this occurs during the orderly shutdown of the session or its connection when the client calls the `MQCloseSession` or the `MQCloseConnection` function. Follow these guidelines in designing your client:

- If a client wants to have one thread producing messages and other threads consuming messages, the client should use a separate session for its producing thread.
- Do not create an asynchronous message consumer while the connection is in started mode.
- A session created with `MQ_SESSION_ASYNC_RECEIVE` mode uses a single thread to run all its consumers' `MQMessageListenerFunc` callback functions. Clients that want concurrent delivery should use multiple sessions.
- Do not call the `MQStopConnection`, `MQCloseSession`, or the `MQCloseConnection` functions from a `MQMessageListenerFunc` callback function. (These calls will not return until delivery of messages has stopped.)
- Call the `MQFreeConnection` function after `MQCloseConnection` and all of the application threads associated with a connection and its sessions, producers, consumers, etc., have returned.

The Message Queue C runtime library provides one thread to a session in `MQ_SESSION_ASYNC_RECEIVE` mode for asynchronous message delivery to its consumers. When the connection is started, all its sessions that have created asynchronous consumers are dedicated to the thread of control that delivers messages. Client code should not use such a session from another thread of control. The only exception to this is the use of `MQCloseSession` and `MQCloseConnection`.

Connection Exceptions

When a connection exception occurs, the Message Queue C library thread that is provided to the connection calls its `MQConnectionExceptionHandlerFunc` callback if one exists. If an `MQConnectionExceptionHandlerFunc` callback is used for multiple connections, it can potentially be called concurrently from different connection threads.

You should not call the `MQCloseConnection` function in an `MQConnectionExceptionHandlerFunc` callback. Instead the callback function should notify another thread to call `MQCloseConnection` and return.

Managing Physical Destination Limits

When creating a topic or queue destination, the administrator can specify how the broker should behave when certain memory limits are reached. Specifically, when the number of unconsumed messages reaching a physical destination exceeds the number specified with the `maxNumMsgs` property or when the total amount of memory allowed for unconsumed messages exceeds the number specified with the `maxTotalMsgBytes` property, the broker takes one of the following actions, depending on the setting of the `limitBehavior` property:

- slows message producers (`FLOW_CONTROL`)
- throws out the oldest message in memory (`REMOVE_OLDEST`)
- throws out the lowest priority message in memory (`REMOVE_LOW_PRIORITY`)
- rejects the newest messages (`REJECT_NEWEST`)

If the default value `REJECT_NEWEST` is specified for the `limitBehavior` property, the broker throws out the newest messages received when memory limits are exceeded. If the message discarded is a persistent message, the producing client gets an error which you should handle by resending the message later.

If any of the other values is selected for the `limitBehavior` property or if the message is not persistent (or persistent and `MQ_ACK_ON_PRODUCE_PROPERTY` is false), the application client is not notified if a message is discarded. Application clients should let the administrator know how they prefer this property to be set for best performance and reliability.

Managing the Dead Message Queue

When a message is deemed undeliverable, it is automatically placed on a special queue called the dead message queue. A message placed on this queue retains all of its original headers (including its original destination) and information is added to the message's properties to explain why it became a dead message.

- For an introduction to dead messages and the dead message queue, see the *Message Queue Technical Overview*.
- For a description of the destination properties and of the broker properties that control the system's use of the dead message queue, see the *Message Queue Administration Guide*.

This section describes the message properties that you can set or examine programatically to determine the following:

- Whether a dead message can be sent to the dead message queue.
- Whether the broker should log information when a message is destroyed or moved to the dead message queue.
- Whether the body of the message should also be stored when the message is placed on the dead message queue.
- Why the message was placed on the dead message queue and any ancillary information.

(Message Queue 3 2005Q1 clients can set properties related to the dead message queue on messages and send those messages to clients compiled against Message Queue 3.5x or earlier versions. However clients receiving such messages cannot examine these properties without recompiling against Message Queue 3 2005Q1 libraries.)

The dead message queue is automatically created by the system and called `mq.sys.dmq`. You can write a Java program that uses the message monitoring API, described in *Message Queue Developer's Guide for Java Clients*, to determine whether that queue is growing, to examine messages on that queue, and so on.

You can set the properties described in [Table 3-3](#) for any message to control how the broker should handle that message if it deems it to be undeliverable. Note that these message properties are needed only to override default destination, or default broker-based behavior.

Table 3-3 Message Properties Relating to Dead Message Queue

Property	Type	Description
<code>JMS_SUN_PRESERVE_UNDELIVERED</code>	Boolean	<p>For a dead message, the default value of unset, specifies that the message should be handled as specified by the <code>useDMQ</code> property of the destination to which the message was sent.</p> <p>A value of <code>true</code> overrides the setting of the <code>useDMQ</code> property and sends the dead message to the dead message queue.</p> <p>A value of <code>false</code> overrides the setting of the <code>useDMQ</code> property and prevents the dead message from being placed in the dead message queue.</p>
<code>JMS_SUN_LOG_DEAD_MESSAGES</code>	Boolean	<p>The default value of unset, will behave as specified by the broker configuration property <code>imq.destination.logDeadMsgs</code>.</p> <p>A value of <code>true</code> overrides the setting of the <code>imq.destination.logDeadMsgs</code> broker property and specifies that the broker should log the action of removing a message or moving it to the dead message queue.</p> <p>A value of <code>false</code> overrides the setting of the <code>imq.destination.logDeadMsgs</code> broker property and specifies that the broker should not log these actions.</p>

Table 3-3 Message Properties Relating to Dead Message Queue (*Continued*)

Property	Type	Description
JMS_SUN_TRUNCATE_MSG_BODY	Boolean	<p>The default value of unset, will behave as specified by the broker property <code>imq.destination.DMQ.truncateBody</code>.</p> <p>A value of <code>true</code> overrides the setting of the <code>imq.destination.DMQ.truncateBody</code> property and specifies that the body of the message should be discarded when the message is placed in the dead message queue.</p> <p>A value of <code>false</code> overrides the setting of the <code>imq.destination.DMQ.truncateBody</code> property and specifies that the body of the message should be stored along with the message header and properties when the message is placed in the dead message queue.</p>

The properties described in [Table 3-4](#) are set by the client runtime for a message placed in the dead message queue.

Table 3-4 Dead Message Properties

Property	Type	Description
JMSXDeliveryCount	Integer	Specifies the most number of times the message was delivered to a given consumer. This value is set only for <code>ERROR</code> or <code>UNDELIVERABLE</code> messages.
JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP	Long	Specifies the time (in milliseconds) when the message was placed on the dead message queue.

Table 3-4 Dead Message Properties (*Continued*)

Property	Type	Description
JMS_SUN_DMQ_UNDELIVERED_REASON	String	<p>Specifies one of the following values to indicate the reason why the message was placed on the dead message queue:</p> <p>OLDEST LOW_PRIORITY EXPIRED UNDELIVERABLE ERROR</p> <p>If the message was marked dead for multiple reasons, for example it was undeliverable and expired, only one reason will be specified by this property.</p> <p>The <code>ERROR</code> value is returned when a message cannot be delivered due to an internal error; this is an unusual condition. In this case, the sender should just resend the message.</p>
JMS_SUN_DMQ_PRODUCING_BROKER	String	<p>For message traffic in broker clusters: specifies the broker name and port number of the broker that placed the message on the dead message queue. A null value indicates that it was the local broker.</p>
JMS_SUN_DMQ_UNDELIVERED_EXCEPTION	String	<p>Specifies the name of the exception (if the message was dead because of an exception) on either the client or the broker.</p>
JMS_SUN_DMQ_UNDELIVERED_COMMENT	String	<p>An optional comment provided when the message is marked dead.</p>
JMS_SUN_DMQ_BODY_TRUNCATED	Boolean	<p>A value of <code>true</code> indicates that the message body was not stored. A value of <code>false</code> indicates that the message body was stored.</p>

Factors Affecting Performance

Application design decisions can have a significant effect on overall messaging performance. In general, the more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is a significant design consideration. You can maximize performance and throughput by choosing to produce and consume non-persistent messages. On the other hand, you can maximize reliability by producing and consuming persistent messages using a transacted session. Between these extremes are a number of options, depending on the needs of your application. This section describes how these options or factors affect performance. They include the following:

- [Delivery Mode \(Persistent/Non-persistent\)](#)
- [Use of Transactions](#)
- [Acknowledgement Mode](#)
- [Durable vs. Non-Durable Subscriptions](#)
- [Use of Selectors \(Message Filtering\)](#)
- [Message Size](#)
- [Message Body Type.](#)

[Table 3-5](#) summarizes how application design factors affect messaging performance. The table shows two scenarios—a high reliability, low performance scenario and a high performance, low reliability scenario—and the choice of application design factors that characterizes each. Between these extremes, there are many choices and trade-offs that affect both reliability and performance.

Table 3-5 Comparison of High Reliability and High Performance Scenarios

Application Design Factor	High Reliability Low Performance Scenario	High Performance Low Reliability Scenario
Delivery mode	Persistent messages	Non-persistent messages
Use of transactions	Transacted sessions	No transactions
Acknowledgement mode	AUTO_ACKNOWLEDGE or CLIENT_ACKNOWLEDGE	DUPS_OK_ACKNOWLEDGE
Durable/non-durable subscriptions	Durable subscriptions	Non-durable subscriptions
Use of selectors	Message filtering	No message filtering
Message size	Small messages	Large messages

Table 3-5 Comparison of High Reliability and High Performance Scenarios (*Continued*)

Application Design Factor	High Reliability Low Performance Scenario	High Performance Low Reliability Scenario
Message body type	Complex body types	Simple body types

NOTE In the discussion that follows, performance data was generated on a two-CPU, 1002 Mhz, Solaris 8 system, using file-based persistence. The performance test first warmed up the Message Queue broker, allowing the Just-In-Time compiler to optimize the system and the persistent database to be primed.

Once the broker was warmed up, a single producer and a single consumer were created, and messages were produced for 30 seconds. The time required for the consumer to receive all produced messages was recorded, and a throughput rate (messages per second) was calculated. This scenario was repeated for different combinations of the application design factors shown in [Table 3-5](#).

Delivery Mode (Persistent/Non-persistent)

Persistent messages guarantee message delivery in case of message server failure. The broker stores these message in a persistent store until all intended consumers acknowledge they have consumed the message.

Broker processing of persistent messages is slower than for non-persistent messages for the following reasons:

- A broker must reliably store a persistent message so that it will not be lost should the broker fail.
- The broker must confirm receipt of each persistent message it receives. Delivery to the broker is guaranteed once the method producing the message returns without an exception.
- Depending on the client acknowledgment mode, the broker might need to confirm a consuming client's acknowledgement of a persistent message.

The differences in performance for persistent and non-persistent modes can be significant--about 25% faster for non-persistent messages.

Use of Transactions

A transaction guarantees that all messages produced in a transacted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit. The C-API supports local transactions.

A message produced or acknowledged in a transacted session is slower than in a non-transacted session for the following reasons:

- Additional information must be stored with each produced message.
- In some situations, messages in a transaction are stored when normally they would not be. For example, a persistent message delivered to a topic destination with no subscriptions would normally be deleted, however, at the time the transaction is begun, information about subscriptions is not available.
- Information on the consumption and acknowledgement of messages within a transaction must be stored and processed when the transaction is committed.

Acknowledgement Mode

Other than using transactions, you can ensure reliable delivery by having the client acknowledge receiving a message. If a session is closed without the client acknowledging the message or if the message server fails before the acknowledgment is processed, the broker redelivers that message, setting the `MQ_REDELIVERED_HEADER_PROPERTY` message header.

For a non-transacted session, the client can choose one of three acknowledgement modes, each of which has its own performance characteristics:

- `AUTO_ACKNOWLEDGE`. The system automatically acknowledges a message once the consumer has processed it. This mode guarantees at most one redelivered message after a provider failure.
- `CLIENT_ACKNOWLEDGE`. The application controls the point at which messages are acknowledged. All messages that have been received in the same session up to the message where the acknowledge function is called upon are acknowledged. If the message server fails while processing a set of acknowledgments, one or more messages in that group might be redelivered.

Note that this behavior models the JMS 1.0.2 specification rather than the JMS 1.1 specification

(Using `CLIENT_ACKNOWLEDGE` mode is similar to using transactions, except there is no guarantee that all acknowledgments will be processed together if a provider fails during processing.)

- `DUPS_OK_ACKNOWLEDGE`. This mode instructs the system to acknowledge messages in a lazy manner. Multiple messages can be redelivered after a provider failure.

Performance is impacted by acknowledgement mode for the following reasons:

- Extra control messages between broker and client are required in `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes. The additional control messages add processing overhead and can interfere with JMS payload messages, causing processing delays.
- In `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes, the client must wait until the broker confirms that it has processed the client's acknowledgment before the client can consume more messages. (This broker confirmation guarantees that the broker will not inadvertently redeliver these messages.)
- The Message Queue persistent store must be updated with the acknowledgement information for all persistent messages received by consumers, thereby decreasing performance.

Durable vs. Non-Durable Subscriptions

Subscribers to a topic destination have either durable or non-durable subscriptions. Durable subscriptions provide increased reliability at the cost of slower throughput for the following reasons:

- The Message Queue message server must persistently store the list of messages assigned to each durable subscription so that should a message server fail, the list is available after recovery.
- Persistent messages for durable subscriptions are stored persistently, so that should a message server fail, the messages can still be delivered after recovery, when the corresponding consumer becomes active. By contrast, persistent messages for non-durable subscriptions are not stored persistently (should a message server fail, the corresponding consumer connection is lost and the message would never be delivered).

For nonpersistent messages, performance is about the same for durable and non durable subscriptions. For persistent messages, performance is about 20% lower for durable subscriptions than for nondurable subscriptions.

Use of Selectors (Message Filtering)

Application developers can have the messaging provider sort messages according to criteria specified in the message selector associated with a consumer and deliver to that consumer only those messages whose property value matches the message selector. For example, if an application creates a subscriber to the topic `WidgetOrders` and specifies the expression `NumberOfOrders >1000` for the message selector, messages with a `NumberOfOrders` property value of 1001 or more are delivered to that subscriber.

Creating consumers with selectors lowers performance (as compared to using multiple destinations) because additional processing is required to handle each message. When a selector is used, it must be parsed so that it can be matched against future messages. Additionally, the message properties of each message must be retrieved and compared against the selector as each message is routed. However, using selectors provides more flexibility in a messaging application and may lower resource requirements at the expense of speed.

For more information on using selectors, see [“Using Selectors Efficiently” on page 58](#)

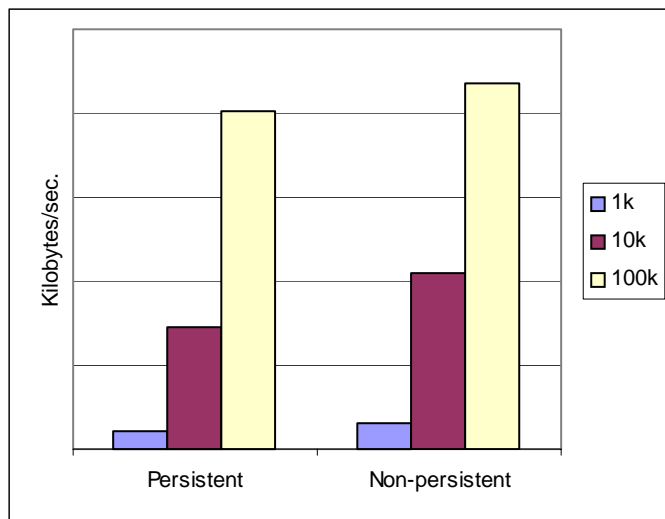
Message Size

Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

However, by batching smaller messages into a single message, the routing and processing of individual messages can be minimized, providing an overall performance gain. In this case, information about the state of individual messages is lost.

[Figure 3-1](#) compares throughput in kilobytes per second for 1k, 10k, and 100k-sized messages for persistent and non-persistent messages. All messages are sent to a queue destination and use `AUTO_ACKNOWLEDGE` acknowledgement mode.

[Figure 3-1](#) shows that in both cases there is less overhead in delivering larger messages compared to smaller messages. You can also see that the almost 50% performance gain of non-persistent messages over persistent messages shown for 1k and 10k-sized messages is not maintained for 100k-sized messages, probably because network bandwidth has become the bottleneck in message throughput for that case.

Figure 3-1 Performance Impact of a Message Size

Message Body Type

The C API supports two message body types:

- `BytesMessage`: Contains a set of bytes in a format determined by the application
- `TextMessage`: Is a simple `MQString`

Since performance varies with the complexity of the data, text messages are slightly more expensive to send than byte messages.

Reference

This chapter provides reference documentation for the Message Queue C-API. It includes information about the following:

- [“Data Types” on page 74](#) describes the C declarations for data types used by Message Queue messaging
- [“Function Reference” on page 84](#) describes the C functions that implement Message Queue messaging
- [“Header Files” on page 183](#) describes the contents of the C-API header files

For information on building C-Message Queue programs, see [Chapter 3, “Client Design Issues” on page 57](#).

For information on how you use the C-API to complete specific programming tasks, see [Chapter 2, “Using the C API” on page 25](#).

Data Types

[Table 4-1](#) summarizes the data types defined by the Message Queue C-API. The table lists data types in alphabetical order and provides cross references for types that require broader discussion.

Note that Message Queue data types designated as *handles* map to opaque structures (objects). Please do not attempt to dereference these handles to get to the underlying objects. Instead, use the functions provided to access the referenced objects.

Table 4-1 Message Queue C-API Data Type Summary

MQType	Description
ConstMQString	A constant MQString.
MQAckMode	An enum used to specify the acknowledgement mode of a session. Possible values include the following: MQ_AUTO_ACKNOWLEDGE MQ_CLIENT_ACKNOWLEDGE MQ_DUPS_OK_ACKNOWLEDGE MQ_SESSION_TRANSACTED. See “Acknowledge Modes” on page 80 for more information.
MQBool	A boolean that can assume one of two values: MQ_TRUE(=1) MQ_FALSE(=0).
MQChar	char type.
MQConnectionHandle	A handle used to reference a Message Queue connection. You get this handle when you call the MQCreateConnection function.
MQConsumerHandle	A handle used to reference a Message Queue consumer. A consumer can be durable, nondurable and synchronous, or asynchronous. You get this handle when you call one of the functions used to create consumers. See “Receiving Messages” on page 49 for more information.
MQDeliveryMode	An enum used to specify whether a message is sent persistently: MQ_NON_PERSISTENT_DELIVERY MQ_PERSISTENT_DELIVERY. You specify this value with the MQSendMessageExt function or the MQSendMessageToDestinationExt function.
MQDestinationHandle	A handle used to reference a Message Queue destination. You get this handle when you call the MQCreateDestination function or the MQCreateTemporaryDestination function.

Table 4-1 Message Queue C-API Data Type Summary (*Continued*)

MQType	Description
MQDestinationType	An enum used to specify the type of a destination: MQ_QUEUE_DESTINATION MQ_TOPIC_DESTINATION. You set the destination type using the MQCreateDestination function or the MQCreateTemporaryDestination function.
MQError	A 32-bit unsigned integer.
MQConnectionExceptionHandlerFunc	The type of a callback function used for connection exception handling. For more information, see “Callback Type for Connection Exception Handling” on page 83.
MQFloat32	A 32-bit floating-point number.
MQFloat64	A 64-bit floating-point number.
MQInt16	A 16-bit signed integer.
MQInt32	A 32-bit signed integer.
MQInt64	A 64-bit signed integer.
MQInt8	An 8-bit signed integer.
MQMessageHandle	A handle used to reference a Message Queue message. You get this handle when you call the MQCreateBytesMessage function, or the MQCreateTextMessage function, or on receipt of a message.
MQMessageListenerFunc	The type of a callback function used for asynchronous message receipt. For more information, see “Callback Type for Asynchronous Messaging” on page 81.
MQMessageType	An enum passed back by the MQGetMessageType and used to specify the type of a message; possible values include the following: MQ_TEXT_MESSAGE MQ_BYTES_MESSAGE MQ_UNSUPPORTED_MESSAGE.
MQProducerHandle	A handle used to reference a Message Queue producer. You get this handle when you call MQCreateMessageProducer or MQCreateMessageProducerForDestination .
MQPropertiesHandle	A handle used to reference Message Queue properties. You use this handle to define or read connection properties and message headers or message properties. See “Working With Properties” on page 28 for more information.

Table 4-1 Message Queue C-API Data Type Summary (*Continued*)

MQType	Description
MQReceiveMode	An enum used to specify whether consumers are synchronous or asynchronous. It can be one of the following: MQ_SESSION_SYNC_RECEIVE MQ_SESSION_ASYNC_RECEIVE. See MQCreateSession for more information.
MQSessionHandle	A handle used to reference a Message Queue session. You get this handle when you call the MQCreateSession function.
MQStatus	A data type returned by nearly all functions defined in <code>mqcrt.h</code> . See “Error Handling” on page 53 for more information on how you handle errors returned by Message Queue functions.
MQString	A null terminated UTF-8 encoded character string
MQType	An enum used to return the type of a single property; possible values include the following: MQ_BOOL_TYPE MQ_INT8_TYPE MQ_INT16_TYPE MQ_INT32_TYPE MQ_INT64_TYPE MQ_FLOAT32_TYPE MQ_FLOAT64_TYPE MQ_STRING_TYPE MQ_INVALID_TYPE

Connection Properties

When you create a connection using the [MQCreateConnection](#) function, you must pass a handle to an object of type `MQPropertiesHandle`. To set the properties referenced by this handle, you do the following:

1. Call the [MQCreateProperties](#) function to get a handle to a newly created properties object
2. Call a function to set one of the connection properties listed in [Table 4-2](#).

Which function you call depends on the type of the property you want to set; for example, to set an `MQString` property, you call the [MQSetStringProperty](#) function; to set a `MQBool` property, you call the [MQSetBoolProperty](#) function; and so on. Each function that sets a property requires that you pass a key name (constant) and value; these are listed and described in [Table 4-2](#).

- When you have set all the properties you want to define for the connection, you can then create the connection, by calling the `MQCreateConnection` function.

The runtime library sets the connection properties that specify the name and version of the Message Queue product; you can retrieve these using the `MQGetMetaData` function. These properties are described at the end of [Table 4-2](#), starting with `MQ_NAME_PROPERTY`.

Table 4-2 Connection Properties

Key Name	Description
<code>MQ_CONNECTION_TYPE_PROPERTY</code>	<p>An <code>MQString</code> specifying the transport protocol of the connection service used by the client. Supported types are TCP or SSL.</p> <p>Default: TCP</p>
<code>MQ_ACK_TIMEOUT_PROPERTY</code>	<p>A 32-bit integer specifying the maximum time in milliseconds that the client runtime will wait for any broker acknowledgement before returning an <code>MQ_TIMEOUT_EXPIRED</code> error. A value of 0 means there is no time-out.</p> <p>Default: 0</p>
<code>MQ_BROKER_HOST_PROPERTY</code>	<p>An <code>MQString</code> specifying the broker host name to which to connect.</p> <p>If you set the property <code>MQ_SSL_BROKER_IS_TRUSTED</code> to <code>false</code>, the value you specify for the property <code>MQ_BROKER_HOST_PROPERTY</code> must match the CN (common name) of the broker's certificate.</p> <p>No default.</p>
<code>MQ_PING_INTERVAL_PROPERTY</code>	<p>A 32-bit integer specifying the interval (in seconds) that the connection can remain idle before the client runtime tests the connection by pinging the broker. (The exact amount of time it takes for the ping to detect connection failure varies with the system's TCP configuration.)</p> <p>A ping interval that is ≤ 0 turns off the ping for the connection. The minimum allowable interval is 1 second. This prevents an application from setting the interval to a value that would affect performance.</p> <p>The ping interval is logged at the INFO level by the C client runtime when a connection is created.</p> <p>Default: 30 seconds</p>
<code>MQ_BROKER_PORT_PROPERTY</code>	<p>A 32-bit integer specifying the broker's primary port number.</p> <p>No default.</p>

Table 4-2 Connection Properties (*Continued*)

Key Name	Description
MQ_ACK_ON_PRODUCE_PROPERTY	<p>An <code>MQBool</code> specifying whether the producing client waits for broker acknowledgement of receipt of message from the producing client.</p> <p>If set to <code>MQ_TRUE</code>, the broker acknowledges receipt of all messages (persistent and non-persistent) from the producing client, and the producing client thread will block waiting for those acknowledgements.</p> <p>If set to <code>MQ_FALSE</code>, broker does not acknowledge receipt of any message (persistent or non-persistent) from the producing client, and the producing client thread will not block waiting for broker acknowledgements.</p> <p>Default: the broker acknowledges receipt of <i>persistent</i> messages only from the producing client, and the producing client thread will block waiting for those acknowledgements.</p>
MQ_ACK_ON_ACKNOWLEDGE_PROPERTY	<p>An <code>MQBool</code> specifying whether the broker confirms (acknowledges) consumer acknowledgements. A consumer acknowledgement can be initiated either by the client's session or by the consuming client, depending on the session acknowledgement mode (see Table 4-3). If the session's acknowledgement mode is <code>MQ_DUPS_OK_ACKNOWLEDGE</code>, this flag has no effect.</p> <p>If set to <code>MQ_TRUE</code>, the broker acknowledges all consuming acknowledgements, and the consuming client thread blocks waiting for these broker acknowledgements.</p> <p>If set to <code>MQ_FALSE</code>, the broker does not acknowledge any consuming client acknowledgements, and the consuming client thread will not block waiting for such broker acknowledgements.</p> <p>Default: <code>MQ_TRUE</code></p> <p>For more information, see the discussion for the MQAcknowledgeMessages function and "Message Acknowledgement" on page 40.</p>
MQ_CONNECTION_FLOW_COUNT_PROPERTY	<p>A 32-bit integer, greater than 0, specifying the number of Message Queue messages in a metered batch. When this number of messages is delivered from the broker to the client runtime, delivery is temporarily suspended, allowing any control messages that had been held up to be delivered. Payload message delivery is resumed upon notification by the client runtime, and continues until the count is again reached.</p> <p>Default: 100</p>

Table 4-2 Connection Properties (*Continued*)

Key Name	Description
MQ_CONNECTION_FLOW_LIMIT_ENBABLED_PROPERTY	<p>An <code>MQBool</code> specifying whether the value <code>MQ_CONNECTION_FLOW_LIMIT_PROPERTY</code> is used to control message flow. Specify <code>MQ_TRUE</code> to use the value and <code>MQ_FALSE</code> otherwise.</p> <p>Default: <code>MQ_FALSE</code></p>
MQ_CONNECTION_FLOW_LIMIT_PROPERTY	<p>A 32-bit integer, greater than 0, specifying the maximum number of unconsumed messages the client runtime can hold for each connection. Note however, that unless <code>MQ_CONNECTION_FLOW_LIMIT_ENBABLED_PROPERTY</code> is <code>MQ_TRUE</code>, this limit is not checked.</p> <p>When the number of unconsumed messages held by the client runtime for the connection exceeds the limit, message delivery stops. It is resumed (in accordance with the flow metering governed by <code>MQ_CONNECTION_FLOW_COUNT_PROPERTY</code>) only when the number of unconsumed messages drops below the value set with this property.</p> <p>This limit prevents a consuming client that is taking a long time to process messages from being overwhelmed with pending messages that might cause it to run out of memory.</p> <p>Default: 1000</p>
MQ_SSL_BROKER_IS_TRUSTED	<p>An <code>MQ_Bool</code> specifying whether the broker is trusted.</p> <p>Default: <code>MQ_TRUE</code></p>
MQ_SSL_CHECK_BROKER_FINGERPRINT	<p>An <code>MQ_Bool</code>. If it is set to <code>MQ_TRUE</code> and if <code>MQ_SSL_BROKER_IS_TRUSTED</code> is <code>MQ_FALSE</code>, the broker's certificate fingerprint is compared with the <code>MQ_SSL_BROKER_CERT_FINGERPRINT</code> property value in case of certificate authorization failure. If they match, the broker's certificate is authorized for use in the SSL connection.</p> <p>Default: <code>MQ_FALSE</code></p>
MQ_SSL_BROKER_CERT_FINGERPRINT	<p>An <code>MQString</code> specifying the MD5 hash, in hex format, of the broker's certificate.</p> <p>Default: <code>NULL</code></p>
MQ_NAME_PROPERTY	<p>An <code>MQString</code> that specifies the name of the Message Queue product. This property is set by the runtime library. See the MQGetMetaData function for more information.</p>
MQ_VERSION_PROPERTY	<p>An <code>MQInt32</code> that specifies the version of the Message Queue product. This property is set by the runtime library. See the MQGetMetaData function for more information.</p>

Table 4-2 Connection Properties (*Continued*)

Key Name	Description
MQ_MAJOR_VERSION_PROPERTY	<p>An MQInt32 that specifies the major version of the Message Queue product. For example, if the version is 3.5.0.1, the major version would be 3.</p> <p>This property is set by the runtime library. See the MQGetMetaData function for more information.</p>
MQ_MINOR_VERSION_PROPERTY	<p>An MQInt32 that specifies the minor version of the Message Queue product. For example, if the version is 3.5.0.1, the minor version would be 5.</p> <p>This property is set by the runtime library. See the MQGetMetaData function for more information.</p>
MQ_MICRO_VERSION_PROPERTY	<p>An MQInt32 that specifies the micro version of the Message Queue product. For example, if the version is 3.5.0.1, the micro version would be 0.</p> <p>This property is set by the runtime library. See the MQGetMetaData function for more information.</p>
MQ_SERVICE_PACK_PROPERTY	<p>An MQInt32 that specifies the service pack version of the Message Queue product. For example, if the version is 3.5.0.1, the service pack version would be 1.</p> <p>This property is set by the runtime library. See the MQGetMetaData function for more information.</p>

Acknowledge Modes

The Message Queue runtime supports reliable delivery by using transacted sessions or through acknowledgement options set at the session level. When you use the [MQCreateSession](#) function to create a session, you must specify an acknowledgement option for that session using the `acknowledgeMode` parameter. The value of this parameter is ignored for transacted sessions.

[Table 4-3](#) describes the effect of the options you can set using the `acknowledgeMode` parameter.

Table 4-3 acknowledgeMode Values

Enum	Description
MQ_AUTO_ACKNOWLEDGE	The session automatically acknowledges each message consumed by the client. This happens when one of the receive functions returns successfully, or when the message listener processing the message returns successfully.
MQ_CLIENT_ACKNOWLEDGE	The client explicitly acknowledges all messages for the session that have been consumed up to the point when the <code>MQAcknowledgeMessages</code> function has been called. See the discussion of the function <code>MQAcknowledgeMessages</code> for additional information.
MQ_DUPS_OK_ACKNOWLEDGE	The session acknowledges after ten messages have been consumed and does not guarantee that messages are delivered and consumed only once.
MQ_SESSION_TRANSACTED	This value is read only. It is set by the library if you have passed <code>MQ_TRUE</code> for the <code>isTransacted</code> parameter to the <code>MQCreateSession</code> function. It is returned to you by the <code>MQGetAcknowledgeMode</code> function if the session is transacted.

Callback Type for Asynchronous Messaging

When you call the `MQCreateAsyncMessageConsumer` function or the `MQCreateAsyncDurableMessageConsumer` function, you must pass the name of an `MQMessageListenerFunc` type callback function that is to be called when the consumer receives a message to the specified destination.

The `MQMessageListenerFunc` type has the following definition:

```
MQError (* MQMessageListenerFunc)(
    const MQSessionHandle sessionHandle,
    const MQConsumerHandle consumerHandle,
    MQMessageHandle messageHandle
    void * callbackData);
```

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. The client runtime specifies this handle when it calls your message listener.
<code>consumerHandle</code>	A handle to the consumer receiving the message. The client runtime specifies this handle when it calls your message listener.
<code>messageHandle</code>	A handle to the incoming message. The client runtime specifies this handle when it calls your message listener.
<code>callbackData</code>	The void pointer that you passed to the function MQCreateAsyncMessageConsumer or the function MQCreateAsyncDurableMessageConsumer .

The body of a message listener function is written by the receiving client. Mainly, the function needs to process the incoming message by examining its header, body, and properties. The client is also responsible for freeing the message handle (either from within the handler or from outside the handler) by calling [MQFreeMessage](#).

In addition, you should observe the following guidelines when writing the message listener function:

- If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge messages that you have received. For more information, see the description of the function [MQAcknowledgeMessages](#).
- Do not try to close the session (or the connection to which it belongs) and consumer handle in the message listener.
- It is possible for a message listener to return an error; however, this is considered a client programming error. If the listener discovers that the message is badly formatted or if it cannot process it for some other reason, it should handle the problem itself by re-directing it to an application-specific bad-message destination and process it later.

If the message listener does return an error, the client runtime will try to redeliver the message once if the session's acknowledge mode is either `MQ_AUTO_ACKNOWLEDGE` or `MQ_DUPS_OK_ACKNOWLEDGE`.

Callback Type for Connection Exception Handling

The client runtime will call this function when a connection exception occurs.

The `MQConnectionExceptionHandlerFunc` type has the following definition:

```
Void (* MQConnectionExceptionHandlerFunc)(
    const MQConnectionHandle connectionHandle,
    MQStatus exception,
    void * callbackData);
```

Parameters

<code>connectionHandle</code>	The handle to the connection on which the connection exception occurred. The client runtime sets this handle when it calls the connection exception handler.
<code>exception</code>	An <code>MQStatus</code> for the connection exception that occurred. The client runtime specifies this value when it calls the exception handler. You can pass this status result to any functions used to handle errors to get an error code or error string. For more information, see “Error Handling” on page 53 .
<code>callbackData</code>	Whatever void pointer was passed as the <code>listenerCallbackData</code> parameter to the <code>MQCreateConnection</code> for more information.

The body of a connection exception listener function is written by the client. This function will only be called synchronously with respect to a single connection. If you install it as the connection exception listener for multiple connections, then it must be reentrant.

Do not try to close the session (or the connection to which it belongs) in the exception listener.

Function Reference

This section describes the C-API functions in alphabetical order. [Table 4-4](#) lists the C-API functions.

Table 4-4 Message Queue C-API Function Summary

Function	Description
MQAcknowledgeMessages	Acknowledges the specified message and all messages received before it on the same session.
MQCloseConnection	Closes the specified connection.
MQCloseMessageConsumer	Closes the specified consumer.
MQCloseMessageProducer	Closes the specified message producer without closing its connection.
MQCloseSession	Closes the specified session.
MQCommitSession	Commits a transaction associated with the specified session.
MQCreateAsyncDurableMessageConsumer	Creates a durable asynchronous message consumer for the specified destination.
MQCreateAsyncMessageConsumer	Creates an asynchronous message consumer for the specified destination.
MQCreateBytesMessage	Creates an <code>MQ_BYTES_MESSAGE</code> message.
MQCreateConnection	Creates a connection to the broker.
MQCreateDestination	Creates a logical destination and passes a handle to it back to you.
MQCreateDurableMessageConsumer	Creates a durable synchronous message consumer for the specified destination.
MQCreateMessageConsumer	Creates a synchronous message consumer for the specified destination.
MQCreateMessageProducer	Creates a message producer with no default destination.
MQCreateMessageProducerForDestination	Creates a message producer with a default destination.
MQCreateProperties	Creates a properties handle.
MQCreateSession	Creates a session and passes back a handle to the session.
MQCreateTemporaryDestination	Creates a temporary destination and passes its handle back to you.
MQCreateTextMessage	Creates a text message.
MQFreeConnection	Releases memory assigned to the specified connection and to all resources associated with that connection.

Table 4-4 Message Queue C-API Function Summary (*Continued*)

Function	Description
MQFreeDestination	Releases memory assigned to the specified destination and to all resources associated with that destination.
MQFreeMessage	Releases memory assigned to the specified message.
MQFreeProperties	Releases the memory allocated to the referenced properties handle.
MQFreeString	Releases the memory allocated to the specified <code>MQString</code> .
MQGetAcknowledgeMode	Passes back the acknowledgement mode of the specified session.
MQGetBoolProperty	Passes back a property of type <code>MQBool</code> .
MQGetBytesMessageBytes	Passes back the address and size of a <code>MQ_BYTES_MESSAGE</code> message body.
MQGetDestinationType	Passes back the type of the specified destination.
MQGetErrorTrace	Returns a string describing the stack at the time the specified error occurred.
MQGetFloat32Property	Passes back the value of the <code>MQFloat32</code> property for the specified key.
MQGetFloat64Property	Passes back the value of the <code>MQFloat64</code> property for the specified key.
MQGetInt16Property	Passes back the value of the <code>MQInt16</code> property for the specified key.
MQGetInt32Property	Passes back the value of the <code>MQInt32</code> property for the specified key.
MQGetInt64Property	Passes back the value of the <code>MQInt64</code> property for the specified key.
MQGetInt8Property	Passes back the value of the <code>MQInt8</code> property for the specified key.
MQGetMessageHeaders	Passes back a handle to the header of the specified message.
MQGetMessageProperties	Passes back a handle to the properties for the specified message.
MQGetMessageReplyTo	Passes back the destination where replies to this message should be sent.
MQGetMessageType	Passes back the type of the specified message.
MQGetMetaData	Passes back Message Queue version information.
MQGetPropertyType	Passes back the type of the specified property key.
MQGetStatusCode	Returns the code for the specified <code>MQStatus</code> result.
MQGetStatusString	Returns a string description for the specified <code>MQStatus</code> result.

Table 4-4 Message Queue C-API Function Summary (*Continued*)

Function	Description
MQGetStringProperty	Passes back the value for the specified property. <i>Type</i> (in the function name) can be <code>String</code> , <code>Bool</code> , <code>Int8</code> , <code>Int16</code> , <code>Int32</code> , <code>Int64</code> , <code>Float32</code> , <code>Float64</code> .
MQGetTextMessageText	Passes back the contents of an <code>MQ_TEXT_MESSAGE</code> message.
MQInitializeSSL	Initializes the SSL library. You must call this function before you create a connection that uses SSL.
MQPropertiesKeyIterationGetNext	Passes back the next property key in the properties handle.
MQPropertiesKeyIterationHasNext	Returns true if there is another property key in a properties object.
MQPropertiesKeyIterationStart	Starts iterating through a properties object.
MQReceiveMessageNoWait	Passes back a handle to a message delivered to the specified consumer.
MQReceiveMessageWait	Passes back a handle to a message delivered to the specified consumer when the message becomes available.
MQReceiveMessageWithTimeout	Passes back a handle to a message delivered to the specified consumer if a message is available within the specified amount of time.
MQRecoverSession	Stops message delivery and restarts message delivery with the oldest unacknowledged message.
MQRollBackSession	Rolls back a transaction associated with the specified session.
MQSendMessage	Sends a message for the specified producer.
MQSendMessageExt	Sends a message for the specified producer and allows you to set priority, time-to-live, and delivery mode.
MQSendMessageToDestination	Sends a message to the specified destination.
MQSendMessageToDestinationExt	Sends a message to the specified destination and allows you to set message header properties.
MQSetBoolProperty	Sets an <code>MQBool</code> property with the specified key to the specified value.
MQSetBytesMessageBytes	Sets the message body for the specified <code>MQ_BYTES_MESSAGE</code> message.
MQSetFloat32Property	Sets an <code>MQFloat 32</code> property with the specified key to the specified value.
MQSetFloat64Property	Sets an <code>MQFloat 64</code> property with the specified key to the specified value.
MQSetInt16Property	Sets an <code>MQInt16</code> property with the specified key to the specified value.

Table 4-4 Message Queue C-API Function Summary (*Continued*)

Function	Description
MQSetInt32Property	Sets an <code>MQInt32</code> property with the specified key to the specified value.
MQSetInt64Property	Sets an <code>MQInt64</code> property with the specified key to the specified value.
MQSetInt8Property	Sets an <code>MQInt8</code> property with the specified key to the specified value.
MQSetMessageHeaders	Sets the header part of the message.
MQSetMessageProperties	Sets the user-defined properties for the specified message.
MQSetMessageReplyTo	Specifies the destination where replies to this message should be sent.
MQSetStringProperty	Sets an <code>MQString</code> property with the specified key to the specified value.
MQSetStringProperty	Sets the message body for the specified <code>MQ_TEXT_MESSAGE</code> message.
MQSetTextMessageText	Defines the body for a text message.
MQStartConnection	Starts the specified connection to the broker and starts or resumes message delivery.
MQStatusIsError	Returns <code>MQ_TRUE</code> if the specified <code>MQStatus</code> result is an error.
MQStopConnection	Stops the specified connection to the broker. This stops the broker from delivering messages.
MQUnsubscribeDurableMessageConsumer	Unsubscribes the specified durable message consumer.

MQAcknowledgeMessages

The `MQAcknowledgeMessages` function acknowledges the specified message and all messages received before it on the same session. This function is valid only if the session is created with acknowledge mode set to `MQ_CLIENT_ACKNOWLEDGE`.

Return Value

```
MQAcknowledgeMessages      (const MQSessionHandle sessionHandle,
                           const MQMessageHandle messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session for the consumer that received the specified message.
<code>messageHandle</code>	A handle to the message that you want to acknowledge. This handle is passed back to you when you receive the message (either by calling one of the receive functions or when a message is delivered to your message listener function.)

Whether you receive messages synchronously or asynchronously, you can call the `MQAcknowledgeMessages` function to acknowledge receipt of the specified message and of all messages that preceded it.

When you create a session you specify one of several acknowledge modes for that session; these are described in [Table 4-3](#). If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge receipt of messages consumed in that session.

By default, the calling thread to the `MQAcknowledgeMessages` function will be blocked until the broker acknowledges receipt of the acknowledgment for the broker consumed. If, when you created the session's connection, you specified the property `MQ_ACK_ON_ACKNOWLEDGE_PROPERTY` to be `MQ_FALSE`, the calling thread will not wait for the broker to acknowledge the acknowledgement.

Common Errors

`MQ_SESSION_NOT_CLIENT_ACK_MODE`

`MQ_MESSAGE_NOT_IN_SESSION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CLOSED`

MQCloseConnection

The `MQCloseConnection` function closes the connection to the broker.

```
MQCloseConnection          (MQConnectionHandle connectionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`connectionHandle` The handle to the connection that you want to close. This handle is created and passed back to you by the [MQCreateConnection](#) function.

Closing the connection closes all sessions, producers, and consumers created from this connection. This also forces all threads associated with this connection that are blocking in the library to return.

Closing the connection does not actually release all the memory associated with the connection. After all the application threads associated with this connection (and its dependent sessions, producers, and consumers) have returned, you should call the [MQFreeConnection](#) function to release these resources.

Common Errors

`MQ_CONCURRENT_DEADLOCK`

(If the function is called from an exception listener or a consumer's message listener.)

MQCloseMessageConsumer

The `MQCloseMessageConsumer` function closes the specified message consumer.

```
MQCloseMessageConsumer      (MQConsumerHandle consumerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>consumerHandle</code>	The handle to the consumer you want to close. This handle is created and passed back to you by one of the functions used to create consumers. This handle is invalid after the function returns.
-----------------------------	---

A session's consumers are automatically closed when you close the session or connection to which they belong. To close a consumer without closing the session or connection to which it belongs, use the [MQCloseMessageConsumer](#) function.

If the consumer you want to close is a durable consumer and you want to close this consumer permanently, you should call the function [MQUnsubscribeDurableMessageConsumer](#) after closing the consumer in order to delete any state information maintained by the broker for this consumer.

Common Errors

`MQ_CONSUMER_NOT_IN_SESSION`

`MQ_BROKER_CONNECTION_CLOSED`

MQCloseMessageProducer

The `MQCloseMessageProducer` function closes a message producer.

```
MQCloseMessageProducer      (MQProducerHandle producerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`producerHandle` A handle for this producer that was passed to you by the [MQCreateMessageProducer](#) function or by the [MQCreateMessageProducerForDestination](#) function. This handle is invalid after the function returns.

Use the `MQCloseMessageProducer` function to close a producer without closing its associated session or connection.

Common Errors

`MQ_PRODUCER_NOT_IN_SESSION`

MQCloseSession

The `MQCloseSession` function closes the specified session.

```
MQCloseSession      (MQSessionHandle sessionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session that you want to close. This handle is created and passed back to you by the MQCreateSession function. This handle is invalid after the function returns.
----------------------------	---

Closing a session closes the resources (producers and consumers) associated with that session and frees up the memory allocated for that session.

There is no need to close the producers or consumers of a closed session.

Common Errors

`MQ_CONCURRENT_DEADLOCK`

(If called from a consumer's message listener in the session.)

MQCommitSession

The `MQCommitSession` function commits a transaction associated with the specified session.

```
MQCommitSession      (const MQSessionHandle sessionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`sessionHandle` The handle to the transacted session that you want to commit.

A transacted session supports a series of transactions. Transactions organize a session's input message stream and output message stream into a series of atomic units. A transaction's input and output units consist of those messages that have been produced and consumed within the session's current transaction. (Note that the receipt of a message cannot be part of the same transaction that produces the message.) When you call the `MQCommitSession` function, its atomic unit of input is acknowledged and its associated atomic unit of output is sent.

The completion of a session's current transaction automatically begins the next transaction. The result is that a transacted session always has a current transaction within which its work is done. Use the [MQRollBackSession](#) function to roll back a transaction.

Common Errors

`MQ_NOT_TRANSACTED_SESSION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQCreateAsyncDurableMessageConsumer

The `MQCreateAsyncDurableMessageConsumer` function creates an asynchronous durable message consumer for the specified destination.

```
MQCreateAsyncDurableMessageConsumer (
    const MQSessionHandle  sessionHandle,
    const MQDestinationHandle  destinationHandle,
    ConstMQString durableName,
    ConstMQString  messageSelector,
    MQBool noLocal,
    MQMessageListenerFunc messageListener,
    void * listenerCallbackData,
    MQConsumerHandle *  consumerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. This handle is passed back by the MQCreateSession function. For this asynchronous durable consumer, the session must have been created with the <code>MQ_SESSION_ASYNC_RECEIVE</code> receive mode.
<code>destinationHandle</code>	A handle to a topic destination on which the consumer receives messages. This handle remains valid after the call.
<code>durableName</code>	An <code>MQString</code> specifying a name for the durable subscriber. The library makes a copy of the <code>durableName</code> string.
<code>messageSelector</code>	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a <code>NULL</code> or empty string to indicate that there is no message selector for this consumer. In this case, all messages are delivered. The library makes a copy of the <code>messageSelector</code> string. For more information about SQL, see <i>X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2</i> , ISBN 1-85912-151-9, March 1966.
<code>noLocal</code>	Specify <code>MQ_TRUE</code> to inhibit delivery of messages published by this consumer's own connection.

<code>messageListener</code>	The name of an <code>MQMessageListenerFunc</code> type callback function that is to be called when this consumer receives a message on the specified destination.
<code>listenerCallbackData</code>	A pointer to data that you want passed to your message listener function when it is called by the library.
<code>consumerHandle</code>	Output parameter for the handle that references the consumer for the specified destination.

In the case of an asynchronous consumer, you should not start a connection before calling the `MQCreateAsyncDurableMessageConsumer` function. (You should create a connection, create a session, set up your asynchronous consumer, create the consumer, and then start the connection.) Attempting to create a consumer when the connection is not stopped, will result in an `MQ_CONCURRENT_ACCESS` error.

The `MQCreateAsyncDurableMessageConsumer` function creates an asynchronous durable message consumer for the specified destination. You can define parameters to filter messages and to inhibit the delivery of messages you published to your own connection. Note that the session's receive mode (sync/async) must be appropriate for the kind of consumer you are creating (sync/async). To create a synchronous durable message consumer for a destination, call the function [MQCreateDurableMessageConsumer](#).

Durable consumers can only be used for topic destinations. If you are creating an asynchronous consumer for a queue destination or if you are not interested in messages that arrive to a topic while you are inactive, you might prefer to use the function [MQCreateAsyncMessageConsumer](#).

The broker retains a record of this durable subscription and makes sure that all messages from the publishers to this topic are retained until they are either acknowledged by this durable subscriber or until they have expired. Sessions with durable subscribers must always provide the same client identifier. (See `MQCreateConnection`, `clientID` parameter.) In addition, each durable consumer must specify a durable name using the `durableName` parameter, which uniquely identifies (for each client identifier) the durable subscription when it is created.

A session's consumers are automatically closed when you close the session or connection to which they belong. However, messages will be routed to the durable subscriber while it is inactive and delivered when the durable consumer is recreated. To close a consumer without closing the session or connection to which it belongs, use the [MQCloseMessageConsumer](#) function. If you want to close a durable consumer permanently, you should call the [MQUnsubscribeDurableMessageConsumer](#) after closing it to delete state information maintained by the Broker on behalf of the durable consumer.

Common Errors

MQ_NOT_ASYNC_RECEIVE_MODE

MQ_INVALID_MESSAGE_SELECTOR

MQ_DESTINATION_CONSUMER_LIMIT_EXCEEDED

MQ_TEMPORARY_DESTINATION_NOT_IN_CONNECTION

MQ_CONSUMER_NO_DURABLE_NAME

MQ_QUEUE_CONSUMER_CANNOT_BE_DURABLE

MQ_CONCURRENT_ACCESS

MQ_SESSION_CLOSED

MQ_BROKER_CONNECTION_CLOSED

MQCreateAsyncMessageConsumer

The `MQCreateAsyncMessageConsumer` function creates an asynchronous message consumer for the specified destination.

```
MQCreateAsyncMessageConsumer
```

```
(const MQSessionHandle sessionHandle,
 const MQDestinationHandle destinationHandle,
 ConstMQString messageSelector,
 MQBool noLocal,
 MQMessageListenerFunc messageListener,
 void * listenerCallBackData,
 MQConsumerHandle * consumerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. This handle is created and passed back to you by the MQCreateSession function. For this asynchronous consumer, the session must have been created with the <code>MQ_SESSION_ASYNC_RECEIVE</code> receive mode.
<code>destinationHandle</code>	A handle to the destination on which the consumer receives messages. This handle remains valid after the call returns.
<code>messageSelector</code>	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a <code>NULL</code> or empty string to indicate that there is no message selector for this consumer. In this case, all messages will be delivered. The library makes a copy of the <code>messageSelector</code> string. For more information about SQL, see <i>X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2</i> , ISBN 1-85912-151-9, March 1966.
<code>noLocal</code>	Specify <code>MQ_TRUE</code> to inhibit delivery of messages published by this consumer's own connection. The setting of this parameter applies only to topic destinations. It is ignored for queues.

<code>messageListener</code>	The name of an <code>MQMessageListenerFunc</code> type callback function that is to be called when this consumer receives a message for the specified destination.
<code>listenerCallbackData</code>	A pointer to data that you want passed to your message listener function when it is called by the library.
<code>consumerHandle</code>	Output parameter for the handle that references the consumer for the specified destination.

In the case of an asynchronous consumer, you should not start a connection before calling the `MQCreateAsyncDurableMessageConsumer` function. (You should create a connection, create a session, set up your asynchronous consumers, create the consumer, and then start the connection.) Attempting to create a consumer when the connection is not stopped will result in an `MQ_CONCURRENT_ACCESS` error.

The `MQCreateAsyncMessageConsumer` function creates an asynchronous message consumer for the specified destination. You can define parameters to filter messages and to inhibit the delivery of messages you published to your own connection. Note that the session's receive mode (sync/async) must be appropriate for the kind of consumer you are creating (sync/async). To create a synchronous message consumer for a destination, use the [MQCreateMessageConsumer](#) function.

If this consumer is on a topic destination, it will only receive messages produced while the consumer is active. If you are interested in receiving messages published while this consumer is not active, you should create a consumer using the [MQCreateAsyncDurableMessageConsumer](#) function instead.

A session's consumers are automatically closed when you close the session or connection to which they belong. To close a consumer without closing the session or connection to which it belongs, use the [MQCloseMessageConsumer](#) function.

Common Errors

`MQ_NOT_ASYNC_RECEIVE_MODE`

`MQ_INVALID_MESSAGE_SELECTOR`

`MQ_DESTINATION_CONSUMER_LIMIT_EXCEEDED`

`MQ_TEMPORARY_DESTINATION_NOT_IN_CONNECTION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQCreateBytesMessage

The `MQCreatesBytesMessage` function creates a bytes message and passes a handle to it back to you.

```
MQCreateBytesMessage      (MQMessageHandle * messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`messageHandle` Output parameter for the handle to the new, empty message.

After you obtain the handle to a bytes message, you can use this handle to define its content with the [MQSetBytesMessageBytes](#) function, to set its headers with the [MQSetMessageHeaders](#) function, and to set its properties with the [MQSetMessageProperties](#) function.

MQCreateConnection

The `MQCreateConnection` function creates a connection to the broker.

If you want to connect to the broker over SSL, you must call the `MQInitializeSSL` function to initialize the SSL library before you create the connection.

`MQCreateConnection`

```
(MQPropertiesHandle propertiesHandle
  ConstMQString username,
  ConstMQString password,
  ConstMQString clientID,
  MQConnectionExceptionHandlerFunc exceptionListener,
  void * listenerCallbackData,
  MQConnectionHandle * connectionHandle);
```

Return Value

`MQStatus`. See the `MQStatusIsError` function for more information.

Parameters

<code>propertiesHandle</code>	<p>A handle that specifies the properties that determine the behavior of this connection. You must create this handle using the <code>MQCreateProperties</code> function before you try to create a connection. This handle will be invalid after the function returns.</p> <p>See Table 4-2 on page 77 for information about connection properties.</p>
<code>username</code>	<p>An <code>MQString</code> specifying the user name to use when connecting to the broker.</p> <p>The library makes a copy of the <code>username</code> string.</p>
<code>password</code>	<p>An <code>MQString</code> specifying the password to use when connecting to the broker.</p> <p>The library makes a copy of the <code>password</code> string.</p>
<code>clientID</code>	<p>An <code>MQString</code> used to identify the connection. If you use the connection for a durable consumer, you must specify a non-NULL client identifier.</p> <p>The library makes a copy of the <code>clientID</code> string.</p>
<code>exceptionListener</code>	<p>A connection-exception callback function used to notify the user that a connection exception has occurred.</p>

<code>listenerCallBackData</code>	A data pointer that can be passed to the connection <code>exceptionListener</code> callback function whenever it is called. The user can set this pointer to any data that may be useful to pass along to the connection exception listener for this connection. Set this to <code>NULL</code> if you do not need to pass data back to the connection exception listener.
<code>connectionHandle</code>	Output parameter for the handle to the connection that is created by this function.

The `MQCreateConnection` function creates a connection to the broker. The behavior of the connection is specified by key values defined in the properties referenced by the `propertiesHandle` parameter. You must use the `MQCreateProperties` function to define these properties.

You cannot change the properties of a connection you have already created. If you need different connection properties, you must close and free the old connection and then create a new connection with the desired properties.

- Use the [MQStartConnection](#) function to start or restart the connection. Use the [MQStopConnection](#) function to stop a connection.
- Use the [MQGetMetaData](#) function to get information about the name of the Message Queue product and its version.
- Use the [MQCloseConnection](#) function to close a connection, and then use the [MQFreeConnection](#) function to free the memory allocated for that connection.

Setting a Client Identifier

To keep track of durable subscriptions, Message Queue uses a unique *client identifier* that associates a client's connection with state information maintained by the message service on behalf of the client. By definition, a client identifier is unique, and applies to only one connection at a time.

The messaging service uses a client identifier in combination with a durable subscription name to uniquely identify each durable subscription. If a durable subscriber is inactive at the time that messages are delivered to a topic destination, the broker retains messages for that subscriber and delivers them when the subscriber once again becomes active.

Handling Connection Exceptions

Use the `exceptionListener` parameter to pass the name of a user-defined callback function that can be called synchronously when a connection exception occurs for this connection. Use the `exceptionCallBackData` parameter to specify any user data that you want to pass to the callback function.

Common Errors

`MQ_INCOMPATIBLE_LIBRARY`

`MQ_CONNECTION_UNSUPPORTED_TRANSPORT`

`MQ_COULD_NOT_CREATE_THREAD`

`MQ_INVALID_CLIENT_ID`

`MQ_CLIENT_ID_IN_USE`

`MQ_COULD_NOT_CONNECT_TO_BROKER`

`MQ_SSL_NOT_INITIALIZED`

This error can be returned if `MQ_CONNECTION_TYPE_PROPERTY` is SSL and you have not called the `MQInitializeSSL` function before creating this connection.

MQCreateDestination

The `MQCreateDestination` function creates a logical destination and passes a handle to it back to you.

```
MQCreateDestination    (const MQSessionHandle sessionHandle
                        ConstMQString destinationName,
                        MQDestinationType destinationType,
                        MQDestinationHandle * destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session with which you want to associate this destination.
<code>destinationName</code>	An <code>MQString</code> specifying the logical name of this destination. The library makes a copy of the <code>destinationName</code> string. See discussion below. Destination names starting with “mq” are reserved and should not be used by clients.
<code>destinationType</code>	An enum specifying the destination type, either <code>MQ_QUEUE_DESTINATION</code> or <code>MQ_TOPIC_DESTINATION</code> .
<code>destinationHandle</code>	Output parameter for the handle to the newly created destination. You can pass this handle to functions sending messages or to message producers or consumers.

The `MQCreateDestination` function creates a logical destination and passes a handle to it back to you. Note that the Message Queue administrator has to also create a physical destination on the broker, whose name and type is the same as the destination created here, in order for messaging to happen. For example, if you use this function to create a queue destination called `myMailQDest`, the administrator has to create a physical destination on the broker named `myMailQDest`.

If you are doing development, you can simplify this process by turning on the `imq.autocreate.topic` or `imq.autocreate.queue` properties for the broker. If you do this, the broker automatically creates a physical destination whenever a message consumer or message producer attempts to access a non-existent destination. The auto-created destination will have the same name as the logical destination name you specified using the `MQCreateDestination` function. By default, the broker has the properties `imq.autocreate.topic` and `imq.autocreate.queue` turned on.

Common Errors

`MQ_INVALID_DESTINATION_TYPE`

`MQ_SESSION_CLOSED`

MQCreateDurableMessageConsumer

The `MQCreateDurableMessageConsumer` function creates a synchronous durable message consumer for the specified topic destination.

```
MQCreateDurableMessageConsumer
```

```
(const MQSessionHandle sessionHandle,
 const MQDestinationHandle destinationHandle,
 ConstMQString durableName,
 ConstMQString messageSelector,
 MQBool noLocal
 MQConsumerHandle * consumerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. This handle is passed back to you by the MQCreateSession function. For this (synchronous) durable consumer, the session must have been created with the <code>MQ_SESSION_SYNC_RECEIVE</code> receive mode.
<code>destinationHandle</code>	A handle to a topic destination on which the consumer receives messages. This handle remains valid after the call returns.
<code>durableName</code>	An <code>MQString</code> specifying the name of the durable subscriber to the topic destination. The library makes a copy of the <code>durableName</code> string.
<code>messageSelector</code>	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a <code>NULL</code> or empty string to indicate that there is no message selector for this consumer. In this case, the consumer receives all messages. The library makes a copy of the <code>messageSelector</code> string. For more information about SQL, see X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2, ISBN 1-85912-151-9, March 1966.

<code>noLocal</code>	Specify <code>MQ_TRUE</code> to inhibit delivery of messages published by this consumer's own connection.
<code>consumerHandle</code>	Output parameter for the handle that references the consumer for the specified destination.

The `MQCreateDurableMessageConsumer` function creates a synchronous message consumer for the specified destination. A durable consumer receives all the messages published to a topic, including the ones published while the subscriber is inactive.

You can define parameters to filter messages and to inhibit the delivery of messages you published to your own connection. Note that the session's receive mode (sync/async) must be appropriate for the kind of consumer you are creating (sync/async). To create an asynchronous durable message consumer for a destination, call the function [MQCreateAsyncDurableMessageConsumer](#).

Durable consumers are for topic destinations. If you are creating a consumer for a queue destination or if you are not interested in messages that arrive to a topic while you are inactive, you should use the function [MQCreateMessageConsumer](#).

The broker retains a record of this durable subscription and makes sure that all messages from the publishers to this topic are retained until they are either acknowledged by this durable subscriber or until they have expired. Sessions with durable subscribers must always provide the same client identifier (see `MQCreateConnection`, `clientID` parameter). In addition, each durable consumer must specify a durable name using the `durableName` parameter, which uniquely identifies (for each client identifier) the durable subscription when it is created.

A session's consumers are automatically closed when you close the session or connection to which they belong. However, messages will be routed to the durable subscriber while it is inactive and delivered when the durable consumer is recreated. To close a consumer without closing the session or connection to which it belongs, use the [MQCloseMessageConsumer](#) function. If you want to close a durable consumer permanently, you should call the [MQUnsubscribeDurableMessageConsumer](#) function after closing it to delete state information maintained by the broker on behalf of the durable consumer.

Common Errors

MQ_NOT_SYNC_RECEIVE_MODE

MQ_INVALID_MESSAGE_SELECTOR

MQ_DESTINATION_CONSUMER_LIMIT_EXCEEDED

MQ_TEMPORARY_DESTINATION_NOT_IN_CONNECTION

MQ_CONSUMER_NO_DURABLE_NAME

MQ_QUEUE_CONSUMER_CANNOT_BE_DURABLE

MQ_CONCURRENT_ACCESS

MQ_SESSION_CLOSED

MQ_BROKER_CONNECTION_CLOSED

MQCreateMessageConsumer

The `MQCreateMessageConsumer` function creates a synchronous message consumer for the specified destination.

`MQCreateMessageConsumer`

```
(const MQSessionHandle sessionHandle,
 const MQDestinationHandle destinationHandle,
 ConstMQString messageSelector,
 MQBool noLocal
 MQConsumerHandle * consumerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. This handle is passed back to you by the MQCreateSession function. For this (synchronous) consumer, the session must have been created with the <code>MQ_SESSION_SYNC_RECEIVE</code> receive mode.
<code>destinationHandle</code>	A handle to the destination on which the consumer receives messages. This handle remains valid after the call returns.
<code>messageSelector</code>	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a <code>NULL</code> or empty string to indicate that there is no message selector for this consumer and that all messages should be returned. The library makes a copy of the <code>messageSelector</code> string. For more information about SQL, see <i>X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2</i> , ISBN 1-85912-151-9, March 1966.
<code>noLocal</code>	Specify <code>MQ_TRUE</code> to inhibit delivery of messages published by this consumer's own connection. This applies only to topic destinations; it is ignored for queues.
<code>consumerHandle</code>	Output parameter for the handle that references the consumer for the specified destination.

The `MQCreateMessageConsumer` function creates a synchronous message consumer for the specified destination. You can define parameters to filter messages and to inhibit the delivery of messages you published to your own connection. Note that the session's receive mode (sync/async) must be appropriate for the kind of consumer you are creating (sync/async). To create an asynchronous message consumer for a destination, use the `MQCreateAsyncMessageConsumer` function.

If the consumer is a topic destination, it can only receive messages that are published while it is active. To receive messages published while this consumer is not active, you should create a consumer using either the `MQCreateDurableMessageConsumer` function or the `MQCreateAsyncDurableMessageConsumer` function, depending on the receive mode you defined for the session.

A session's consumers are automatically closed when you close the session or connection to which they belong. To close a consumer without closing the session or connection to which it belongs, use the `MQCloseMessageConsumer` function.

Common Errors

`MQ_NOT_SYNC_RECEIVE_MODE`

`MQ_INVALID_MESSAGE_SELECTOR`

`MQ_DESTINATION_CONSUMER_LIMIT_EXCEEDED`

`MQ_TEMPORARY_DESTINATION_NOT_IN_CONNECTION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQCreateMessageProducer

The `MQCreateMessageProducer` function creates a message producer that does not have a specified destination.

```
MQCreateMessageProducer    (const MQSessionHandle sessionHandle,
                            MQProducerHandle * producerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this producer should belong.
<code>producerHandle</code>	Output parameter for the handle that references the producer.

The `MQCreateMessageProducer` function creates a message producer that does not have a specified destination. In this case, you will specify the destination when sending the message itself by using either the [MQSendMessageToDestination](#) function or the [MQSendMessageToDestinationExt](#) function.

Using the `MQCreateMessageProducer` function is appropriate when you want to use the same producer to send messages to a variety of destinations. If, on the other hand, you want to use one producer to send many messages to the same destination, you should use the [MQCreateMessageProducerForDestination](#) function instead.

A session's producers are automatically closed when you close the session or connection to which they belong. To close a producer without closing the session or connection to which it belongs, use the [MQCloseMessageProducer](#) function.

Common Errors

`MQ_SESSION_CLOSED`

MQCreateMessageProducerForDestination

The `MQCreateMessageProducerForDestination` function creates a message producer with a specified destination.

```
MQCreateMessageProducerForDestination
(
    const MQSessionHandle sessionHandle,
    const MQDestinationHandle destinationHandle,
    MQProducerHandle * producerHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this producer belongs.
<code>destinationHandle</code>	A handle to the destination where you want this producer to send all messages. This handle remains valid after the call returns.
<code>producerHandle</code>	Output parameter for the handle that references the producer.

The `MQCreateMessageProducerForDestination` function creates a message producer with a specified destination. All messages sent out by this producer will go to that destination. Use the [MQSendMessage](#) function or the [MQSendMessageExt](#) function to send messages for a producer with a specified destination.

Use the [MQCreateMessageProducer](#) function when you want to use one producer to send messages to a variety of destinations.

A session's producers are automatically closed when you close the session or connection to which they belong. To close a producer without closing the session or connection to which it belongs, use the [MQCloseMessageProducer](#) function.

Common Errors

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQCreateProperties

The `MQCreateProperties` function creates a properties handle and passes it back to the caller.

```
MQCreateProperties      (MQPropertiesHandle * propertiesHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	Output parameter for the handle that references the newly created properties object.
-------------------------------	--

Use the `MQCreateProperties` function to get a properties handle. You can then use the appropriate `MQSet...Property` function to set the desired properties.

MQCreateSession

The `MQCreateSession` function creates a session, defines its behavior, and passes back a handle to the session.

```
MQCreateSession      (const MQConnectionHandle connectionHandle,
                     MQBool isTransacted,
                     MQAckMode acknowledgeMode,
                     MQReceiveMode receiveMode
                     MQSessionHandle * sessionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>connectionHandle</code>	The handle to the connection to which this session belongs. This handle is passed back to you by the MQCreateConnection function. You can create multiple sessions on a single connection.
<code>isTransacted</code>	An <code>MQBool</code> specifying whether this session is transacted. Specify <code>MQ_TRUE</code> if the session is transacted. In this case, the <code>acknowledgeMode</code> parameter is ignored.
<code>acknowledgeMode</code>	An enumeration of the possible kinds of acknowledgement modes for the session. See Table 4-3 on page 81 for information on these values. After you have created a session, you can determine its acknowledgement mode by calling the MQGetAcknowledgeMode function.
<code>receiveMode</code>	An enumeration specifying whether this session will do synchronous or asynchronous message receives. Specify <code>MQ_SESSION_SYNC_RECEIVE</code> or <code>MQ_SESSION_ASYNC_RECEIVE</code> . If the session is only for producing messages, the <code>receiveMode</code> has no significance. In that case, specify <code>MQ_SESSION_SYNC_RECEIVE</code> to optimize the session's resource use.
<code>sessionHandle</code>	A handle to this session. You will need to pass this handle to the functions you use to manage the session and to create destinations, consumers, and producers associated with this session.

The `MQCreateSession` function creates a new session and passes back a handle to it in the `sessionHandle` parameter. The number of sessions you can create for a single connection is limited only by system resources. A session is a single-thread context for producing and consuming messages. You can create multiple producers and consumers for a session, but you are restricted to use them serially. In effect, only a single logical thread of control can use them.

A session with a registered message listener is dedicated to the thread of control that delivers messages to the listener. This means that if you want to send messages, for example, you must create another session with which to do this. The only operations you can perform on a session with a registered listener, is to close the session or the connection.

After you create a session, you can create the producers, consumers, and destinations that use the session context to do their work.

- For a session that is not transacted, use the `MQRecoverSession` function to restart message delivery with the last unacknowledged message.
- For a session that is transacted, use the `MQRollBackSession` function to roll back any messages that were delivered within this transaction. Use the `MQCommitSession` function to commit all messages associated with this transaction.
- For a session that has `acknowledgeMode` set to `MQ_CLIENT_ACKNOWLEDGE`, use `MQAcknowledgeMessages` to acknowledge consumed messages.
- Use the `MQCloseSession` function to close a session and all its associated producers and consumers. This function also frees memory allocated for the session.

MQCreateTemporaryDestination

The `MQCreateTemporaryDestination` function creates a temporary destination and passes its handle back to you.

```
MQCreateTemporaryDestination (const MQSessionHandle sessionHandle
                              MQDestinationType destinationType,
                              MQDestinationHandle * destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session with which you want to associate this destination.
<code>destinationType</code>	An enum specifying the destination type, either <code>MQ_QUEUE_DESTINATION</code> or <code>MQ_TOPIC_DESTINATION</code> .
<code>destinationHandle</code>	Output parameter for the handle to the newly created temporary destination.

You can use a temporary destination to implement a simple request/reply mechanism. When you pass the handle of a temporary destination to the `MQSetMessageReplyTo` function, the consumer of the message can use that handle as the destination to which it sends a reply.

Temporary destinations are explicitly created by client applications; they are deleted when the connection is closed. They are maintained (and named) by the broker only for the duration of the connection for which they are created. Temporary destinations are system-generated uniquely for their connection and only their own connection is allowed to create message consumers for them.

For more information, see the *Message Queue Administration Guide*.

Common Errors

`MQ_INVALID_DESTINATION_TYPE`

`MQ_SESSION_CLOSED`

MQCreateTextMessage

The `MQCreatesTextMessage` function creates a text message and passes a handle to it back to you.

```
MQCreateTextMessage          ( MQMessageHandle * messageHandle );
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`messageHandle` Output parameter for the handle to the new, empty message.

After you obtain the handle to a text message, you can use this handle to define its content with the [MQSetStringProperty](#) function, to set its headers with the [MQSetMessageHeaders](#) function, and to set its properties with the [MQSetMessageProperties](#) function.

MQFreeConnection

The `MQFreeConnection` function deallocates memory assigned to the specified connection and to all resources associated with that connection.

```
MQFreeConnection          (MQConnectionHandle connectionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`connectionHandle` A handle to the connection you want to free.

You must call this function after you have closed the connection with the [MQCloseConnection](#) function and after all of the application threads associated with this connection and its dependent sessions, producers, and consumers have returned.

You must not call this function while an application thread is active in a library function associated with this connection or one of its dependent sessions, producers, consumers, and destinations.

Calling this function does not release resources held by a message or a destination associated with this connection. You must free memory allocated for a message or a destination by explicitly calling the `MQFreeMessage` or the `MQFreeDestination` function.

Common Errors

`MQ_STATUS_CONNECTION_NOT_CLOSED`

MQFreeDestination

The `MQFreeDestination` function frees memory allocated for the destination referenced by the specified handle.

```
MQFreeDestination          (MQDestinationHandle destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`destinationHandle` A handle to the destination you want to free.

Calling the `MQFreeConnection` or the `MQCloseSession` function does not automatically free destinations created for the connection or for the session.

MQFreeMessage

The `MQFreeMessage` function frees memory allocated for the message referenced by the specified handle.

```
MQFreeMessage          (MQMessageHandle messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`messageHandle` A handle to the message you want to free.

Calling the `MQFreeConnection` function does not automatically free messages associated with that connection.

MQFreeProperties

The `MQFreeProperties` function frees the memory allocated to the referenced properties object.

```
MQFreeProperties      (MQPropertiesHandle propertiesHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`propertiesHandle` A handle to the properties object you want to free.

You should not free a properties handle if the properties handle passed to a function becomes invalid on its return. If you do, you will get an error.

MQFreeString

The `MQFreeString` function frees the memory allocated for the specified `MQString`.

```
MQFreeString          (MQString statusString);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`statusString` An `MQString` returned by the `MQGetStatusString` function or by the `MQGetErrorTrace` function.

MQGetAcknowledgeMode

The `MQGetAcknowledgeMode` function passes back the acknowledgement mode of the specified session.

```
MQGetAcknowledgeMode    (const MQSessionHandle sessionHandle  
                        MQAckMode * ackMode);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session whose acknowledgement mode you want to determine.
<code>ackMode</code>	Output parameter for the <code>ackMode</code> . The <code>ackMode</code> returned can be one of four enumeration values. See Table 4-3 on page 81 for information about these values.

If you want to change the acknowledge mode, you need to create another session with the desired mode.

MQGetBoolProperty

The `MQGetBoolProperty` function passes back the value of the `MQBool` property for the specified key.

```
MQGetBoolProperty    (const MQPropertiesHandle propertiesHandle,  
                     ConstMQString key,  
                     MQBool * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetBytesMessageBytes

The `MQGetBytesMessageBytes` function passes back the address and size of a bytes message body.

```
MQGetBytesMessageBytes    (const MQMessageHandle messageHandle,
                           const MQInt8 * messageBytes
                           MQInt32 * messageBytesSize);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message that is passed to you when you receive a message.
<code>messageBytes</code>	Output parameter that contains the start address of the bytes that constitute the body of this bytes message.
<code>messageBytesSize</code>	Output parameter that contains the size of the message body in bytes.

After you obtain the handle to a message, you can use the [MQGetMessageType](#) to determine its type and, if the type is `MQ_BYTES_MESSAGE`, you can use the `MQGetBytesMessageBytes` function to retrieve the message bytes (message body).

The bytes message passed to you by this function is not a copy. You should not modify the bytes or attempt to free it.

MQGetDestinationType

The `MQGetDestinationType` passes back the type of the specified destination.

```
MQGetDestinationType    (const MQDestinationHandle destinationHandle,
                        MQDestinationType * destinationType);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>destinationHandle</code>	A handle to the destination whose type you want to know.
<code>destinationType</code>	Output parameter for the destination type; either <code>MQ_QUEUE_DESTINATION</code> or <code>MQ_TOPIC_DESTINATION</code> .

Use the `MQGetDestinationType` function to determine the type of a destination: queue or topic. There may be times when you do not know the type of the destination to which you are replying; for example, when you get a handle from the `MQGetMessageReplyTo` function. Because the semantics of queue and topic destinations differ, you need to determine the type of a destination in order to reply appropriately.

Once you have created a destination with a specified type, you cannot change the type dynamically. If you want to change the type of a destination, you need to free the destination using the [MQFreeDestination](#) function and then to create a new destination, with the desired type, using the [MQCreateDestination](#) or the [MQCreateTemporaryDestination](#) function.

MQGetErrorTrace

The `MQGetErrorTrace` function returns an `MQString` describing the error trace at the time when a function call failed for the calling thread.

```
MQString MQGetErrorTrace ()
```

Having found that a Message Queue function has not returned successfully, you can get an error trace when the error occurred by calling the `MQGetErrorTrace` function in the same thread that called the unsuccessful Message Queue function.

The `MQGetErrorTrace` function returns an `MQString` describing the error trace if it can determine this information. The function will return a `NULL` string if there is no error trace available.

The following is an example of an error trace output.

```
connect:../../../../src/share/cclient/io/TCPsocket.cpp:195:mq:-5981
readBrokerPorts:../../../../src/share/cclient/client/PortMapper
                    Client.cpp:48:mq:-5981
connect:../../../../src/share/cclient/client/protocol/
                    TCPProtocolHandler.cpp:111:mq:-5981
connectToBroker:../../../../src/share/cclient/client/Connection.
                    cpp:412:mq:-5981
openConnection:../../../../src/share/cclient/client/Connection.
                    cpp:227:mq:1900
MQCreateConnectionExt:../../../../src/share/cclient/cshim/
                    iMQConnectionShim.cpp:102:mq:1900
```

You must call the [MQFreeString](#) function to free the `MQString` returned by the `MQGetErrorTrace` function when you are done.

MQGetFloat32Property

The `MQGetFloat32Property` function passes back the value of the `MQFloat32` property for the specified key.

```
MQGetFloat32Property    (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQFloat32 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetFloat64Property

The `MQGetFloat64Property` function passes back the value of the `MQFloat64` property for the specified key.

```
MQGetFloat64Property    (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQFloat64 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetInt16Property

The `MQGetInt16Property` function passes back the value of the `MQInt16` property for the specified key.

```
MQGetInt16Property (const MQPropertiesHandle propertiesHandle,
                   ConstMQString key,
                   MQInt16 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetInt32Property

The `MQGetInt32Property` function passes back the value of the `MQInt32` property for the specified key.

```
MQGetInt32Property (const MQPropertiesHandle propertiesHandle,  
                  ConstMQString key,  
                  MQInt32 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetInt64Property

The `MQGetInt64Property` function passes back the value of the `MQInt64` property for the specified key.

```
MQGetint64Property    (const MQPropertiesHandle propertiesHandle,
                      ConstMQString key,
                      MQInt64 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetInt8Property

The `MQGetInt8Property` function passes back the value of the `MQInt8` property for the specified key.

```
MQGetInt8Property      (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQInt8 * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter for the property value.

Common Errors

`MQ_NOT_FOUND`

`MQ_INVALID_TYPE_CONVERSION`

MQGetMessageHeaders

The `MQGetMessageHeaders` function passes back a handle to the message headers.

```
MQGetMessageHeaders      (const MQMessageHandle messageHandle
                          MQPropertiesHandle * headersHandle) ;
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	The message handle.
<code>headersHandle</code>	Output parameter for the handle to the message header properties.

The `MQGetMessageHeaders` function passes back a handle to the message headers. The message header includes the fields described in [Table 4-5](#). Note that most of the fields are set by the send function; the client can optionally set only two of these fields for sending messages.

Table 4-5 Message Header Properties

Key	Type	Set By
<code>MQ_CORRELATION_ID_HEADER_PROPERTY</code>	<code>MQString</code>	Client (optional)
<code>MQ_MESSAGE_TYPE_HEADER_PROPERTY</code>	<code>MQString</code>	Client (optional)
<code>MQ_PERSISTENT_HEADER_PROPERTY</code>	<code>MQBool</code>	Send function
<code>MQ_EXPIRATION_HEADER_PROPERTY</code>	<code>MQInt64</code>	Send function
<code>MQ_PRIORITY_HEADER_PROPERTY</code>	<code>MQInt8</code>	Send function
<code>MQ_TIMESTAMP_HEADER_PROPERTY</code>	<code>MQInt64</code>	Send function
<code>MQ_MESSAGE_ID_HEADER_PROPERTY</code>	<code>MQString</code>	Send function
<code>MQ_REDELIVERED_HEADER_PROPERTY</code>	<code>MQBool</code>	Message Broker

You are responsible for freeing the `headersHandle` after you are done with it. Use the `MQFreeProperties` function to free the handle.

Use the `MQGetMessageProperties` function to determine whether any application-defined properties were set for this message and to find out their value.

MQGetMessageProperties

The `MQGetMessageProperties` function passes back the user-defined properties for a message.

```
MQGetMessageProperties    (const MQMessageHandle messageHandle,
                          MQPropertiesHandle * propsHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message whose properties you want to get.
<code>propertiesHandle</code>	Output parameter for the handle to the message properties.

The `MQGetMessageProperties` function allows you to get application-defined properties for a message. Properties allow an application, via message selectors, to select or filter messages on its behalf using application-specific criteria. Having obtained the handle, you can either use one of the `MQGet...Property` functions to get a value (if you know the key name) or you can iterate through the properties using the [MQPropertiesKeyIterationStart](#) function.

You will need to call the function [MQFreeProperties](#) to free the resources associated with this handle after you are done using it.

Common Errors

`MQ_NO_MESSAGE_PROPERTIES`

MQGetMessageReplyTo

The `MQGetMessageReplyTo` function passes back the destination where replies to this message should be sent.

```
MQGetMessageReplyTo (const MQMessageHandle messageHandle,
                    MQDestinationHandle * destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message expecting a reply. This is the handle that is passed back to you when you receive the message.
<code>destinationHandle</code>	Output parameter for the handle to the reply destination.

The sender uses the [MQSetMessageReplyTo](#) function to specify a destination where replies to the message can be sent. This can be a normal destination or a temporary destination. The receiving client can pass the message handle to the [MQGetMessageReplyTo](#) function and determine whether a destination for replies has been set up for the message by the sender and what that destination is. The consumer of the message can then use that handle as the destination to which it sends a reply.

You might need to call the [MQGetDestinationType](#) function to determine the type of the destination whose handle is returned to you: queue or topic so that you can set up your reply appropriately.

The advantage of setting up a temporary destination for replies is that Message Queue automatically creates a physical destination for you, rather than your having to have the administrator create one, when the broker's `auto.create.destination` property is turned off.

You are responsible for freeing the destination handle by calling the function [MQFreeDestination](#).

Common Errors

`MQ_NO_REPLY_TO_DESTINATION`

MQGetMessageType

The `MQGetMessageType` function passes back information about the type of a message: `MQ_TEXT_MESSAGE` or `MQ_BYTES_MESSAGE`.

```
MQGetMessageType          (const MQMessageHandle messageHandle,  
                           MQMessageType * messageType);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message whose type you want to determine.
<code>messageType</code>	Output parameter that contains the message type: <code>MQ_TEXT_MESSAGE</code> or <code>MQ_BYTES_MESSAGE</code> .

After you obtain the handle to a message, you can determine the type of the message using the `MQGetMessageType` function. Having determined its type, you can use the [MQGetTextMessageText](#) function or the [MQGetBytesMessageBytes](#) function to obtain the message content.

Note that other message types might be added in the future. You should not design your code so that it only expects two possible message types.

MQGetMetaData

The `MQGetMetaData` function returns name and version information for the current Message Queue product.

```
MQGetMetaData          (const MQConnectionHandle connectionHandle,
                       MQPropertiesHandle * propertiesHandle)
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>connectionHandle</code>	The handle to the connection that you want the meta information about.
<code>propertiesHandle</code>	Output parameter that contains the properties handle.

The Message Queue product you are using is identified by a name and a version number. For example: “Sun Java(tm) System Message Queue 3.5.0.1.” The version number consists of a major, minor, micro, and service pack component. For example, the major part of version 3.5.0.1 is 3; the minor is 5; the micro is 0; the service pack is 1.

The name and version information of the Message Queue product are set by the library when you call the [MQCreateConnection](#) function to create the connection. You can retrieve this information by calling the `MQGetMetaData` function and passing a properties handle. Once the function returns and passes the handle back, you can use one of the `MQGet...Properties` functions to determine the value of a property (key). These properties are described at the end of [Table 4-2 on page 77](#).

MQGetPropertyType

The `MQGetPropertyType` function returns the type of the property value for a property key in the specified properties handle.

```
MQGetPropertyType    (const MQPropertiesHandle  propertiesHandle,
                    ConstMQString key,
                    MQType * propertyType);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle that you want to access.
<code>key</code>	The property key for which you want to get the type of the property value.
<code>propertyType</code>	Output parameter for the type of the property value.

Use the appropriate `MQGet...Property` function to find the value of the specified property key.

Common Errors

`MQ_NOT_FOUND`

MQGetStatusCode

The `MQGetStatusCode` function returns the error code associated with specified status.

```
MQError MQGetStatusCode      (const MQStatus status);
```

Parameters

`status` The status returned by any Message Queue function that returns an `MQStatus`.

Having found that a Message Queue function has not returned successfully, you can determine the reason by passing the return status. This function will return the error code associated with the specified status. These codes are listed and described in [Appendix A on page 185](#).

Some functions might return an `MQStatus` that contains an NSPR or NSS library error code instead of a Message Queue error code when they fail. For NSPR and NSS library error codes, the `MQGetStatusString` function will return the symbolic name of the NSPR or NSS library error code. See NSPR and NSS public documentation for NSPR and NSS error code symbols and their interpretation at the following locations:

- For NSPR error codes, see the “NSPR Error Handling” chapter:
<http://www.mozilla.org/projects/nspr/reference/html/index.html>
- For SSL and SEC error codes, see the “NSS and SSL Error Codes” chapter:
<http://www.mozilla.org/projects/security/pki/nss/ref/ssl/>

To obtain an `MQString` that describes the error, use the `MQGetStatusString` function. To get an error trace associated with the error, use the `MQGetErrorTrace` function.

MQGetStatusString

The `MQGetStatusString` function returns an `MQString` describing the specified status.

```
MQString MQGetStatusString (const MQStatus status);
```

Parameters

<code>status</code>	The status returned by any Message Queue function that returns an <code>MQStatus</code> .
---------------------	---

Having found that a Message Queue function has not returned successfully, you can determine the reason why by passing the return status. This function will return an `MQString` describing the error associated with the specified status.

To obtain the error code for the specified `status`, use the [MQGetStatusCode](#) function. To get an error trace associated with the error, use the [MQGetErrorTrace](#) function.

You must call the `MQFreeString` function to free the `MQString` returned by the `MQGetStatusString` function when you are done.

MQGetStringProperty

The `MQGetStringProperty` function passes back the value of the specified key for the specified `MQString` property.

```
MQGetStringProperty    (const MQPropertiesHandle propertiesHandle,  
                       ConstMQString key,  
                       ConstMQString * value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose property value for the specified key you want to get.
<code>key</code>	The name of a property key.
<code>value</code>	Output parameter that points to the value of the specified key

You should not modify or attempt to free the value returned.

MQGetTextMessageText

The `MQGetTextMessageText` function passes back the contents of a text message.

```
MQGetTextMessageText      (const MQMessageHandle messageHandle,  
                          ConstMQString * messageText);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to an <code>MQ_TEXT_MESSAGE</code> message that is passed to you when you receive a message.
<code>messageText</code>	The output parameter that points to the message text.

After you obtain the handle to a message, you can use the [MQGetMessageType](#) to determine its type and, if the type is text, you can use the `MQGetTextMessageText` function to retrieve the message text.

The `MQString` passed to you by this function is not a copy. You should not modify the bytes or attempt to free it.

MQInitializeSSL

The `MQInitializeSSL` function initializes the SSL library.

```
MQInitializeSSL (ConstMQString certificateDatabasePath);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>certificateDatabasePath</code>	An <code>MQString</code> specifying the path to the directory that contains the certificate data base files.
--------------------------------------	--

The Message Queue C-API library uses NSS to support the SSL transport protocol between the Message Queue C client and the Message Queue broker.

Before you connect to a broker over SSL, you must initialize the SSL library by calling the `MQInitializeSSL` function. If your client uses secure connections, you must call this function once and only once before you create *any* connection, even if that connection is not an SSL connection.

The `certificateDatabasePath` parameter specifies the path to the NSS certificate database where `cert7.db` or `cert8.db`, `key3.db`, and `secmod.db` files are located.

The work required to configure secure communication includes initializing the SSL library using the `MQInitializeSSL` function. There may be additional work, depending on whether the broker is trusted (the default setting) and on whether you want to provide an additional means of verification if the broker is not trusted and the initial attempt to create a secure connection fails. For complete information see [“Working With Secure Connections” on page 35](#).

You must take care if the client application using secure Message Queue connections uses NSS (for other purposes) directly as well and does NSS initialization. For additional information, see [“Coordinating NSS Initialization” on page 37](#).

Common Errors

`MQ_INCOMPATIBLE_LIBRARY`

`MQ_SSL_ALREADY_INITIALIZED`

`MQ_SSL_INIT_ERROR`

MQPropertiesKeyIterationGetNext

The `MQPropertiesKeyIterationGetNext` function passes back the address of the next property key in the referenced properties handle.

```
MQPropertiesKeyIterationGetNext      (const MQPropertiesHandle
                                     propertiesHandle,
                                     ConstMQString * key);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A properties handle whose contents you want to access.
<code>key</code>	The output parameter for the next properties key in the iteration. You should not attempt to modify or free it.

To get message properties:

1. Start the process by calling the `MQPropertiesKeyIterationStart` function.
2. Loop using the [MQPropertiesKeyIterationHasNext](#) function.
3. Extract the name of each property key by calling the [MQPropertiesKeyIterationGetNext](#) function.
4. Determine the type of the property value for a given key by calling the [MQGetPropertyType](#) function.
5. Use the appropriate `MQGet...Property` function to find the property value for the specified property key.

If you know the property key, you can just use the appropriate `MQGet...Property` function to access its value.

You should not modify or free the property key that is passed back to you by this function. Note that this function is not multi-thread-safe.

MQPropertiesKeyIterationHasNext

The `MQPropertiesKeyIterationHasNext` function returns `MQ_TRUE` if there are additional property keys left in the iteration.

```
MQPropertiesKeyIterationHasNext
    (const MQPropertiesHandle  propertiesHandle);
```

Return Value

`MQBool`

Parameters

`propertiesHandle` A properties handle that you want to access.

To get message properties:

1. Start the process by calling the `MQPropertiesKeyIterationStart` function.
2. Loop using the `MQPropertiesKeyIterationHasNext` function.
3. Extract the name of each property key by calling the `MQPropertiesKeyIterationGetNext` function.
4. Determine the type of the property value for a given key by calling the `MQGetPropertyType` function.
5. Use the appropriate `MQGet...Property` function to find the value for the specified property key.

If you know the property key, you can just use the appropriate `MQGet...Property` function to get its value. Note that this function is not multi-thread-safe.

MQPropertiesKeyIterationStart

The `MQPropertiesKeyIterationStart` function starts or resets the iteration process or the specified properties handle.

```
MQPropertiesKeyIterationStart      (const PropertiesHandle
                                   propertiesHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`propertiesHandle` A properties handle that you want to access.

To get message properties:

1. Start the process by calling the `MQPropertiesKeyIterationStart` function.
2. Loop using the [MQPropertiesKeyIterationHasNext](#) function.
3. Extract the name of each property key by calling the [MQPropertiesKeyIterationGetNext](#) function.
4. Determine the type of the property value for a given key by calling the [MQGetPropertyType](#) function.
5. Use the appropriate `MQGet...Property` function to find the property value for the specified property key.

If you know the property key, you can just use the appropriate `MQGet...Property` function to get its value. Note that this function is not multi-thread-safe.

MQReceiveMessageNoWait

The `MQReceiveMessageNoWait` function passes a handle back to a message delivered to the specified consumer if a message is available.

```
MQReceiveMessageNoWait      (const MQConsumerHandle consumerHandle,
                             MQMessageHandle * messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>consumerHandle</code>	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
<code>messageHandle</code>	Output parameter for the handle to the message to be received. You are responsible for freeing the message handle when you are done by calling the MQFreeMessage function.

This function can only be called if the session is created with receive mode `MQ_SESSION_SYNC_RECEIVE`. The `MQReceiveMessageNoWait` function passes a handle back to you in the `messageHandle` parameter if there is a message arrived for the consumer specified by the `consumerHandle` parameter. If there is no message for the consumer, the function returns immediately with an error.

When you create a session, you specify one of several acknowledge modes for that session; these are described in [Table 4-3](#). If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge messages that you have received. For more information, see the description of the function [MQAcknowledgeMessages](#).

Because distributed applications involve greater processing time, such an application might not behave as expected if it were run locally. For example, calling the `MQReceiveMessageNoWait` function might return `MQ_NO_MESSAGE` even when there is a message available to be retrieved.

If a client connects to the broker and immediately calls the `MQReceiveMessageNoWait`, it is possible that the message queued for the consuming client is in the process of being transmitted from the broker to the client. The client runtime has no knowledge of what is on the broker, so when it sees that there is no message available on the client's internal queue, it returns with `MQ_NO_MESSAGE`.

You can avoid this problem by having your client use one of the synchronous receive methods that specifies a timeout interval.

You can use the `MQReceiveMessageWait` function if you want the receive function to block while waiting for a message to arrive. You can use the `MQReceiveMessageWithTimeout` function to wait for a specified time for a message to arrive.

Common Errors

`MQ_NOT_SYNC_RECEIVE_MODE`

`MQ_CONCURRENT_ACCESS`

`MQ_NO_MESSAGE`

`MQ_CONSUMER_CLOSED`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQReceiveMessageWait

The `MQReceiveMessageWait` function passes a handle back to a message delivered to the specified consumer when the message becomes available.

```
MQReceiveMessageWait      (const MQConsumerHandle consumerHandle,
                          MQMessageHandle * messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>consumerHandle</code>	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
<code>messageHandle</code>	Output parameter for the handle to the message to be received. You are responsible for freeing the message handle when you are done by calling the MQFreeMessage function.

This function can only be called if the session is created with receive mode `MQ_SESSION_SYNC_RECEIVE`. The `MQReceiveMessageWait` function passes a handle back to you in the `messageHandle` parameter if there is a message arrived for the consumer specified by the `consumerHandle` parameter. If there is no message for the consumer, the function blocks until a message is delivered.

When you create a session, you specify one of several acknowledge modes for that session; these are described in [Table 4-3](#). If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the `MQAcknowledgeMessages` function to acknowledge messages that you have received. For more information, see the description of the function [MQAcknowledgeMessages](#).

You can use the [MQReceiveMessageNoWait](#) function instead if you do not want to block while waiting for a message to arrive. You can use the function [MQReceiveMessageWithTimeout](#) to wait for a specified time for a message to arrive.

Common Errors

MQ_NOT_SYNC_RECEIVE_MODE

MQ_CONCURRENT_ACCESS

MQ_CONSUMER_CLOSED

MQ_SESSION_CLOSED

MQ_BROKER_CONNECTION_CLOSED

MQReceiveMessageWithTimeout

The `MQReceiveMessageWithTimeout` function passes a handle back to a message delivered to the specified consumer if a message is available within the specified amount of time.

```
MQReceiveMessageWithTimeout    (const MQConsumerHandle consumerHandle,
                               MQInt32 timeoutMilliseconds,
                               MQMessageHandle * messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

<code>consumerHandle</code>	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
<code>timeoutMilliseconds</code>	The number of milliseconds to wait for a message to arrive.
<code>messageHandle</code>	Output parameter for the handle to the message to be received. You are responsible for freeing the message handle when you are done by calling the MQFreeMessage function.

This function can only be called if the session is created with receive mode `MQ_SESSION_SYNC_RECEIVE`. The [MQReceiveMessageWithTimeout](#) function passes a handle back to you in the `messageHandle` parameter if a message arrives for the consumer specified by the `consumerHandle` parameter in the amount of time specified by the `timeoutMilliseconds` parameter. If no message arrives within the specified amount of time, the function returns an error.

When you create a session, you specify one of several acknowledge modes for that session; these are described in [Table 4-3](#). If you specify `MQ_CLIENT_ACKNOWLEDGE` as the acknowledge mode for the session, you must explicitly call the [MQAcknowledgeMessages](#) function to acknowledge messages that you have received. For more information, see the description of the function [MQAcknowledgeMessages](#).

You can use the [MQReceiveMessageWait](#) function to block while waiting for a message to arrive. You can use the [MQReceiveMessageNoWait](#) function if you do not want to wait for the message to arrive.

Common Errors

MQ_NOT_SYNC_RECEIVE_MODE

MQ_CONCURRENT_ACCESS

MQ_TIMEOUT_EXPIRED

MQ_CONSUMER_CLOSED

MQ_SESSION_CLOSED

MQ_BROKER_CONNECTION_CLOSED

MQRecoverSession

The `MQRecoverSession` function stops message delivery and restarts message delivery with the oldest unacknowledged message.

```
MQRecoverSession      (const MQSessionHandle sessionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`sessionHandle` The handle to the session that you want to recover.

You can only call this function for sessions that are not transacted. To rollback message delivery for a transacted session, use the [MQRollBackSession](#) function. This function may be most useful if you use the `MQ_CLIENT_ACKNOWLEDGE` mode.

All consumers deliver messages in a serial order. Acknowledging a received message automatically acknowledges all messages that have been delivered to the client.

Restarting a session causes it to take the following actions:

- Stop message delivery in this session
- Mark all messages that might have been delivered but not acknowledged as “redelivered”
- Restart the delivery sequence including all unacknowledged messages that had been previously delivered. (Redelivered messages might not be delivered in their original delivery order.)

Common Errors

`MQ_TRANSACTED_SESSION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQRollBackSession

The `MQRollBackSession` function rolls back a transaction associated with the specified session.

```
MQRollBackSession      (const MQSessionHandle sessionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`sessionHandle` The handle to the transacted session that you want to roll back.

A transacted session groups messages into an atomic unit known as a transaction. As messages are produced or consumed within a transaction, the broker tracks the various sends and receives, completing these operations only when you call the [MQCommitSession](#) function.

If a send or receive operation fails, you must use the `MQRollBackSession` function to roll back the entire transaction. This means that those messages that have been sent are destroyed and those messages that have been consumed are automatically recovered.

Common Errors

`MQ_NOT_TRANSACTED_SESSION`

`MQ_CONCURRENT_ACCESS`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQSendMessage

The `MQSendMessage` function sends a message using the specified producer.

```
MQSendMessage          (const MQProducerHandle producerHandle,
                       const MQMessageHandle messageHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>producerHandle</code>	The handle to the producer sending this message. This handle is passed back to you by the MQCreateMessageProducerForDestination function.
<code>messageHandle</code>	A handle to the message you want to send.

The `MQSendMessage` function sends the specified message on behalf of the specified producer to the destination associated with the message producer. If you use this function to send a message, the following message header fields are set to default values when the send completes.

- `MQ_PERSISTENT_HEADER_PROPERTY` will be set to `MQ_PERSISTENT_DELIVERY`.
This means that the calling thread will be blocked, waiting for the broker to acknowledge receipt of your messages, unless you set the connection property `MQ_ACK_ON_PRODUCE_PROPERTY` to `MQ_FALSE`.
- `MQ_PRIORITY_HEADER_PROPERTY` will be set to 4.
- `MQ_EXPIRATION_HEADER_PROPERTY` will be set to 0, which means that the message will never expire.

If you set those message properties, they will be ignored when a message is sent. To send a message with these properties set to different values, you can use the [MQSendMessageExt](#) function to specify different values for these properties.

You cannot use this function with a producer that is created without a specified destination.

Common Errors

MQ_PRODUCER_NO_DESTINATION

MQ_PRODUCER_CLOSED

MQ_SESSION_CLOSED

MQ_BROKER_CONNECTION_CLOSED

MQSendMessageExt

The `MQSendMessageExt` function sends a message using the specified producer and allows you to specify selected message header properties.

```
MQSendMessageExt          (const MQProducerHandle producerHandle,
                           const MQMessageHandle messageHandle
                           MQDeliveryMode msgDeliveryMode,
                           MQInt8 msgPriority,
                           MQInt64 msgTimeToLive);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>producerHandle</code>	The handle to the producer sending this message. This handle is passed back to you by the MQCreateMessageProducerForDestination function.
<code>messageHandle</code>	A handle to the message you want to send.
<code>msgDeliveryMode</code>	An enum <code>MQ_PERSISTENT_DELIVERY</code> <code>MQ_NONPERSISTENT_DELIVERY</code> .
<code>msgPriority</code>	A integer value of 0 through 9; 0 being the lowest priority and 9 the highest.
<code>msgTimeToLive</code>	An integer value specifying in milliseconds how long the message will live before it expires. When a message is sent, its expiration time is calculated as the sum of its time-to-live value and current GMT. A value of 0 indicates that the message will never expire.

The `MQSendMessageExt` function sends the specified message on behalf of the specified producer to the destination associated with the message producer. Use this function if you want to change the default values for the message header properties as shown in the next table.

Property	Default value
<code>msgDeliveryMode</code>	<code>MQ_PERSISTENT_DELIVERY</code>
<code>msgPriority</code>	4
<code>msgTimeToLive</code>	0, meaning no expiration limit

If you set these message headers using the `MQSetMessageHeaders` function before the send, they will be ignored when the message is sent. When the send completes, these message headers hold the values that are set by the send.

You cannot use this function with a producer that is created without a specified destination.

You can set the broker property `MQ_ACK_ON_PRODUCE_PROPERTY` to make sure that the message has reached its destination on the broker:

- By default, the broker acknowledges receiving persistent messages only.
- If you set the property to `MQ_TRUE`, the broker acknowledges receipt of all messages (persistent and non-persistent) from the producing client.
- If you set the property to `MQ_FALSE`, the broker does not acknowledge receipt of any message (persistent or non-persistent) from the producing client.

Note that “acknowledgement” in this case is not programmatic but internally implemented. That is, the client thread is blocked and does not return until the broker acknowledges messages it receives from the producing client.

Common Errors

`MQ_PRODUCER_NO_DESTINATION`

`MQ_INVALID_PRIORITY`

`MQ_INVALID_DELIVERY_MODE`

`MQ_PRODUCER_CLOSED`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQSendMessageToDestination

The `MQSendMessageToDestination` function sends a message using the specified producer to the specified destination.

```
MQSendMessageToDestination
(
    const MQProducerHandle producerHandle,
    const MQMessageHandle messageHandle,
    const MQDestinationHandle destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>producerHandle</code>	The handle to the producer sending this message. This handle is passed back to you by the MQCreateMessageProducer function.
<code>messageHandle</code>	A handle to the message you want to send.
<code>destinationHandle</code>	A handle to the destination where you want to send the message.

The `MQSendMessageToDestination` function sends the specified message on behalf of the specified producer to the specified destination. If you use this function to send a message, the following message header fields are set as follows when the send completes.

- `MQ_PERSISTENT_HEADER_PROPERTY` will be set to `MQ_PERSISTENT_DELIVERY`.
This means that the caller will be blocked, waiting for broker acknowledgement for the receipt of your messages unless you set the connection property `MQ_ACK_ON_PRODUCE_PROPERTY` to `MQ_FALSE`.
- `MQ_PRIORITY_HEADER_PROPERTY` will be set to 4.
- `MQ_EXPIRATION_HEADER_PROPERTY` will be set to 0, which means that the message will never expire.

To send a message with these properties set to different values, you must use the [MQSendMessageToDestinationExt](#) function, which allows you to set these three header properties.

If you set these message headers using the `MQSetMessageHeaders` function before the send, they will be ignored when the message is sent. When the send completes, these message headers hold the values that are set by the send.

You cannot use this function with a producer that is created without a specified destination.

Common Errors

`MQ_PRODUCER_HAS_DEFAULT_DESTINATION`

`MQ_PRODUCER_CLOSED`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQSendMessageToDestinationExt

The `MQSendMessageToDestinationExt` function sends a message to the specified destination for the specified producer and allows you to set selected message header properties.

```
MQSendMessageToDestinationExt
```

```
(const MQProducerHandle producerHandle,
 const MQMessageHandle messageHandle,
 const MQDestinationHandle destinationHandle,
 MQDeliveryMode msgDeliveryMode,
 MQInt8 msgPriority,
 MQInt64 msgTimeToLive);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>producerHandle</code>	The handle to the producer sending this message. This handle is passed back to you when you call the MQCreateMessageProducer function.
<code>messageHandle</code>	A handle to the message you want to send.
<code>destinationHandle</code>	A handle to the destination where you want to send the message.
<code>msgDeliveryMode</code>	An enum of either <code>MQ_PERSISTENT_DELIVERY</code> or <code>MQ_NONPERSISTENT_DELIVERY</code> .
<code>msgPriority</code>	A integer value of 0 through 9; 0 being the lowest priority and 9 the highest.
<code>msgTimeToLive</code>	An integer value specifying in milliseconds how long the message will live before it expires. When a message is sent, its expiration time is calculated as the sum of its time-to-live value and current GMT. A value of 0 indicates that the message will never expire.

The `MQSendMessageToDestinationExt` function sends the specified message on behalf of the specified producer to the specified destination. Use this function if you want to change the default values for the message header properties as shown below:

Property	Default value
<code>msgDeliveryMode</code>	<code>MQ_PERSISTENT_DELIVERY</code>
<code>msgPriority</code>	4
<code>msgTimeToLive</code>	0, meaning no expiration limit

If these default values suit you, you can use the `MQSendMessageToDestination` function to send the message.

You cannot use this function with a producer that is created with a specified destination.

You can set the broker property `MQ_ACK_ON_PRODUCE_PROPERTY` to make sure that the message has reached its destination on the broker:

- By default, the broker acknowledges receiving persistent messages only from the producing client.
- If you set the property to `MQ_TRUE`, the broker acknowledges receipt of all messages (persistent and non-persistent) from the producing client.
- If you set the property to `MQ_FALSE`, the broker does not acknowledge receipt of any message (persistent or non-persistent) from the producing client.

Note that “acknowledgement” in this case is not programmatic but internally implemented. That is, the client thread is blocked and does not return until the broker acknowledges messages it receives.

Common Errors

`MQ_PRODUCER_HAS_DEFAULT_DESTINATION`

`MQ_INVALID_PRIORITY`

`MQ_INVALID_DELIVERY_MODE`

`MQ_PRODUCER_CLOSED`

`MQ_SESSION_CLOSED`

`MQ_BROKER_CONNECTION_CLOSED`

MQSetBoolProperty

The `MQSetBoolProperty` function sets an `MQBool` property with the specified key to the specified value.

```
MQSetBoolProperty      (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQBool value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of the property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQBool</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetBytesMessageBytes

The `MQSetBytesMessageBytes` function defines the body for a bytes message.

```
MQSetBytesMessageBytes    (const MQMessageHandle messageHandle,
                           const MQInt8 * messageBytes,
                           MQInt32 messageSize);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to an <code>MQ_BYTES_MESSAGE</code> message whose body you want to set.
<code>messageBytes</code>	A pointer to the bytes you want to set. The library makes a copy of the message bytes.
<code>messageSize</code>	An integer specifying the number of bytes in <code>messageBytes</code> .

After you obtain the handle to a bytes message from `MQCreateBytesMessage`, you can use this handle to define its body with the `MQSetBytesMessageBytes` function, to set its application-defined properties with the `MQSetMessageProperties` function, and to set certain message headers with the `MQSetMessageHeaders` function.

MQSetFloat32Property

The `MQSetFloat32Property` function sets an `MQFloat32` property with the specified key to the specified value.

```
MQSetFloat32Property    (const MQPropertiesHandle propertiesHandle,
                        ConstMQString key,
                        MQFloat32 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQFloat32</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetFloat64Property

The `MQSetFloat64Property` function sets an `MQFloat64` property with the specified key to the specified value.

```
MQSetFloat64Property    (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQFloat64 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQFloat64</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetInt16Property

The `MQSetInt16Property` function sets an `MQInt16` property with the specified key to the specified value.

```
MQSetInt16Property    (const MQPropertiesHandle propertiesHandle,  
                      ConstMQString key,  
                      MQInt16 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQInt16</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetInt32Property

The `MQSetInt32Property` function sets an `MQInt32` property with the specified key to the specified value.

```
MQSetInt32Property    (const MQPropertiesHandle propertiesHandle,  
                      ConstMQString key,  
                      MQInt32 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQInt32</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetInt64Property

The `MQSetInt64Property` function sets an `MQInt64` property with the specified key to the specified value.

```
MQSetInt64Property    (const MQPropertiesHandle propertiesHandle,  
                      ConstMQString key,  
                      MQInt64 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set.
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQInt64</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetInt8Property

The `MQSetInt8Property` function sets an `MQInt8` property with the specified key to the specified value.

```
MQSetInt8Property      (const MQPropertiesHandle propertiesHandle,  
                        ConstMQString key,  
                        MQInt8 value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set
<code>key</code>	The name of a property key. The library makes a copy of the property key.
<code>value</code>	The <code>MQInt8</code> property value.

Common Errors

`MQ_HASH_VALUE_ALREADY_EXISTS`

MQSetMessageHeaders

The `MQSetMessageHeaders` function creates the header part of the message.

```
MQSetMessageHeaders      (const MQMessageHandle messageHandle
                          MQPropertiesHandle headersHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message.
<code>headersHandle</code>	A handle to the header properties object. This handle will be invalid after the function returns.

After you have created a properties handle and defined values for message header properties using one of the `MQSet...Property` functions, you can pass the handle to the `MQSetMessageHeaders` function to define the message header properties.

The message header properties are described in [Table 4-6](#). For sending messages, the client can only set two of these: the correlation ID property and the message type property. The client is not required to set these; they are provided for the client's convenience. For example, the client can use the key `MQ_MESSAGE_TYPE_HEADER_PROPERTY` to sort incoming messages according to application-defined message types.

Table 4-6 Message Header Properties

Key	Type	Set By
<code>MQ_CORRELATION_ID_HEADER_PROPERTY</code>	<code>MQString</code>	Client (optional)
<code>MQ_MESSAGE_TYPE_HEADER_PROPERTY</code>	<code>MQString</code>	Client (optional)
<code>MQ_PERSISTENT_HEADER_PROPERTY</code>	<code>MQBool</code>	Send function
<code>MQ_EXPIRATION_HEADER_PROPERTY</code>	<code>MQInt64</code>	Send function
<code>MQ_PRIORITY_HEADER_PROPERTY</code>	<code>MQInt8</code>	Send function
<code>MQ_TIMESTAMP_HEADER_PROPERTY</code>	<code>MQInt64</code>	Send function
<code>MQ_MESSAGE_ID_HEADER_PROPERTY</code>	<code>MQString</code>	Send function

Table 4-6 Message Header Properties (*Continued*)

Key	Type	Set By
MQ_REDELIVERED_HEADER_PROPERTY	MQBool	Message Broker

Header properties that are not specified in the `headersHandle` are not affected. You cannot use this function to override header properties that are set by the broker or the send function. The header properties for persistence, expiration, and priority (Table 4-6) are set to default values if the user called the `MQSendMessage` or `MQSendMessageToDestination` function, or they are set to values the user specifies (in parameters) if the user called the `MQSendMessageExt` or the `MQSendMessageToDestinationExt` function.

Use the `MQSetBytesMessageBytes` function or the `MQSetStringProperty` to set the body of a message. Use the `MQSetMessageProperties` function to set the application-defined properties of a message that are not part of the header.

Common Errors

MQ_PROPERTY_WRONG_VALUE_TYPE

MQSetMessageProperties

The `MQSetMessageProperties` function sets the specified properties for a message.

```
MQSetMessageProperties    (const MQMessageHandle messageHandle,
                          MQPropertiesHandle propsHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message whose application-defined properties you want to set.
<code>propertiesHandle</code>	A handle to a properties object that you have created and set using one of the set property functions. This handle is invalid after the function returns.

After you obtain the handle to a message, you can use this handle to define its body with the [MQSetStringProperty](#) function or the [MQSetBytesMessageBytes](#) function, to set its header properties with the [MQSetMessageHeaders](#) function, and to set its application-defined properties with the [MQSetMessageProperties](#) function.

Property values are set prior to sending a message. The `MQSetMessageProperties` function allows you to set application-defined properties for a message. Properties allow an application, via message selectors, to select or filter, messages on its behalf using application-specific criteria.

You define the message properties and their values using the [MQCreateProperties](#) function to create a properties object, then you use one of the set property functions to define each key and value in it. See [“Working With Properties” on page 28](#) for more information.

To change the properties of a message, call the `MQSetMessageProperties` function, passing a different properties handle.

MQSetMessageReplyTo

The `MQSetMessageReplyTo` function specifies the destination where replies to this message should be sent.

```
MQSetMessageReplyTo
    (const MQMessageHandle messageHandle,
     const MQDestinationHandle destinationHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message expecting a reply.
<code>destinationHandle</code>	The destination to which the reply is sent. Usually this is a handle to a destination that you created using the MQCreateDestination function or the function MQCreateTemporaryDestination . The handle is still valid when this function returns.

The sender uses the `MQSetMessageReply` function to specify a destination where replies to the message can be sent. This can be a normal destination or a temporary destination. The receiver of a message can use the [MQGetMessageReplyTo](#) function to determine whether a sender has set up a destination where replies are to be sent. The advantage of setting up a temporary destination for replies is that Message Queue automatically creates a physical destination for you, rather than your having to have the administrator create one if the broker's `auto_create_destination` property is turned off.

MQSetStringProperty

The `MQSetStringProperty` function sets an `MQString` property with the specified key `t` to the specified value.

```
MQSetStringProperty (const MQPropertiesHandle propertiesHandle,
                    ConstMQString key,
                    ConstMQString value);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>propertiesHandle</code>	A handle to the properties object whose property value for the specified key you want to set. You get this handle from the MQCreateProperties function.
<code>key</code>	The name of a property key. The library makes a copy of the property key
<code>value</code>	The property value to set. The library makes a copy of the value.

The library makes a copy of the property key and also makes a copy of the value.

MQSetTextMessageText

The `MQSetTextMessageText` function defines the body for a text message.

```
MQSetTextMessageText      (const MQMessageHandle messageHandle,  
                           ConstMQString messageText);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>messageHandle</code>	A handle to a message whose text body you want to set.
<code>messageText</code>	An <code>MQString</code> specifying the message text. The library makes a copy of the message text.

After you obtain the handle to a text message, you can use this handle to define its body with the [MQSetStringProperty](#) function. You can set its application-defined properties with the [MQSetMessageProperties](#) function, and you can set certain message headers with the [MQSetMessageHeaders](#) function.

MQStartConnection

The `MQStartConnection` function starts the specified connection to the broker and starts or resumes message delivery.

```
MQStartConnection      (const MQConnectionHandle connectionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`connectionHandle` The handle to the connection that you want to start. This handle is the handle that is created and passed back to you by the [MQCreateConnection](#) function.

When a connection is created it is in stopped mode. Until you call this function, messages are not delivered to any consumers. Call this function to start a connection or to restart a connection that has been stopped with the [MQStopConnection](#) function. To create an asynchronous consumer, you could have the connection in stopped mode, and start or restart the connection after you have set up the asynchronous message consumer.

Use the [MQCloseConnection](#) function to close a connection, and then use the [MQFreeConnection](#) function to free the memory allocated to the connection.

Common Errors

`MQ_BROKER_CONNECTION_CLOSED`

MQStatusIsError

The `MQStatusIsError` function returns `MQ_TRUE` if the status parameter passed to it represents an error.

```
MQBool MQStatusIsError      (const MQStatus status);
```

Parameters

<code>status</code>	The status returned by any Message Queue function that returns an <code>MQStatus</code> .
---------------------	---

Nearly all Message Queue C library functions return an `MQStatus`. You can pass this status result to the `MQStatusIsError` function to determine whether your call succeeded or failed. If the `MQStatusIsError` function returns `MQ_TRUE (=1)`, the function failed; if it returns `MQ_FALSE (=0)`, the function returned successfully.

If the `MQStatusIsError` returns `MQ_TRUE`, you can get more information about the error that occurred by passing the status returned to the [MQGetStatusCode](#) function. This function will return the error code associated with the specified status.

To obtain an `MQString` that describes the error, use the [MQGetStatusString](#) function. To get an error trace associated with the error, use the [MQGetErrorTrace](#) function.

MQStopConnection

The `MQStopConnection` function stops the specified connection to the broker. This stops the broker from delivering messages.

```
MQStopConnection      (const MQConnectionHandle connectionHandle);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

`connectionHandle` The handle to the connection that you want to stop. This handle is passed back to you by the [MQCreateConnection](#) function.

You can restart message delivery by calling the [MQStartConnection](#) function. When the connection has stopped, delivery to all the connection's message consumers is inhibited: synchronous receives block, and messages are not delivered to message listeners. This call blocks until receives and/or message listeners in progress have completed.

You should not call `MQStopConnection` in a message listener callback function.

Use the [MQCloseConnection](#) function to close a connection, and then use the [MQFreeConnection](#) function to free the memory allocated to the connection.

Common Errors

`MQ_BROKER_CONNECTION_CLOSED`

`MQ_CONCURRENT_DEADLOCK`

MQUnsubscribeDurableMessageConsumer

The `MQUnsubscribeDurableMessageConsumer` function unsubscribes the specified durable message consumer.

```
MQUnsubscribeDurableMessageConsumer
    (const MQSessionHandle sessionHandle,
     ConstMQString durableName);
```

Return Value

`MQStatus`. See the [MQStatusIsError](#) function for more information.

Parameters

<code>sessionHandle</code>	The handle to the session to which this consumer belongs. This handle is created and passed back to you by the MQCreateSession function.
<code>durableName</code>	An <code>MQString</code> specifying the name of the durable subscriber.

When you call the `MQUnsubscribeDurableMessageConsumer` function, the client runtime instructs the broker to delete the state information that the broker maintains for this consumer. If you try to delete a durable consumer while it has an active topic subscriber or while a received message has not been acknowledged in the session, you will get an error. You should only unsubscribe a durable message consumer after closing it.

Common Errors

`MQ_CANNOT_UNSUBSCRIBE_ACTIVE_CONSUMER`

`MQ_CONSUMER_NOT_FOUND`

Header Files

The Message Queue C-API is defined in the header files listed in [Table 4-7](#). The files are listed in alphabetical order. The file `mqcrt.h` includes all the Message Queue C-API header files.

Table 4-7 Message Queue C-API Header Files

File Name	Contents
<code>mqbasictypes.h</code>	Defines the types <code>MQBool</code> , <code>MQInt8</code> , <code>MQInt16</code> , <code>MQInt32</code> , <code>MQInt64</code> , <code>MQFloat32</code> , <code>MQFloat64</code> .
<code>mqbytes-message.h</code>	Function prototypes for creating, getting, setting bytes message.
<code>mqcallback-types.h</code>	Asynchronous receive and connection exception handling callback types.
<code>mqconnection.h</code>	Function prototypes for creating, managing, and closing connections. Function prototype for creating session.
<code>mqconnection-props.h</code>	Connection property constants
<code>mqconsumer.h</code>	Function prototypes for synchronous receives and closing the consumer.
<code>mqcrt.h</code>	All Message Queue C-API public header files.
<code>mqdestination.h</code>	Function prototypes to free destinations and get information about destinations.
<code>mqerrors.h</code>	Error codes
<code>mqheader-props.h</code>	Message header property constants
<code>mqmessage.h</code>	Function prototypes for getting and setting parts of message, freeing message, and acknowledging message.
<code>mqproducer.h</code>	Function prototypes for sending messages and closing the message producer.
<code>mqproperties.h</code>	Function prototypes for creating, setting, and getting properties
<code>mqsession.h</code>	Function prototypes for managing and closing sessions; for creating destinations, message producers and message consumers.
<code>mqssl.h</code>	Function declaration for initializing the SSL library.
<code>mqstatus.h</code>	Function prototypes for getting error information.

Table 4-7 Message Queue C-API Header Files (*Continued*)

File Name	Contents
<code>mqtext-message.h</code>	Function prototypes for creating, getting, setting text message.
<code>mqtypes.h</code>	Enumeration of types that can be stored in a properties object, of types of message that can be received, of acknowledgement modes, of delivery modes, of destination types, of session receiving modes, and of handle types.
<code>mqversion.h</code>	Version information constant definitions.

Message Queue C API Error Codes

Having found that a Message Queue function has not returned successfully, you can determine the reason by passing the return status of that function to the `MQGetStatusCode` function, which returns the error code associated with the specified status. This appendix lists the error codes that can be returned and provides a description that is associated with that code. You can retrieve the error string (description) by calling the `MQGetStatusString` function.

Some Message Queue functions, when they fail, might return an `MQStatus` result that contains an NSPR or NSS library error code instead of a Message Queue error code. For NSPR and NSS library error codes, the `MQGetStatusString` function returns the symbolic name of the NSPR or NSS library error code. Please see NSPR and NSS public documentation for NSPR and NSS error code symbols and their interpretation at the following locations:

- For NSPR error codes, see the “NSPR Error Handling” chapter:
<http://www.mozilla.org/projects/nspr/reference/html/index.html>
- For NSS error codes, see the “NSS and SSL Error Codes” chapter:
<http://www.mozilla.org/projects/security/pki/nss/ref/ssl/>

When checking a Message Queue function for return errors, you should only reference the Message Queue common error code symbol names in order to maintain maximum compatibility with future releases. For each function, [Chapter 4, “Reference” on page 73](#), lists the common error codes that can be returned by that function.

For information on error handling, see [“Error Handling” on page 53](#).

Error Codes

[Table A-1](#) lists the error codes in alphabetical order. For each code listed, it provides a description for the error code and notes whether it is a common error (Common).

Table A-1 Message Queue C Client Error Codes

Code	Common	Description
MQ_ACK_STATUS_NOT_OK		Acknowledgement status is not OK
MQ_ADMIN_KEY_AUTH_MISMATCH		Admin key authorization mismatch
MQ_BAD_VECTOR_INDEX		Bad vector index
MQ_BASE64_ENCODE_FAILURE		Base 64 encode failure.
MQ_BASIC_TYPE_SIZE_MISMATCH		Message Queue basic type size mismatch
MQ_BROKER_BAD_REQUEST		Broker: bad request
MQ_BROKER_BAD_VERSION		Broker: bad version
MQ_BROKER_CONFLICT		Broker: conflict
MQ_BROKER_CONNECTION_CLOSED	X	Broker connection is closed.
MQ_BROKER_ENTITY_TOO_LARGE		Broker: entity too large
MQ_BROKER_ERROR		Broker: error
MQ_BROKER_FORBIDDEN		Broker: forbidden
MQ_BROKER_GONE		Broker: gone
MQ_BROKER_INVALID_LOGIN		Broker: invalid login
MQ_BROKER_NOT_ALLOWED		Broker: not allowed
MQ_BROKER_NOT_FOUND		Broker: not found
MQ_BROKER_NOT_IMPLEMENTED		Broker: not implemented
MQ_BROKER_PRECONDITION_FAILED		Broker: precondition failed
MQ_BROKER_RESOURCE_FULL		Broker: resource full
MQ_BROKER_TIMEOUT		Broker: timeout
MQ_BROKER_UNAUTHORIZED		Broker: unauthorized
MQ_BROKER_UNAVAILABLE		Broker: unavailable
MQ_CANNOT_UNSUBSCRIBE_ACTIVE_CONSUMER	X	Cannot unsubscribe an active consumer.

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_CLIENTID_IN_USE	X	Client id already in use
MQ_CONCURRENT_ACCESS	X	Concurrent access
MQ_CONCURRENT_DEADLOCK	X	Operation may cause deadlock
MQ_CONCURRENT_NOT_OWNER		Concurrent access not owner
MQ_CONNECTION_CREATE_SESSION_ERROR		Connection failed to create a session.
MQ_CONNECTION_OPEN_ERROR		Connection failed to open a connection.
MQ_CONNECTION_START_ERROR		Connection start failed.
MQ_CONNECTION_UNSUPPORTED_TRANSPORT	X	The transport specified is not supported.
MQ_CONSUMER_CLOSED	X	The consumer was closed.
MQ_CONSUMER_EXCEPTION		An exception occurred on the consumer.
MQ_CONSUMER_NO_DURABLE_NAME	X	There is no durable name specified
MQ_CONSUMER_NO_SESSION		The consumer has no session.
MQ_CONSUMER_NOT_FOUND	X	Message consumer not found
MQ_CONSUMER_NOT_IN_SESSION	X	The consumer is not part of this session.
MQ_CONSUMER_NOT_INITIALIZED		The consumer has not been initialized.
MQ_COULD_NOT_CONNECT_TO_BROKER	X	Could not connect to Broker
MQ_COULD_NOT_CREATE_THREAD	X	Could not create thread
MQ_DESTINATION_CONSUMER_LIMIT_EXCEEDED	X	The number of consumers on the destination exceeded limit.
MQ_DESTINATION_NO_CLASS		The message does not have a destination class
MQ_DESTINATION_NO_NAME		The message does not have a destination name.
MQ_DESTINATION_NOT_TEMPORARY		The destination is not temporary
MQ_END_OF_STREAM		End of stream
MQ_FILE_NOT_FOUND		The property file could not be found
MQ_FILE_OUTPUT_ERROR		File output error
MQ_HANDLED_OBJECT_IN_USE		The object could not be deleted because there is another reference to it.
MQ_HANDLED_OBJECT_INVALID_HANDLE_ERROR		The object is invalid (i.e. it has not been deleted).

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_HANDLED_OBJECT_NO_MORE_HANDLES		A handle could not be allocated because the supply of handles has been exhausted.
MQ_HASH_TABLE_ALLOCATION_FAILED		The hash table could not be allocated
MQ_HASH_VALUE_ALREADY_EXISTS	X	The hash value already exists in the hash table.
MQ_INCOMPATIBLE_LIBRARY	X	The library is incompatible
MQ_INPUT_STREAM_ERROR		Input stream error
MQ_INTERNAL_ERROR		Generic internal error
MQ_INVALID_ACKNOWLEDGE_MODE	X	Invalid acknowledge mode
MQ_INVALID_AUTHENTICATE_REQUEST		Invalid authenticate request
MQ_INVALID_CLIENTID	X	Invalid client id
MQ_INVALID_CONSUMER_ID		Invalid consumer id
MQ_INVALID_DELIVERY_MODE	X	Invalid delivery mode.
MQ_INVALID_DESTINATION_TYPE	X	Invalid destination type.
MQ_INVALID_ITERATOR		Invalid iterator
MQ_INVALID_MESSAGE_SELECTOR	X	Invalid message selector.
MQ_INVALID_PACKET		Invalid packet
MQ_INVALID_PACKET_FIELD		Invalid packet field
MQ_INVALID_PORT		Invalid port
MQ_INVALID_PRIORITY	X	Invalid priority
MQ_INVALID_RECEIVE_MODE	X	Invalid receive mode.
MQ_INVALID_TRANSACTION_ID		Invalid transaction id
MQ_INVALID_TYPE_CONVERSION	X	The object could not be converted invalid input
MQ_MD5_HASH_FAILURE		MD5 Hash failure
MQ_MESSAGE_NO_DESTINATION		The message does not have a destination
MQ_MESSAGE_NOT_IN_SESSION	X	The message was not delivered to the session.
MQ_NEGATIVE_AMOUNT		Negative amount
MQ_NO_AUTHENTICATION_HANDLER		No authentication handler

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_NO_CONNECTION		The Session's connection has been closed
MQ_NO_MESSAGE	X	There was no message to receive.
MQ_NO_MESSAGE_PROPERTIES	X	There are no message properties
MQ_NO_REPLY_TO_DESTINATION	X	The message does not have a reply to destination.
MQ_NOT_ASYNC_RECEIVE_MODE	X	Session not in async receive mode.
MQ_NOT_FOUND	X	Not found
MQ_NOT_IPV4_ADDRESS		Not an IPv4 Address
MQ_NOT_SYNC_RECEIVE_MODE	X	Session not in sync receive mode.
MQ_NOT_TRANSACTED_SESSION	X	Session is not transacted.
MQ_NULL_PTR_ARG	X	NULL pointer passed to method
MQ_NULL_STRING		The string is NULL
MQ_NUMBER_NOT_UINT16		Number not a UINT16
MQ_OBJECT_NOT_CLONABLE		The object cannot be cloned
MQ_OUT_OF_MEMORY	X	Out of memory
MQ_PACKET_OUTPUT_ERROR		Packet output error
MQ_POLL_ERROR		Poll error
MQ_PORTMAPPER_ERROR		Portmapper error
MQ_PORTMAPPER_INVALID_INPUT		Portmapper returned invalid.
MQ_PORTMAPPER_WRONG_VERSION		Portmapper is the wrong version
MQ_PRODUCER_CLOSED	X	Producer closed.
MQ_PRODUCER_HAS_DESTINATION	X	The producer has a specified destination
MQ_PRODUCER_NO_DESTINATION	X	The producer does not have a specified destination.
MQ_PRODUCER_NOT_IN_SESSION	X	The producer is not part of this session
MQ_PROPERTY_FILE_ERROR		There was an error reading from the property file
MQ_PROPERTY_NULL		Property is NULL.
MQ_PROPERTY_WRONG_VALUE_TYPE	X	Property has the wrong value type

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_PROTOCOL_HANDLER_AUTHENTICATE_FAILED		Authenticating to the broker failed.
MQ_PROTOCOL_HANDLER_DELETE_DESTINATION_FAILED		Deleting destination failed
MQ_PROTOCOL_HANDLER_ERROR		Protocol Handler error
MQ_PROTOCOL_HANDLER_GOODBYE_FAILED		Error in saying goodbye to broker.
MQ_PROTOCOL_HANDLER_HELLO_FAILED		Error saying hello to the broker.
MQ_PROTOCOL_HANDLER_READ_ERROR		Reading a packet from the broker failed.
MQ_PROTOCOL_HANDLER_RESUME_FLOW_FAILED		Error resume flow from broker.
MQ_PROTOCOL_HANDLER_SET_CLIENTID_FAILED		Setting client id failed.
MQ_PROTOCOL_HANDLER_START_FAILED		Starting broker connection failed.
MQ_PROTOCOL_HANDLER_STOP_FAILED		Stopping broker connection failed.
MQ_PROTOCOL_HANDLER_UNEXPECTED_REPLY		Received an unexpected reply from the broker.
MQ_PROTOCOL_HANDLER_WRITE_ERROR		Writing a packet to the broker failed.
MQ_QUEUE_CONSUMER_CANNOT_BE_DURABLE	X	A queue consumer cannot be durable
MQ_READ_CHANNEL_DISPATCH_ERROR		Read channel couldn't dispatch packet.
MQ_READQTABLE_ERROR		ReadQTable error
MQ_RECEIVE_QUEUE_CLOSED		The receive queue is closed.
MQ_RECEIVE_QUEUE_ERROR		The Session is not associated with a connection.
MQ_REFERENCED_FREED_OBJECT_ERROR		A freed object was referenced.
MQ_REUSED_CONSUMER_ID		Reused consumer id
MQ_SEND_NOT_FOUND	X	The destination to which this message was sent could not be found.
MQ_SEND_RESOURCE_FULL	X	The destination is full and is rejecting new messages.
MQ_SEND_TOO_LARGE	X	The message exceeds the single message size limit for the server or for the destination.
MQ_SERIALIZE_BAD_CLASS_UID		Serialize bad class UID
MQ_SERIALIZE_BAD_HANDLE		Serialize bad handle
MQ_SERIALIZE_BAD_MAGIC_NUMBER		Serialize bad magic number

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_SERIALIZE_BAD_SUPER_CLASS		Serialize bad super class
MQ_SERIALIZE_BAD_VERSION		Serialize bad version
MQ_SERIALIZE_CANNOT_CLONE		Serialize cannot clone
MQ_SERIALIZE_CORRUPTED_HASHTABLE		Serialize corrupted hashtable
MQ_SERIALIZE_NO_CLASS_DESC		Serialize no class description
MQ_SERIALIZE_NOT_CLASS_DEF		Serialize not class definition
MQ_SERIALIZE_NOT_CLASS_HANDLE		Serialize not a class object
MQ_SERIALIZE_NOT_HASHTABLE		Serialize not a hashtable
MQ_SERIALIZE_NOT_OBJECT_HANDLE		Serialize not a handle object
MQ_SERIALIZE_STRING_CONTAINS_NULL		Serialize string contains NULL
MQ_SERIALIZE_STRING_TOO_BIG		Serialize string too big
MQ_SERIALIZE_TEST_ERROR		Serialize testing error
MQ_SERIALIZE_UNEXPECTED_BYTES		Serialize unexpected bytes
MQ_SERIALIZE_UNRECOGNIZED_CLASS		Serialize unrecognized class
MQ_SESSION_CLOSED	X	Session closed
MQ_SESSION_NOT_CLIENT_ACK_MODE	X	Session is not in client acknowledge mode
MQ_SOCKET_CLOSE_FAILED		Could not close the socket
MQ_SOCKET_CONNECT_FAILED		Could not connect socket to the host
MQ_SOCKET_ERROR		Socket error
MQ_SOCKET_READ_FAILED		Could not read from the socket
MQ_SOCKET_SHUTDOWN_FAILED		Could not shutdown socket
MQ_SOCKET_WRITE_FAILED		Could not write to the socket
MQ_SSL_ALREADY_INITIALIZED	X	SSL has already been initialized
MQ_SSL_CERT_ERROR		SSL certification error
MQ_SSL_ERROR		SSL error
MQ_SSL_INIT_ERROR		SSL initialization error
MQ_SSL_NOT_INITIALIZED	X	SSL not initialized
MQ_SSL_SOCKET_INIT_ERROR		SSL socket initialization error

Table A-1 Message Queue C Client Error Codes (*Continued*)

Code	Common	Description
MQ_STATUS_CONNECTION_NOT_CLOSED	X	The connection cannot be deleted because it was not closed.
MQ_STATUS_INVALID_HANDLE	X	The handle is invalid
MQ_STRING_NOT_NUMBER		String not a number
MQ_SUCCESS	X	Success
MQ_TCP_ALREADY_CONNECTED		TCP already connected.
MQ_TCP_CONNECTION_CLOSED		TCP connection is closed.
MQ_TCP_INVALID_PORT		Invalid TCP port.
MQ_TEMPORARY_DESTINATION_NOT_IN_CONNECTION	X	The temporary destination is not in the connection.
MQ_TIMEOUT_EXPIRED	X	Timeout expired
MQ_TRANSACTED_SESSION	X	Session is transacted.
MQ_TRANSACTION_ID_IN_USE		Transaction id in use.
MQ_TYPE_CONVERSION_OUT_OF_BOUNDS		The object conversion failed because the value is out of bounds
MQ_UNEXPECTED_ACKNOWLEDGEMENT		Received an unexpected acknowledgement
MQ_UNEXPECTED_NULL		Unexpected null
MQ_UNINITIALIZED_STREAM		Uninitialized stream
MQ_UNRECOGNIZED_PACKET_TYPE		The packet type was unrecognized
MQ_UNSUPPORTED_ARGUMENT_VALUE		Unsupported argument value
MQ_UNSUPPORTED_AUTH_TYPE		Unsupported authentication type
MQ_UNSUPPORTED_MESSAGE_TYPE		The JMS message type is not supported
MQ_VECTOR_TOO_BIG		Vector too big
MQ_WRONG_ARG_BUFFER_SIZE		Buffer is the wrong size

Index

A

acknowledgements
 data type for 74
 periodic 81

B

broker
 acknowledging consumed messages 78
 acknowledging sent messages 78
 certificate for 79
 control messages 78, 79
 host port for 77
 name for 77
 security 77, 79
broker acknowledgements
 automatic 81

C

C API
 header files 19
 runtime library 19
checklist for client deployment 24
client acknowledgements 88
 explicit 81
client identifier (ClientID) 102

connection properties
 iterating through 148
 type of 140
connections
 closing 90
 creating 33, 101
 creating properties for 29, 113
 exceptions 75, 83
 freeing 54, 118
 freeing properties of 54, 121
 handle to 74
 orderly shutdown 38
 properties of 75, 76
 secure, initializing 145
 specifying 34
 starting 179
 stopping 181
 timed out limit 77
 transport protocol for 77
ConstMQString type 74
consumers
 asynchronous 51
 closing 91
 creating asynchronous 98
 creating asynchronous durable 95
 creating durable 106
 creating synchronous 109
 handle to 74
 ping interval 77
 synchronous 50, 51, 149, 151, 153
 type of 76
 unsubscribing durable 182
 working with 50

D

- dead message queue [63](#)
- delivery modes [68](#)
 - data type for [74](#)
- deployment checklist for client applications [24](#)
- destinations
 - creating [42, 104](#)
 - creating temporary [116](#)
 - freeing [119](#)
 - getting type of [126](#)
 - handle to [74](#)
 - type of [75](#)
- directory variables
 - IMQ_HOME [14](#)
- distributed applications and synchronous consumers [51, 149](#)
- durable subscriptions
 - performance impact of [70](#)

E

- environment variables, *See* directory variables
- error handling
 - error trace [127](#)
 - error type [75](#)
 - getting status code [141](#)
 - MQStatus type [76](#)
 - status string [142](#)
- exceptions
 - listener for [75](#)

F

- fixed integer type support [21](#)
- FLOW_CONTROL property [62](#)

H

- header files [19, 183](#)

I

- IMQ_HOME directory variable [14](#)

J

- JMS clients
 - deployment checklist [24](#)
 - factors impacting performance [67](#)
 - programming model [19](#)
 - requirements for deployment [24](#)
 - setup summary [26](#)
- JMS specification [16](#)
- JMS_SUN_DMQ_BODY_TRUNCATED property [66](#)
- JMS_SUN_DMQ_PRODUCING_BROKER property [66](#)
- JMS_SUN_DMQ_UNDELIVERED_COMMENTS property [66](#)
- JMS_SUN_DMQ_UNDELIVERED_EXCEPTION property [66](#)
- JMS_SUN_DMQ_UNDELIVERED_REASON property [66](#)
- JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP property [65](#)
- JMS_SUN_LOG_DEAD_MESSAGES property [64](#)
- JMS_SUN_PRESERVE_UNDELIVERED property [64](#)
- JMS_SUN_TRUNCATE_MSG_BODY property [65](#)
- JMSCorrelationID message header field [45](#)
- JMSDeliveryMode message header field [45](#)
- JMSDestination message header field [45](#)
- JMSExpiration message header field [45](#)
- JMSMessageID message header field [45](#)
- JMSPriority message header field [45](#)

JMSRedelivered message header field 45
 JMSReplyTo message header field 45
 JMSTimestamp message header field 45
 JMSType message header field 45
 JMSXDeliveryCount property 65

L

listeners, message
 data type for 75, 81
 logging 55

M

memory management 54
 message acknowledgements 40
 message consumption
 asynchronous 81
 message headers
 getting 134
 properties 48
 setting 173
 message properties
 default values for 48
 getting 30, 136
 handle to 75
 iterating through 31, 148
 setting 175
 type of 140
 Message Queue
 fixed integer type support 21
 header files 183
 meta data for 139
 name of 79
 version of 79
 Message Queue programs, building 19
 message selector 58

messages
 acknowledging 88
 body 46
 body type, and performance 72
 composing 44
 correlation id 134
 creating bytes type 100
 creating text type 117
 expiration of 134
 filtering 52
 freeing 120
 getting text of 144
 getting type of 138
 handle to 75
 limit of unconsumed 79
 mode of 134
 ordering of 59
 prioritizing 59
 priority of 134
 processing 52
 receiving 49
 redelivered status 134
 reply-to destination 137, 176
 selector for 95, 98, 106
 selectors 58
 sending 47, 157, 159, 161, 163
 set text of 178
 size, and performance 71
 type of 75, 134
 messages properties
 creating 113
 freeing 121
 mq.sys.dmqueue 63
 MQ_ACK_ON_ACKNOWLEDGE_PROPERTY 41,
 78
 MQ_ACK_ON_PRODUCE_PROPERTY 78
 MQ_ACK_TIMEOUT_PROPERTY 34, 77
 MQ_AUTO_ACKNOWLEDGE enum 81
 MQ_BOOL_TYPE property 76
 MQ_BROKER_NAME_PROPERTY 77
 MQ_BROKER_PORT_PROPERTY 77
 MQ_BYTES_MESSAGE message type 75

- MQ_CLIENT_ACKNOWLEDGE enum 81
- MQ_CONNECTION_FLOW_COUNT_PROPERTY 35, 78
- MQ_CONNECTION_FLOW_LIMIT_ENABLED_PROPERTY 35
- MQ_CONNECTION_FLOW_LIMIT_ENABLED_PROPERTY 79
- MQ_CONNECTION_FLOW_LIMIT_PROPERTY 35, 79
- MQ_CONNECTION_TYPE_PROPERTY 77
- MQ_CORRELATION_ID_HEADER_PROPERTY 134
- MQ_DUPS_OK_ACKNOWLEDGE enum 81
- MQ_EXPIRATION_HEADER_PROPERTY 48, 134
- MQ_FLOAT32_TYPE property 76
- MQ_FLOAT64_TYPE property 76
- MQ_INT16_TYPE property 76
- MQ_INT32_TYPE property 76
- MQ_INT64_TYPE property 76
- MQ_INT8_TYPE property 76
- MQ_INVALID_TYPE property 76
- MQ_LOG_FILE 55
- MQ_LOG_LEVEL 55
- MQ_MESSAGE_ID_HEADER_PROPERTY 134
- MQ_MESSAGE_TYPE_HEADER_PROPERTY 134
- MQ_NAME_PROPERTY 79
- MQ_PERSISTENT_HEADER_PROPERTY 48, 134
- MQ_PING_INTERVAL_PROPERTY 34, 77
- MQ_PRIORITY_HEADER_PROPERTY 48, 134
- MQ_REDELIVERED_HEADER_PROPERTY 134
- MQ_SESSION_ASYNC_RECEIVE 41
- MQ_SESSION_ASYNC_RECEIVE consumer type 76
- MQ_SESSION_SYNC_RECEIVE 41
- MQ_SESSION_SYNC_RECEIVE consumer type 76
- MQ_SESSION_TRANSACTED enum 81
- MQ_SSL_BROKER_CERT_FINGERPRINT 37
- MQ_SSL_BROKER_IS_TRUSTED 77, 79
- MQ_SSL_CHECK_BROKER_FINGERPRINT 37, 79
- MQ_STRING_TYPE property 76
- MQ_TEXT_MESSAGE message type 75
- MQ_TIMESTAMP_HEADER_PROPERTY 134
- MQ_UNSUPPORTED_MESSAGE message type 75
- MQ_VERSION_PROPERTY 79
- MQAckMode type 74
- MQAcknowledgeMessages function 88
- MQBool type 74
- MQChar type 74
- MQCloseConnection function 90
- MQCloseMessageConsumer function 91
- MQCloseMessageProducer function 92
- MQCloseSession function 93
- MQCommitSession function 94
- MQConnectionExceptionListenerFunc type 75, 83
- MQConnectionHandle type 74
- MQConsumerHandle type 74
- MQCreateAsyncDurableMessageConsumer function 95
- MQCreateAsyncMessageConsumer function 98
- MQCreateBytesMessage function 100
- MQCreateConnection function 101
- MQCreateDestination function 104
- MQCreateDurableMessageConsumer function 106
- MQCreateMessageConsumer function 109
- MQCreateMessageProducer function 111
- MQCreateMessageProducerForDestination function 112
- MQCreateProperties function 113
- MQCreateSession function 114
- MQCreateTemporaryDestination function 116
- MQCreateTextMessage function 117
- mqcrt library 21
- mqcrt runtime library, 64-bit support 19
- MQDeliveryMode type 74
- MQDestinationHandle type 74
- MQDestinationType type 75
- MQError type 75
- MQFloat64 type 75
- MQFreeConnection function 118
- MQFreeDestination function 119
- MQFreeMessage function 120
- MQFreeProperties function 121
- MQFreeString function 122
- MQGetAcknowledgeMode function 123

- MQGetBoolProperty function 124
- MQGetBytesMessageBytes function 125
- MQGetDestinationType function 126
- MQGetErrorTrace function 127
- MQGetFloat32Property function 128
- MQGetFloat64Property function 129
- MQGetInt16Property function 130
- MQGetInt32Property function 131
- MQGetint64Property function 132
- MQGetInt8Property function 133
- MQGetMessageHeaders function 134
- MQGetMessageProperties function 136
- MQGetMessageReplyTo function 137
- MQGetMessageType function 138
- MQGetMetaData function 139
- MQGetPropertyType function 140
- MQGetStatusCode function 141
- MQGetStatusString function 142
- MQGetStringProperty function 143
- MQGetTextMessageText function 144
- MQInitializeSSL function 145
- MQInt32 type 75
- MQInt8 type 75
- MQMessageHandle type 75
- MQMessageListenerFunc type 75, 81
- MQMessageType type 75
- MQProducerHandle type 75
- MQPropertiesHandle type 75
- MQPropertiesKeyIterationGetNext function 146
- MQPropertiesKeyIterationHasNext function 147
- MQPropertiesKeyIterationStart function 148
- MQReceiveMessageNoWait function 149
- MQReceiveMessageWait function 151
- MQReceiveMessageWithTimeout function 153
- MQReceiveMode type 76
- MQRecoverSession function 155
- MQRollBackSession function 156
- MQSendMessage function 157
- MQSendMessageExt function 159
- MQSendMessageToDestination function 161
- MQSendMessageToDestinationExt function 163
- MQSessionHandle type 76
- MQSetBoolProperty function 165
- MQSetBytesMessageBytes function 166
- MQSetFloat32Property function 167
- MQSetFloat64Property function 168
- MQSetInt16Property function 169
- MQSetInt32Property function 170
- MQSetInt64Property function 171
- MQSetInt8Property function 172
- MQSetMessageHeaders function 173
- MQSetMessageProperties function 175
- MQSetMessageReplyTo function 176
- MQSetStringProperty function 177
- MQSetTextMessageText function 178
- MQStartConnection function 179
- MQStatus type 76
- MQStatusIsError function 180
- MQStopConnection function 181
- MQString type 76
- MQType type 76
- MQUnsubscribeDurableMessageConsumer function 182

N

- NSPR library 21
- NSS library 21

P

- performance
 - factors impacting, *See* performance impact factors
- performance and reliability 67
- performance impact factors
 - acknowledgement mode 69
 - delivery mode 68
 - durable subscriptions 70
 - message body type 72
 - message size 71
 - selectors 71
 - transactions 69

physical destination properties [62](#)

ping interval [34](#), [77](#)

producers

 closing [92](#)

 creating [111](#)

 creating for destination [112](#)

 handle to [75](#)

 ping interval [77](#)

programming examples

 build instructions [22](#)

 running [23](#)

R

REJECT_NEWEST property [62](#)

reliability and performance [67](#)

REMOVE_LOW_PRIORITY property [62](#)

REMOVE_OLDEST property [62](#)

runtime library

 64-bit support [19](#)

S

sample programs

 compiler options for [21](#)

 running [23](#)

secure connections [35](#)

selectors [58](#), [71](#)

sessions

 acknowledge mode of [123](#)

 closing [93](#)

 committing [94](#)

 creating [40](#), [114](#)

 handle to [76](#)

 managing [42](#)

 recovering [155](#)

 rolling back [156](#)

 transacted [40](#), [81](#)

T

thread management [60](#)

transactions

 committing [94](#)

 performance impact of [69](#)

 working with [40](#)