Sun Java™ System

Message Queue 3.5
C Client Developer's Guide

Service Pack 1

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

Part No: 817-6025-10

Contents

List of Figures 7

List of Tables 9

List of Procedures 11

Preface 13
Audience for This Guide 13
Organization of This Guide 14
Conventions 14
Text Conventions 14
Directory Variable Conventions 15
Other Documentation Resources 16
The Message Queue Documentation Set 16
Online Help 17
Example Client Applications 17
The Java Message Service (JMS) Specification 17
Related Third-Party Web Site References 18

Chapter 1 Introduction 19
What Is Message Queue? 19
Message Queue Features 20
Java and C Interfaces 21
Product Editions 22
Message Queue Messaging System Architecture 22

The JMS Programming Model 24
Message 24
Header 25
Properties 26
Message Body Types 26
Destination 27
Connection 27
Session 27
Message Producer 28
Message Consumer 28
Message Listener 28
Client Design Issues 29
Programming Domains 29
Client Identifiers 30
Reliable Messaging 30
Delivery Mode 30
Acknowledgements and Transactions 31
Persistent Storage 32
Performance Trade-offs 32
Message Production and Consumption 33
Message Production 34
Message Consumption 34
Synchronous and Asynchronous Consumption 36
Message Selection 37
Message Order and Priority 37
Configuring Connections 38
Connection Handling 38
Reliability 39
Flow Control 39
Security 40
Version Information 40
Managing Flow Control 40
Delivery Mode 40
Acknowledgement Mode 41
Message Flow Metering 42

Chapter 2 Building and Running Message Queue C Clients 43
Getting Ready 43
Building Programs 43
Providing Runtime Support 45
Working With the Sample C-Client Programs 45
Building the Sample Programs 45
To Compile and Link on Solaris 45
To Compile and Link on Linux 46
To Compile on Windows 46
To Link on Windows 46
Running the Sample Programs 46

Chapter 3 Using the C API 47
Message Queue C Client Setup Operations 48
To Set Up a Message Queue C Client to Produce Messages 48
To Set Up a Message Queue C Client to Consume Messages Synchronously 49
To Set Up a Message Queue C Client to Consume Messages Asynchronously 49
Working With Properties 50
Setting Connection and Message Properties 50
To Set Properties for a Connection 51
To Set Message Properties 52
Getting Message Properties 52
To Iterate Through a Properties Handle 53
Working With Connections 54
Working With Secure Connections 56
Shutting Down Connections 57
Working With Sessions and Destinations 57
Creating a Session 58
Transacted Sessions 58
Message Acknowledgement 58
Receive Mode 59
Managing a Session 59
Creating Destinations 59

Working With Messages 61
Composing Messages 61
Sending a Message 62
Receiving Messages 64
Working With Consumers 65
Receiving a Message Synchronously 66
Receiving a Message Asynchronously 66
Processing a Message 67
Error Handling 68
To Handle Errors in Your Code 68
Memory Management 69
Thread Management 70
Message Queue C Runtime Thread Model 70
Concurrent Use of Handles 70
Single-Threaded Session Control 71
Connection Exceptions 72
Logging 72

Chapter 4 Reference 73
Data Types 73
Connection Properties 76
Acknowledge Modes 80
Callback Type for Asynchronous Messaging 81
Callback Type for Connection Exception Handling 82
Function Reference 83
Header Files 183

Appendix A Message Queue C API Error Codes 185
Error Codes 186

Index 193

List of Figures

Figure 1-1 Message Queue System Architecture 23
Figure 1-2 JMS Programming Objects 25
Figure 1-3 Messaging Operations 33
Figure 1-4 Message Delivery to Message Queue Client Runtime 35

List of Tables

Table 1 Book Contents 14
Table 2 Document Conventions 14
Table 3 Message Queue Directory Variable Used by C Clients 15
Table 4 Message Queue Documentation Set 16
Table 1-1 JMS-defined Message Header 25
Table 1-2 C-API Message Body Types 27
Table 2-1 Locations of C-API Libraries and Header Files 44
Table 2-2 Preprocessor Definitions for Supporting Fixed-Size Integer Types 44
Table 3-1 Functions Used to Set Properties 50
Table 3-2 Functions Used to Get Message Properties 53
Table 3-3 Functions Used to Work with Connections 54
Table 3-4 Functions Used to Work with Sessions 57
Table 3-5 Functions Used to Work with Destinations 59
Table 3-6 Functions Used to Construct Messages 61
Table 3-7 Functions for Sending Messages 63
Table 3-8 Functions Used to Receive Messages 64
Table 3-9 Functions Used to Process Messages 67
Table 3-10 Functions Used in Handling Errors 68
Table 3-11 Functions Used to Free Memory 69
Table 3-12 Thread Model for NSPR GLOBAL Threads 70
Table 3-13 Handles and Concurrency 70
Table 4-1 Message Queue C-API Data Type Summary 74
Table 4-2 Connection Properties 77
Table 4-3 `acknowledgeMode` Values 80

Table 4-4 Message Queue C-API Function Summary 83
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 45
To Compile and Link on Linux 46
To Compile on Windows 46
To Link on Windows 46
To Set Up a Message Queue C Client to Produce Messages 48
To Set Up a Message Queue C Client to Consume Messages Synchronously 49
To Set Up a Message Queue C Client to Consume Messages Asynchronously 49
To Set Properties for a Connection 51
To Set Message Properties 52
To Iterate Through a Properties Handle 53
To Handle Errors in Your Code 68

Preface

This book provides programming and reference information for developers working with Sun Java™ System Message Queue (formerly Sun™ ONE Message Queue) 3.5 SP1, 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:

- Audience for This Guide
- Organization of This Guide
- Conventions
- Other Documentation Resources

Audience for This Guide

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

Organization of This Guide

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 basic concepts, operations, and architecture of Message Queue messaging. Contains some material on programming and configuration issues to improve performance and throughput.
Chapter 2, “Building and Running Message Queue C Clients”	Explains how to compile and link 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.
Chapter 3, “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 and thread handling.
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

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

Text Conventions

Table 2 Document Conventions

Format	Description
italics	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.

Table 2 Document Conventions *(Continued)*

Format	Description
`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 URL's.
`[]`	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
`IMQ_HOME`	This is generally used in Message Queue documentation to refer to the Message Queue base directory (root installation directory): • On Solaris, there is no root Message Queue installation directory. Therefore, `IMQ_HOME` 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 `C:\Program Files\Sun\MessageQueue3`). • On Linux, there is no root Message Queue installation directory. Therefore, `IMQ_HOME` is not used in Message Queue documentation to refer to file locations on Linux.

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 (/).

Other Documentation Resources

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
Message Queue Installation Guide	Developers and administrators	Explains how to install Message Queue software on Solaris, Linux, and Windows platforms.
Message Queue Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
Message Queue Java Client Developer's Guide	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using the Message Queue implementation of the JMS or SOAP/JAXM APIs.
Message Queue Administration Guide	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using Message Queue administration tools.
Message Queue C Client Developer's Guide	Developers	Provides programming and reference documentation for developers of Message Queue C client programs.

Online Help

Message Queue 3.5 SP1 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 3.5 SP1 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

A number of example applications that provide sample client application code are included in a directory that depends upon the operating system (see the *Message Queue Administration Guide*).

Sample applications that illustrate the C-API are listed and described in Chapter 2, "Building and Running Message Queue C Clients" on page 43.

See the README file located in that directory for guidance on how to run the sample programs.

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.

Chapter 1

Introduction

This chapter provides an overall introduction to Sun Java™ System Message Queue 3.5 SP1 (formerly Sun™ ONE Message Queue) and to JMS concepts and programming issues of interest to developers. It is written specifically for the C developer who wants to interface with a Message Queue Message Service in order to send messages to and receive messages from another Message Queue client. Message Queue clients can reside on the same or on different platforms. The chapter covers the following topics:

- "What Is Message Queue?" on page 19
- "Message Queue Messaging System Architecture" on page 22
- "The JMS Programming Model" on page 24
- "Client Design Issues" on page 29
- "Configuring Connections" on page 38
- "Managing Flow Control" on page 40

What Is Message Queue?

The Message Queue product is a standards-based solution to the problem of reliable, asynchronous messaging for distributed applications. Message Queue is an enterprise messaging system that implements the Java™ Message Service (JMS) open standard: it is a JMS provider.

With Message Queue software, processes running on different platforms and operating systems can connect to a common Message Queue message server (broker) to send and receive information. Application developers are free to focus on the business logic of their applications, rather than on the low-level details of how their applications communicate across a network. 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 Java Client Developer's Guide*, to send messages to and receive messages from a Message Queue broker.

Message Queue administrators can use a variety of tools to set up destinations on the broker and to configure the broker in response to performance and reliability requirements. Administrative functions and tools are described in the *Message Queue Administration Guide*.

Message Queue Features

Message Queue has features that exceed the minimum requirements of the JMS specification. Among these features are the following:

Centralized administration Provides both command-line and GUI tools for administering a Message Queue message service and managing application-specific aspects of messaging, such as destinations and security.

Scalable message service Allows you to service increasing numbers of JMS clients (components or applications) by balancing the load among a number of Message Queue message service components (*brokers*) working in tandem (*multi-broker cluster*).

Tunable performance Lets you increase performance of the Message Queue message service when less reliability of delivery is acceptable.

Multiple transports Supports the ability of JMS clients to communicate with each other over a number of different transports and using secure (SSL) connections.

C-API Allows you to integrate legacy systems into a Message Queue messaging system and to create light-weight clients that do not require an underlying JVM.

See the *Message Queue Release Notes* for documentation of JMS compliance-related issues.

Java and C Interfaces

While this manual revisits a number of topics presented in the *Message Queue Java Client Developer's Guide*, there are differences between the two interfaces and the JMS features they support. Some of these differences are summarized below, but this list is not exhaustive. If you plan to write a Message Queue C client, you should read this manual in full.

The C interface, compared to the Java interface

- Does not support the use of administered objects.
- Supports only two message types (text and bytes); it does not support map, stream, or object message types.
- Does not support consumer-based flow control
- Does not support queue browsers.
- Does not support JMS application server facilities (ConnectionConsumer, distributed transactions).
- Does not support distributed transactions.
- Does not support receiving SOAP messages sent by a Message Queue Java client.

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

Product Editions

The Message Queue product is available in two editions: Platform and Enterprise—each corresponding to a different licensed capacity. The C-API is only supported on the Enterprise Edition. For more information about these editions and for instructions on how you upgrade Message Queue from one edition to another, see the the *Message Queue Installation Guide*.

Message Queue Messaging System Architecture

This section briefly describes the main parts of a Message Queue messaging system. While as a developer, you do not need to be familiar with the details of all of these parts or how they interact, a high-level understanding of the basic architecture will help you understand features of the system that impact Message Queue C client design and development.

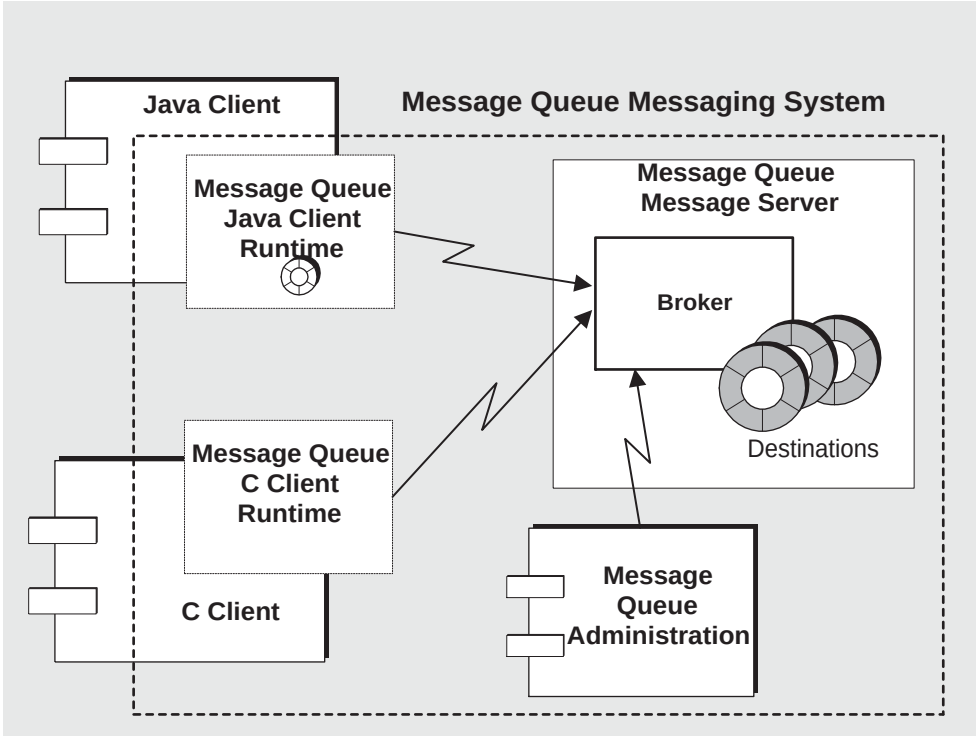
The main parts of a Message Queue messaging system, shown in Figure 1-1, are the following:

Message Queue Client A Message Queue client can be written in C or Java, and it can send and/or receive Message Queue messages.

Message Queue message server The Message Queue message server is the heart of a messaging system. It consists of a broker that provides delivery services for the system. These services include connections to C or Java clients, message routing and delivery, persistence, security, and logging. The message server also maintains physical destinations to which clients send messages, and from which the messages are delivered to consuming clients. The Message Queue message server is described in detail in the *Message Queue Administration Guide*.

Message Queue client runtime The Message Queue C and Java client runtimes provide Message Queue C and Java clients respectively with an interface to the Message Queue message server. They support all operations needed for clients to send messages to destinations and to receive messages from such destinations. The Message Queue C client runtime is described in detail in "Message Production and Consumption" on page 33.

Figure 1-1 Message Queue System Architecture



Message Queue message service The Message Queue message service includes one or more Message Queue servers and Message Queue client runtime support.

Message QueueAdministration Message Queue provides a number of administrative tools for managing a Message Queue messaging system. These tools are used to manage the message server, security, messaging application resources, and persistent data. These tools are generally used by Message Queue administrators and are described in the *Message Queue Administration Guide*.

Message Queue Messaging System The Message Queue messaging system includes the Message Queue message service and Message Queue administration.

The JMS Programming Model

This section briefly describes the programming model of the JMS specification. The JMS programming model is the foundation for the design of a Message Queue C client. Although the C-API does not provide an exhaustive implementation of the JMS programming model, this section is provided as a review of the most important concepts and terminology (defined for that model), which also apply to Message Queue C client design.

In the JMS programming model, JMS clients (components or applications) interact using a JMS application programming interface (API) to send and receive messages. In this context, it is important to understand that a C client's interface 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.

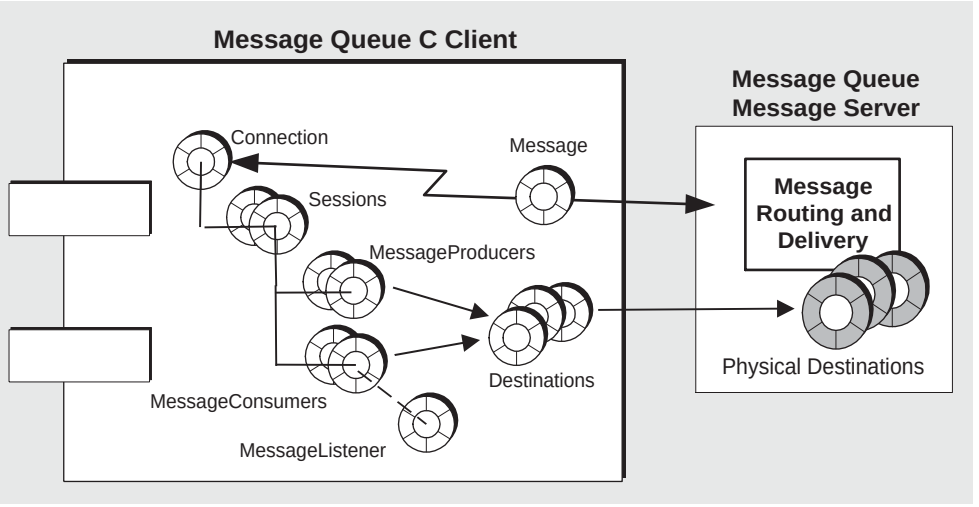
This section introduces the C data types and functions used by a Message Queue C client for delivery of messages. The main data types, which are opaque to the user and accessible only through the C functions, are shown in Figure 1-2 and described in the following sections.

Message

In the Message Queue product, data is exchanged using JMS messages—messages that conform to the JMS specification. According to the JMS specification, a message is composed of three parts: a header, properties, and a body.

Properties are optional—they provide values that clients can use to filter messages. A body is also optional—it contains the actual data to be exchanged.

Figure 1-2 JMS Programming Objects



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. The following table lists the header fields defined (and required) by JMS and their corresponding names, as defined by the C-API.

Table 1-1 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

Table 1-1 JMS-defined Message Header *(Continued)*

JMS Message Header Field	C-API Message Header Property Name
`JMSTimestamp`	MQ_TIMESTAMP_HEADER_PROPERTY
`JMSRedelivered`	MQ_REDELIVERED_HEADER_PROPERTY
`JMSCorrelationID`	MQ_CORRELATION_ID_HEADER_PROPERTY
`JMSReplyTo`	Set by the `MQSetMessageReplyTo` function, and obtained by the `MQGetMessageReplyTo` function.
`JMSType`	MQ_MESSAGE_TYPE_HEADER_PROPERTY

For additional information about each property type and the agent who sets it, see Table 4-6 on page 173.

Properties

When data is sent between two processes, other information besides the payload data can be sent with it. These descriptive fields, or *properties*, can provide additional information about the data; for example, which process created it, the time it was created, and information that uniquely identifies the structure of each piece of data. Properties (which can be thought of as an extension of the header) consist of property name and property value pairs, as specified by a C client. A C client can set message properties when initializing a handle to a properties data type and passing that handle to the `MQSetMessageProperties` function.

Having registered an interest in a particular destination, consuming clients can fine-tune their selection by specifying certain property values as selection criteria. For instance, a client might indicate an interest in Payroll messages (rather than Facilities) but only Payroll items concerning part-time employees located in New Jersey. Messages that do not meet the specified criteria are not delivered to the consumer.

Message Body Types

JMS specifies six classes (or types) of messages. The C-API supports only two of these types, as described in Table 1-2. 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 1-2.

Table 1-2 C-API Message Body Types

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

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.

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

Connection

A *connection* is a JMS client's configured connection to a Message Queue message service. Both allocation of communication resources and authentication of a client take place when a connection is created. Hence it is a relatively heavy-weight object, and most clients do all their messaging with a single connection. A connection is used to create sessions.

Session

A *session* is a single-threaded context for producing and consuming messages. While there is no restriction on the number of threads that can use a session, the session should not be used *concurrently* by multiple threads. It is used to create the message producers and consumers that send and receive messages, and defines a

serial order for the messages it consumes and the messages it produces. A session supports reliable delivery through a number of acknowledgement options or by using transactions. A transacted session can combine a series of sequential operations into a single transaction that can span a number of producers and consumers. You need to create a session before you can create its consumers or producers.

Message Producer

A client uses a *message producer* to send messages to a physical destination. You can create a message producer with a specified destination or you can specify a destination when you send each message. You can also specify a delivery mode, priority, and time-to-live for a message producer that govern all messages sent by a producer, except when explicitly over-ridden.

Message Consumer

A client uses a *message consumer* to receive messages from a physical destination. A message consumer can have a message selector that allows the message service to deliver only those messages to the consumer that match the selection criteria. A message consumer can support either synchronous or asynchronous consumption of messages (see "Synchronous and Asynchronous Consumption" on page 36).

Message Listener

To support asynchronous communication, a Message Queue C client must write a callback function of type `MQMessageListenerFunc`. You pass a pointer to this function when you create an asynchronous message consumer. A client is said to *consume* a message when a session thread invokes this callback function.

Client Design Issues

This section describes a number of messaging issues that impact Message Queue C client design.

Programming Domains

When you create a session, you can specify one of two message delivery models: point-to-point and publish/subscribe. You specify the message delivery model for a C-Message Queue client by specifying either `MQ_QUEUE_DESTINATION` or `MQ_TOPIC_DESTINATION` for the `destinationType` parameter when you call the `MQCreateDestination` function.

Point-to-Point (Queue Destinations) A message is delivered from a producer to one consumer. In this delivery model, the destination type is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, depending on the queue's delivery policy, to one of the consumers registered for the queue. Any number of producers can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only *one* consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

Publish/Subscribe (Topic destinations) A message is delivered from a producer to any number of consumers. In this delivery model, the destination type is a *topic*. Messages are first delivered to the topic destination, then delivered to *all* active consumers that have *subscribed* to the topic. Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribed consumers. Topic destinations also support the notion of *durable subscriptions.* A durable subscription represents a durable consumer that is registered with the topic destination but can be inactive at the time that messages are delivered. When the consumer subsequently becomes active, it receives the messages. If there are no consumers registered for a topic destination, the topic does not hold messages it receives, unless it has durable subscriptions for inactive consumers.

Client Identifiers

Clients need to be identified to a broker both for authentication purposes and to keep track of durable subscriptions.

For authentication purposes, you need to provide a user name and password. The administrator is responsible for setting up a user repository against which the broker can validate this name and password. See the *Message Queue Administration Guide* for more information.

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.

Client identifiers are used in combination with a durable subscription name (see "Publish/Subscribe (Topic destinations)" on page 29) to make sure that each durable subscription corresponds to only one user. 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. The only way for the broker to identify the subscriber is through its client ID. You can specify a client ID using the `clientID` parameter to the `MQCreateConnection` function.

Reliable Messaging

Reliable messaging depends on a message's delivery mode and the use of transactions or acknowledgements to ensure the reliability of persistent messages.

Delivery Mode

JMS defines two *delivery modes*: persistent and non-persistent:

- Persistent messages are guaranteed to be delivered and successfully consumed once and only once. Reliability is at a premium for such messages.
- Non-persistent messages are guaranteed to be delivered at most once. Reliability is not a major concern for such messages.

A message's delivery mode is set to be persistent by default. You can override this setting by using the `MQSendMessageExt` function and setting the delivery mode to `MQ_NONPERSISTENT_DELIVERY`.

Reliable messaging guarantees the delivery of persistent messages to and from a destination. There are two aspects of assuring reliability in the case of *persistent* messages. One is to assure that their delivery to and from a message service is successful. The other is to assure that the message service does not lose these messages before delivering them to consumers.

Acknowledgements and Transactions

You can ensure reliable messaging by using either of two general mechanisms supported by a Message Queue session: *acknowledgements* or *transactions*.

Acknowledgements

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 broker'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.

In the case of message consumers, you can specify one of several acknowledge modes for the consuming session when you create that session. 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. For more information about a session's acknowledge modes, see "Acknowledge Modes" on page 80 and the description of the `MQ_ACK_ON_ACKNOWLEDGE_PROPERTY` in Table 4-2 on page 77.

Transactions

A session can also be configured as *transacted*, in which case work spanning a session's producers or consumers is combined into an atomic unit—a *transaction*. The Message Queue-C API provides functions for committing, or rolling back a transaction. (See "Transacted Sessions" on page 58 for more information.) The C runtime does not support distributed transactions, that is a transaction cannot include operations involving other resource managers, such as database systems.

As messages are produced or consumed within a transaction, the broker tracks the various sends and receives, completing these operations only when the client issues a call to commit the transaction. If a particular send or receive operation within the transaction fails, an exception is raised. The application can handle the

exception by ignoring it, retrying the operation, 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.)

Persistent Storage

The other important aspect of reliability is assuring that once persistent messages are delivered to their destinations, the message service does not lose them before they are delivered to consumers. This means that upon delivery of a persistent message to its destination, the message service must place it in a persistent data store. If the message service goes down for any reason, it can recover the message and deliver it to the appropriate consumers. While this adds overhead to message delivery, it also adds reliability.

A message service must also store durable subscriptions. This is because to guarantee delivery in the case of topic destinations, it is not sufficient to recover only persistent messages. The message service must also recover information about durable subscriptions for a topic, otherwise it would not be able to deliver a message to durable consumers when they become active.

Messaging applications that are concerned about guaranteeing delivery of persistent messages must either employ queue destinations or employ durable subscriptions to topic destinations.

The way in which the message service handles persistent messages depends upon a session's delivery mode. For more information, see "Delivery Mode" on page 30.

Performance Trade-offs

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

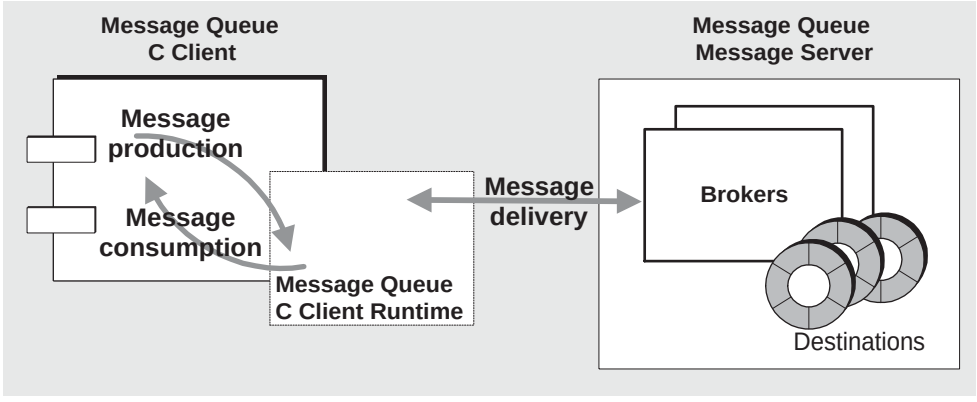
using a transacted session. Between these extremes are a number of options, depending on the needs of an application, including the use of Message Queue-specific persistence and acknowledgement properties (see "Managing Flow Control" on page 40.).

Message Production and Consumption

The Message Queue C client runtime provides Message Queue C clients with an interface to the Message Queue message server—it supplies these clients with all the data types and functions introduced in "The JMS Programming Model" on page 24. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

This section provides a high level description of how the Message Queue C client runtime supports message production and consumption. Figure 1-3 on page 33 illustrates how message production and consumption involve an interaction between clients and the Message Queue C client runtime, while message delivery involves an interaction between the Message Queue C client runtime and Message Queue message servers.

Figure 1-3 Messaging Operations



Once a client has created a connection to a broker, created a session as a single-threaded context for message delivery, and created a message producer or a message consumer to access particular destinations in a message server, production (sending) or consumption (receiving) of messages can proceed.

Message Production

In message production, a message is created by the client, and sent over a connection to a destination on a broker. If the message delivery mode has been set to persistent (guaranteed delivery, once and only once), the client thread blocks until the broker acknowledges that the message was delivered to its destination and stored in the broker's persistent data store. If the message is not persistent, no broker acknowledgement is returned by the broker, and the client thread does not block.

In the case of persistent messages, to increase throughput on sends, you can set the connection to *not* require broker acknowledgement (see "Connection Properties" on page 76), but this eliminates the guarantee that persistent messages are reliably delivered.

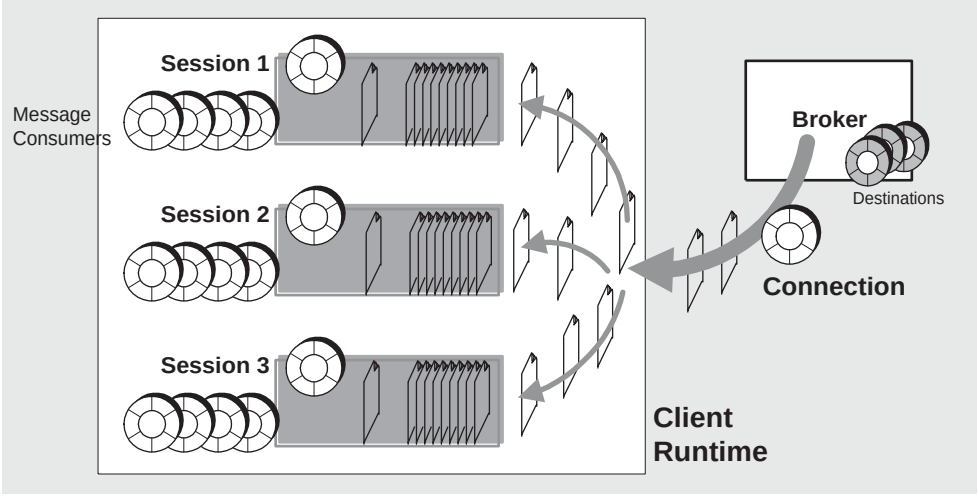
Message Consumption

Message consumption is more complex than production. Messages arriving at a destination on a broker are delivered over a connection to the Message Queue client runtime under the following conditions:

- The client has set up a consumer for the given destination.
- The selection criteria for the consumer, if any, match that of messages arriving at the given destination.
- The connection has been told to start delivery of messages.

Messages delivered over the connection are distributed to the appropriate Message Queue sessions where they are queued to be consumed by the appropriate message consumers, as shown in Figure 1-4.

Figure 1-4 Message Delivery to Message Queue Client Runtime



Messages are fetched off each session queue one at a time (a session is single threaded). A message can consumed synchronously or asynchronously. A message is said to be *consumed* either when one of the `MQReceiveMessage...` functions returns (synchronously) or when the callback function associated with the asynchronous consumer returns.

When a broker delivers messages to the client runtime, it marks the messages accordingly, but does not really know if they have been consumed. Therefore, the broker waits for the client to acknowledge receipt of a message before deleting the message from the broker's destination. If a connection fails, and another connection is subsequently established, the broker will re-deliver all previously delivered but unconsumed messages, setting their message header `MQ_REDLIEVERED_HEADER_PROPERTY` field.

There are three acknowledgment options that you can set for a client session:

- `AUTO_ACKNOWLEDGE`: the session automatically acknowledges each message consumed by the client.
- `CLIENT_ACKNOWLEDGE`: the client explicitly acknowledges after one or more messages have been consumed. This option gives the client the most control. This acknowledgement takes place by calling the `MQAcknowledgeMessages` function, causing the session to acknowledge all messages that have been consumed by the session up to that point. (This could include messages consumed asynchronously by many different message listeners in the session, independent of the *order* in which they were consumed.)

- DUPS_OK_ACKNOWLEDGE: the session acknowledges after ten messages have been consumed (this value is not currently configurable) and doesn't guarantee that messages are delivered and consumed only once. Clients use this mode if they don't care if messages are processed more than once.

Each of the three acknowledgement options requires a different level of processing and bandwidth overhead. AUTO_ACKNOWLEDGE consumes the most overhead and guarantees reliability on a message by message basis, while DUPS_OK_ACKNOWLEDGE consumes the least overhead, but allows for duplicate delivery of messages.

In the case of the AUTO_ACKNOWLEDGE or CLIENT_ACKNOWLEDGE options, the threads performing the acknowledgement, or committing a transaction, will block, waiting for the broker to return an acknowledgement of the client acknowledgement. This broker acknowledgement guarantees that the broker has deleted the corresponding persistent message and will not send it twice—which could happen were the client or broker to fail, or the connection to fail, at the wrong time.

To increase throughput, you can configure the connection to *not* require broker acknowledgement of client acknowledgements, but this eliminates the guarantee that persistent messages are delivered once and only once.

NOTE In the DUPS_OK_ACKNOWLEDGE mode, the session does not wait for broker acknowledgements. This option is 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_REDLIEVERED_HEADER_PROPERTY.

Synchronous and Asynchronous Consumption

There are two ways a Message Queue C client can consume messages: either synchronously or asynchronously.

In *synchronous consumption*, a client gets a message by calling one of the MQReceive... functions. The client thread blocks until the function returns. This means that if no message is available, the client blocks until a message does become available or until the receive function times out (if it was called with a time-out specified). In this model, a client thread can only consume messages one at a time.

In *asynchronous consumption*, a client creates a callback function of type `MQMessageListenerFunc` and passes a pointer to it as a parameter to one of the `MQCreateAsync...MessageConsumer` functions. A client consumes a message when the session invokes this function. In this model, the client thread does not block because the thread listening for and consuming the message belongs to the Message Queue client runtime.

Message Selection

JMS defines a mechanism by which a message service can perform message filtering and routing based on criteria placed in message selectors. A producing client can define application-specific properties for a message, and a consuming client can indicate its interest in messages using selection criteria based on such properties. This simplifies the work of the client and eliminates the overhead of delivering messages to clients that do not need them. However, it adds some additional overhead to the message service processing the selection criteria. Message selector syntax and semantics are outlined in the JMS specification.

Use the `MQSetMessageProperties` function to set properties that can be used in message filtering.

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, the 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.

Configuring Connections

The Message Queue client runtime supports all the operations described in "Message Production and Consumption" on page 33. It also provides connection properties that you can set to specify a broker to connect to, configure a secure connection, optimize resources, performance, and message throughput.

Connection properties can be grouped into the following categories:

- Connection Handling
- Reliability
- Flow Control
- Security
- Version Information

Each of these categories is discussed in the following sections with a description of the properties that you can set to configure the behavior of the broker. All broker properties are described in detail in Table 4-2 on page 77.

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`.

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 Production and Consumption" on page 33. 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 and flow 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.

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

Security

The C-API supports the SSL transport protocol, which supports SSL v2, SSL v3, and TLS standards. For more information on how to set up and create a secure connection, see "Working With Secure Connections" on page 56 for more information.

Version Information

Properties that specify the version of the Message Queue product are set by the C client runtime and can be read using the `MQGetMetaData` function.

Managing Flow Control

Because of the mechanisms by which messages are delivered to and from a broker, and because of the Message Queue control messages used to assure reliable delivery, there are a number of factors that affect reliability and performance. These factors include: delivery mode, acknowledgement mode, message flow metering, and message flow limits.

Although these factors are quite distinct, their interactions can complicate the task of balancing reliability with performance. Specifically, because client messages and Message Queue control messages flow across the same connection between the client and the broker, you need to understand how to balance the requirement for reliability with the need for throughput. This section describes how you can balance these requirements to manage flow control.

Delivery Mode

The delivery mode specifies whether a message is to be delivered at most once (non-persistent) or once and only once (persistent). These different reliability requirements imply different degrees of overhead. Specifically, the management of persistent messages requires greater use of broker control messages flowing across a connection.

Acknowledgement Mode

The setting of the acknowledgement mode impacts reliability and affects the number of client and broker acknowledgement messages passing over a connection:

- In the `AUTO_ACKNOWLEDGE` mode, a client message-consumed acknowledgement and broker acknowledgement (a confirmation of the client message-consumed acknowledgement) are required for each consumed message, and the delivery thread blocks waiting for the broker acknowledgement.

  In this mode, with a synchronous receiver, it is possible for a message to be partially processed, but lost, if the system fails before the message is consumed. For increased reliability, you can use the `CLIENT_ACKNOWLEDGE` mode or a transacted session to guarantee no message is lost if the system fails.

- In the `CLIENT_ACKNOWLEDGE` mode client message-consumed acknowledgements and broker acknowledgements are batched (rather than being sent one-by-one). This conserves connection bandwidth and generally reduces the overhead for broker acknowledgements, as compared to the `AUTO_ACKNOWLEDGE` mode.

- In the `DUPS_OK_ACKNOWLEDGE` mode, throughput is improved even further, because client acknowledgements are batched and because the client thread does not block (broker acknowledgements are not requested). However, in this case, the same message can be delivered and consumed more than once.

Message Flow Metering

The connection property MQ_CONNECTION_FLOW_COUNT_PROPERTY governs the batching of messages so that only a set number are delivered; when the batch has been delivered, delivery of JMS messages is suspended, and pending control messages are delivered. This cycle repeats, as other batches of JMS messages are delivered, followed by queued-up control messages.

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. If, on the other hand, the client has only simple consumers on a connection using DUPS_OK mode, you can increase the value of MQ_CONNECTION_FLOW_COUNT_PROPERTY without compromising performance.

Chapter 2

Building and Running Message Queue C Clients

This chapter provides information about building Message Queue C client applications and making sure these programs have adequate run-time support. It also lists the sample Message Queue C Client programs that are included with the Message Queue installation, and explains how you should run them

For information on how to use the API, see Chapter 3, “Using the C API” on page 47. For complete reference information, please see Chapter 4, “Reference” on page 73.

Getting Ready

Message Queue provides several sample Message Queue C-client applications that illustrate how to send and receive messages. These sample applications are installed in the `...demo\C` directory. 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 Programs

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

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`. Note that the Message Queue C-API runtime library is a 32-bit library. For each platform, Table 2-1 lists the installed location of the header files and the supporting runtime library.

Table 2-1 Locations of C-API Libraries and Header Files

Platform	Library	Header File
Solaris	`/opt/SUNWimq/lib`	`/opt/SUNWimq/include`
Linux	`/opt/imq/lib`	`/opt/imq/include`
Windows	`IMQ_HOME\lib`	`IMQ_HOME\include`

You should use the appropriate compiler for your platform, as described in the *Message Queue Installation Guide*.

When compiling a Message Queue C client application, you need to specify the preprocessor definition for supporting Message Queue fixed-size integer types. The preprocessor definition for each platform is shown in Table 2-2.

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

Platform	Definition
Solaris	SOLARIS
Linux	LINUX
Windows	WIN32

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 `msvcrt`.

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 "Building Programs" on page 43 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 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 are 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.

Building the Sample Programs

The following commands are meant to 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 -lmqcrt Producer.c
```

➤ **To Compile and Link on Linux**

```
g++ -DLINUX -D REENTRANT -I/opt/imq/include -o Producer \
    -L/opt/imq/lib -lmqcrt 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 mqcrt.lib
```

Running the Sample Programs

Sample C client program files are installed in the `...demo\C` directory. These 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.

The sample programs expect you to specify a destination as a command-line argument. You can either create one or more physical destinations on the broker by using the administration utility `imqcmd` before running the sample programs, or you can use the broker's auto-creation feature by specifying any destination name on the command line used to start the program.

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 …`demo\C` directory 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
```

Chapter 3

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 48
- "Working With Properties" on page 50
- "Working With Connections" on page 54
- "Working With Sessions and Destinations" on page 57
- "Working With Messages" on page 61
- "Error Handling" on page 68
- "Memory Management" on page 69
- "Thread Management" on page 70
- "Logging" on page 72

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 2, "Building and Running Message Queue C Clients" on page 43.

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 3-9 on page 67, 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 3-1 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 3-1 Functions Used to Set Properties

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

Table 3-1 Functions Used to Set Properties *(Continued)*

Function	Description
MQSetFloat32Property	Sets an MQFloat32 property.
MQSetFloat64Property	Sets an MQFloat64 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 3-1 illustrates how you create a properties handle and how you use it for setting connection properties.

Code Example 3-1 Setting Connection Properties

```
MQStatus status;
MQPropertiesHandle propertiesHandle = MQ_INVALID_HANDLE;

status = (MQCreateProperties(&propertiesHandle);

status = (MQSetStringProperty(propertiesHandle,
           MQ_BROKER_HOST_PROPERTY, “localhost”));
```

Code Example 3-1 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 `MQSetMessageProperties`.

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 then iterate through the properties and then use the appropriate `MQGet...Property` function to determine the type and value of each property.

Table 3-2 lists the functions you use to iterate through a properties handle and to obtain the type and value of each property.

Table 3-2 Functions Used to Get Message Properties

Function	Description
MQPropertiesKeyIterationStart	Starts the iteration process through the specified properties handle
MQPropertiesKeyIterationHasNext	Returns MQ_TRUE if there are additional property keys left in the iteration.
MQPropertiesKeyIterationGetNext	Passes back the address of the next property key in the referenced property handle.
MQGetPropertyType	Gets the type of the specified property.
MQGetBoolProperty	Gets the value of the specified MQBool type property.
MQGetStringProperty	Gets the value of the specified MQString type property.
MQGetInt8Property	Gets the value of the specified MQInt8 type property.
MQGetInt16Property	Gets the value of the specified MQInt16 type property.
MQGetInt32Property	Gets the value of the specified MQInt32 type property.
MQGetInt64Property	Gets the value of the specified MQInt64 type property.
MQGetFloat32Property	Gets the value of the specified MQFloat32 type property.
MQGetFloat64Property	Gets the value of the specified MQFloat64 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 3-2 illustrates how you implement these steps.

Code Example 3-2 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. These 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
- Whether the broker can handle secure messaging

You use the functions listed in Table 3-3 to create, start, stop, and close a connection.

Table 3-3 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.

Table 3-3 Functions Used to Work with Connections *(Continued)*

Function	Description
`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 50 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 82.
- If you want a secure connection, call the `MQIntitializeSSL` function to initialize security. This initializes the SSL library. See "Working With Secure Connections" on page 56 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 57.

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.

Working With Secure Connections

To set up a secure connection, you need to call the `MQInitializeSSl` function once (and only once) before you call the `MQCreateConnection` function, and you must set the `MQ_CONNECTION_TYPE_PROPERTY` to SSL. Depending on the operating system, you might also need to locate or create NSS certificate database files.

The `MQInitializeSSl` function initializes the NSS library. The `certificateDatabasePath` parameter you pass to the `MQInitializeSSL` function should point to a directory that contains the NSS files `cert`*N*`.db`, `key`*N*`.db`, and `secmod.db` (where *N* is a numeric digit). These certificate database files are opened read-only by the `MQInitializeSSL` function. You can generate the NSS certificate database files by using the Netscape or Mozilla browser. You can find the NSS certificate database files in the directory where the Netscape or Mozilla browser stores user settings, preferences, and bookmarks. For Mozilla, these files might not be created automatically. In that case, you can have them created by doing the following:

1. Start the Mozilla browser.
2. Choose Edit > Preferences > Privacy & Security > Certificates
3. Click the Manager Certificates... button.

Solaris 8 or 9 comes with the Netscape browser, as does RedHat Linux. For Windows, you can download the Mozilla browser from the following location:

`http://www.mozilla.org/`

After you call the `MQInitializeSSL` function, you can call `MQCreateConnection` to create an SSL connection to the Message Queue broker by setting the connection property `MQ_CONNECTION_TYPE_PROPERTY` to `SSL`. Setting the connection property `MQ_SSL_BROKER_IS_TRUSTED` to `MQ_TRUE` (the default is `MQ_FALSE`) is not tested or supported in this release. Before running your Message Queue C client application over SSL, you should configure the Message Queue broker to enable SSL-based connection services. See the *Message Queue Administration Guide* for instructions on configuring the broker.

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.

Table 3-4 describes the functions you use to create and manage sessions.

Table 3-4 Functions Used to Work with Sessions

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

Message Acknowledgement

When a message is delivered to a receiving client, the broker waits for the client to acknowledge receipt of a message before deleting the message from the broker's destination.

The receiving client can control messaging reliability by setting the session's acknowledge mode to one of the following values:

- `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.
- `MQ_DUPS_OK_ACKNOWLEDGE` specifies that the session acknowledges receipt of messages after each ten messages are consumed.

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.

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, see "Single-Threaded Session Control" on page 71.

- 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 3-5 lists the functions you use to create and to get information about destinations.

Table 3-5 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.

Table 3-5 Functions Used to Work with Destinations *(Continued)*

`MQGetDestinationType`	Returns the type (queue or topic) of the specified destination.

The `MQCreateDestination` function creates a destination object and passes a handle to it back to you. In a production environment, the Message Queue administrator has to also create a *physical destination* on the broker, whose name and type is the same as that of the destination object, in order for messaging to happen. 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`.

By default, the `imq.autocreate.topic` and `imq.autocreate.queue` properties for the broker 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.

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.

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 3-6 lists the functions you use to construct messages.

Table 3-6 Functions Used to Construct Messages

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

You begin by creating a message using either the MQCreateBytesMessage function or the MQCreateTextMessage function. Either of these functions return a message handle that you can then pass to the functions you use to set the message body, header, and properties using the functions listed in Table 3-6.

- Use the MQSetStringProperty 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 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 50.

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 3-7 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 used 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 3-7 Functions for Sending Messages

Function	Action
MQCreateMessageProducer	Creates a message producer with no specified destination.
MQCreateMessageProducerForDestination	Creates a message producer with a specified destination.
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 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 61.

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.

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 exception is thrown 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 3-8 lists the functions you use to create message consumers and to receive messages.

Table 3-8 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.

Table 3-8 Functions Used to Receive Messages *(Continued)*

Function	Description
`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 67, in order to read message properties and information stored in the header and body of the message.

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 67, 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 3-9.

Table 3-9 Functions Used to Process Messages

Function	Description
MQGetMessageHeaders	Gets message header properties.
MQGetMessageProperties	Gets user-defined message properties.
MQGetMessageType	Gets the message type: MQ_TEXT_MESSAGE or MQ_BYTES_MESSAGE
MQGetTextMessageText	Gets the body of an MQ_TEXT_MESSAGE message.
MQGetBytesMessageBytes	Gets the body of an MQ_BYTES_MESSAGE 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 52.

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 3-10 lists the functions you use to get error information.

Table 3-10 Functions Used in Handling Errors

Function	Description
`MQStatusIsError`	Returns an `MQ_TRUE` if the specified `MQStatus` is an error.
`MQGetStatusCode`	Returns the error code for the specified `MQStatus`.
`MQGetStatusString`	Returns a descriptive string for the specified `MQStatus`.
`MQGetErrorTrace`	Returns the calling thread's current error trace or `NULL` 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 3-11 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 3-11 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 `MQString`.

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.

Thread Management

This section addresses a number of thread management issues that you should be aware of in designing and programming a Message Queue C client.

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 3-12 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 3-12 Thread Model for NSPR GLOBAL Threads

Platform	Thread Model
Solaris	pthreads
Linux	pthreads
Windows	Win32 threads (from Microsoft Visual C++ runtime library `msvcrt`)

Concurrent Use of Handles

Table 3-13 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 3-13 Handles and Concurrency

Handle	Supports Concurrent Use
`MQDestinationHandle`	YES
`MQConnectionHandle`	YES
`MQSessionHandle`	NO
`MQProducerHandle`	NO
`MQConsumerHandle`	NO

Table 3-13 Handles and Concurrency *(Continued)*

Handle	Supports Concurrent Use
`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.

- 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_SESION_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.)
- You should 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 `MQConnectionExceptionListenerFunc` callback if one exists. If an `MQConnectionExceptionListenerFunc` 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 `MQConnectionExceptionListenerFunc` callback. Instead the callback function should notify another thread to call `MQCloseConnection` and return.

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.

Chapter 4

Reference

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

- "Data Types" on page 73 describes the C declarations for data types used by Message Queue messaging
- "Function Reference" on page 83 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 2, "Building and Running Message Queue C Clients" on page 43.

For information on how you use the C-API to complete specific programming tasks, see Chapter 3, "Using the C API" on page 47.

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 64 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.
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.
MQConnectionExceptionListenerFunc	The type of a callback function used for connection exception handling. For more information, see "Callback Type for Connection Exception Handling" on page 82.
MQFloat32	A 32-bit floating-point number.

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

MQType	Description
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 50 for more information.
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 mqcrt.h. See "Error Handling" on page 68 for more information on how you handle errors returned by Message Queue functions.
MQString	A null terminated UTF-8 encoded character string

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

MQType	Description
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.
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.

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
MQ_CONNECTION_TYPE_PROPERTY	An MQString specifying the transport protocol of the connection service used by the client. Supported types are TCP or SSL. **Default**: TCP
MQ_ACK_TIMEOUT_PROPERTY	A 32-bit integer specifying the maximum time in milliseconds that the client runtime will wait for any broker acknowledgement before returning an MQ_TIMEOUT_EXPIRED error. A value of 0 means there is no time-out. **Default**: 0
MQ_BROKER_HOST_PROPERTY	An MQString specifying the broker host name to which to connect. **No default.**
MQ_BROKER_PORT_PROPERTY	A 32-bit integer specifying the broker's primary port number. **No default.**
MQ_ACK_ON_PRODUCE_PROPERTY	An MQBool specifying whether the producing client waits for broker acknowledgement of receipt of message from the producing client. If set to MQ_TRUE, 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. If set to MQ_FALSE, 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. **Default**: the broker acknowledges receipt of *persistent* messages only from the producing client, and the producing client thread will block waiting for those acknowledgements.

Table 4-2 Connection Properties *(Continued)*

Key Name	Description
MQ_ACK_ON_ACKNOWLEDGE_PROPERTY	An MQBool 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 MQ_DUPS_OK_ACKNOWLEDGE, this flag has no effect. If set to MQ_TRUE, the broker acknowledges all consuming acknowledgements, and the consuming client thread blocks waiting for these broker acknowledgements. If set to MQ_FALSE, the broker does not acknowledge any consuming client acknowledgements, and the consuming client thread will not block waiting for such broker acknowledgements. **Default**: MQ_TRUE For more information, see the discussion for the MQAcknowledgeMessages function and "Message Acknowledgement" on page 58.
MQ_CONNECTION_FLOW_COUNT_PROPERTY	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. **Default**: 100
MQ_CONNECTION_FLOW_LIMIT_ENBABLED _PROPERTY	An MQBool specifying whether the value MQ_CONNECTION_FLOW_LIMIT_PROPERTY is used to control message flow. Specify MQ_TRUE to use the value and MQ_FALSE otherwise. **Default**: MQ_FALSE

Table 4-2 Connection Properties *(Continued)*

Key Name	Description
MQ_CONNECTION_FLOW_LIMIT_PROPERTY	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 MQ_CONNECTION_FLOW_LIMIT_ENBABLED_PROPERTY is MQ_TRUE, this limit is not checked. 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 MQ_CONNECTION_FLOW_COUNT_PROPERTY) only when the number of unconsumed messages drops below the value set with this property. 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. **Default**: 1000
MQ_SSL_BROKER_IS_TRUSTED	An MQ_Bool specifying whether the broker is trusted. **Default**: MQ_TRUE
MQ_SSL_CHECK_BROKER_FINGERPRINT	An MQ_Bool. If it is set to MQ_TRUE and if MQ_SSL_BROKER_IS_TRUSTED is MQ_FALSE, the broker's certificate fingerprint is compared with the MQ_SSL_BROKER_CERT_FINGERPRINT property value in case of certificate authorization failure. If they match, the broker's certificate is authorized for use in the SSL connection. **Default**: MQ_FALSE
MQ_SSL_BROKER_CERT_FINGERPRINT	An MQString specifying the MD5 hash, in hex format, of the broker's certificate. **Default**: NULL
MQ_NAME_PROPERTY	An MQString that specifies the name of the Message Queue product.This property is set by the runtime library. See the MQGetMetaData function for more information.
MQ_VERSION_PROPERTY	An MQInt32 that specifies the version of the Message Queue product. This property is set by the runtime library. See the MQGetMetaData function for more information.
MQ_MAJOR_VERSION_PROPERTY	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. This property is set by the runtime library. See the MQGetMetaData function for more information.

Table 4-2 Connection Properties *(Continued)*

Key Name	Description
MQ_MINOR_VERSION_PROPERTY	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. This property is set by the runtime library. See the MQGetMetaData function for more information.
MQ_MICRO_VERSION_PROPERTY	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. This property is set by the runtime library. See the MQGetMetaData function for more information.
MQ_SERVICE_PACK_PROPERTY	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. This property is set by the runtime library. See the MQGetMetaData function for more information.

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 MQAcknowledgeMessages function has been called. See the discussion of the function MQAcknowledgeMessages 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.

Table 4-3 acknowledgeMode Values *(Continued)*

Enum	Description
MQ_SESSION_TRANSACTED	This value is read only. It is set by the library if you have passed MQ_TRUE for the isTransacted parameter to the MQCreateSession function. It is returned to you by the MQGetAcknowledgeMode 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

sessionHandle	The handle to the session to which this consumer belongs. The client runtime specifies this handle when it calls your message listener.
consumerHandle	A handle to the consumer receiving the message. The client runtime specifies this handle when it calls your message listener.
messageHandle	A handle to the incoming message. The client runtime specifies this handle when it calls your message listener.
callbackData	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 must 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 MQConnectionExceptionListenerFunc type has the following definition:

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

Parameters

`connectionHandle`	The handle to the connection on which the connection exception occurred. The client runtime sets this handle when it calls the connection exception handler.
`exception`	An `MQStatus` 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 68.
`callbackData`	Whatever void pointer was passed as the `listenerCallbackData` parameter to the `MQCreateConnection` 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.

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

Function	Description
MQCreateAsyncDurableMessageConsumer	Creates a durable asynchronous message consumer for the specified destination.
MQCreateAsyncMessageConsumer	Creates an asynchronous message consumer for the specified destination.
MQCreateBytesMessage	Creates an MQ_BYTES_MESSAGE message.
MQCreateConnection	Creates a connection to the broker.
MQCreateDestination	Creates a 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.
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 MQString.
MQGetAcknowledgeMode	Passes back the acknowledgement mode of the specified session.
MQGetBoolProperty	Passes back a property of type MQBool.
MQGetBytesMessageBytes	Passes back the address and size of a MQ_BYTES_MESSAGE message body.
MQGetDestinationType	Passes back the type of the specified destination.

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

Function	Description
MQGetErrorTrace	Returns a string describing the stack at the time the specified error occurred.
MQGetFloat32Property	Passes back the value of the MQFloat32 property for the specified key.
MQGetFloat64Property	Passes back the value of the MQFloat64 property for the specified key.
MQGetInt16Property	Passes back the value of the MQInt16 property for the specified key.
MQGetInt32Property	Passes back the value of the MQInt32 property for the specified key.
MQGetInt64Property	Passes back the value of the MQInt64 property for the specified key.
MQGetInt8Property	Passes back the value of the MQInt8 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 MQStatus result.
MQGetStatusString	Returns a string description for the specified MQStatus result.
MQGetStringProperty	Passes back the value for the specified property. *Type* (in the function name) can be String, Bool, Int8, Int16, Int32, Int64, Float32, Float64.
MQGetTextMessageText	Passes back the contents of an MQ_TEXT_MESSAGE 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.

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

Function	Description
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 MQBool property with the specified key to the specified value.
MQSetBytesMessageBytes	Sets the message body for the specified MQ_BYTES_MESSAGE message.
MQSetFloat32Property	Sets an MQFloat 32 property with the specified key to the specified value.
MQSetFloat64Property	Sets an MQFloat 64 property with the specified key to the specified value.
MQSetInt16Property	Sets an MQInt16 property with the specified key to the specified value.
MQSetInt32Property	Sets an MQInt 32 property with the specified key to the specified value.
MQSetInt64Property	Sets an MQInt64 property with the specified key to the specified value.
MQSetInt8Property	Sets an MQInt8 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 MQString property with the specified key to the specified value.

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

Function	Description
MQSetStringProperty	Sets the message body for the specified MQ_TEXT_MESSAGE 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 MQ_TRUE if the specified MQStatus 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

`sessionHandle`	The handle to the session for the consumer that received the specified message.
`messageHandle`	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

`consumerHandle`	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

`sessionHandle`	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

sessionHandle	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 MQ_SESSION_ASYNC_RECEIVE receive mode.
destinationHandle	A handle to a topic destination on which the consumer receives messages. This handle remains valid after the call.
durableName	An MQString specifying a name for the durable subscriber. The library makes a copy of the durableName string.
messageSelector	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a NULL 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 messageSelector 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.
noLocal	Specify MQ_TRUE to inhibit delivery of messages published by this consumer's own connection.

`messageListener`	The name of an `MQMessageListenerFunc` type callback function that is to be called when this consumer receives a message on the specified destination.
`listenerCallbackData`	A pointer to data that you want passed to your message listener function when it is called by the library.
`consumerHandle`	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_EXCEEDEED`

`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

`sessionHandle`	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 `MQ_SESSION_ASYNC_RECEIVE` receive mode.
`destinationHandle`	A handle to the destination on which the consumer receives messages. This handle remains valid after the call returns.
`messageSelector`	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a `NULL` 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 `messageSelector` 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.
`noLocal`	Specify `MQ_TRUE` 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.

`messageListener`	The name of an `MQMessageListenerFunc` type callback function that is to be called when this consumer receives a message for the specified destination.
`listenerCallbackData`	A pointer to data that you want passed to your message listener function when it is called by the library.
`consumerHandle`	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_EXCEEDEED`

`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,
             MQConnectionExceptionListenerFunc exceptionListener,
             void * listenerCallBackData,
             MQConnectionHandle * connectionHandle);
```

Return Value

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

Parameters

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

`listenerCallBackData`	A data pointer that can be passed to the connection `exceptionListener` 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 `NULL` if you do not need to pass data back to the connection exception listener.
`connectionHandle`	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 `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.

- 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.

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 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

`sessionHandle`	The handle to the session with which you want to associate this destination.
`destinationName`	An `MQString` specifying the logical name of this destination. The library makes a copy of the `destinationName` string. See discussion below. Destination names starting with “mq” are reserved and should not be used by clients.
`destinationType`	An enum specifying the destination type, either `MQ_QUEUE_DESTINATION` or `MQ_TOPIC_DESTINATION`.
`destinationHandle`	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

sessionHandle	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 MQ_SESSION_SYNC_RECEIVE receive mode.
destinationHandle	A handle to a topic destination on which the consumer receives messages. This handle remains valid after the call returns.
durableName	An MQString specifying the name of the durable subscriber to the topic destination. The library makes a copy of the durableName string.
messageSelector	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a NULL 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 messageSelector 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.

`noLocal`	Specify `MQ_TRUE` to inhibit delivery of messages published by this consumer's own connection.
`consumerHandle`	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_LIMITE_EXCEEDEED

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

`sessionHandle`	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 `MQ_SESSION_SYNC_RECEIVE` receive mode.
`destinationHandle`	A handle to the destination on which the consumer receives messages. This handle remains valid after the call returns.
`messageSelector`	An expression (based on SQL92 conditional syntax) that specifies the criteria upon which incoming messages should be selected for this consumer. Specify a `NULL` 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 `messageSelector` 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.
`noLocal`	Specify `MQ_TRUE` to inhibit delivery of messages published by this consumer's own connection. This applies only to topic destinations; it is ignored for queues.
`consumerHandle`	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_EXCEEDEED

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

sessionHandle	The handle to the session to which this producer should belong.
producerHandle	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

`sessionHandle`	The handle to the session to which this producer belongs.
`destinationHandle`	A handle to the destination where you want this producer to send all messages. This handle remains valid after the call returns.
`producerHandle`	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

`propertiesHandle`	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

`connectionHandle`	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.
`isTransacted`	An `MQBool` specifying whether this session is transacted. Specify `MQ_TRUE` if the session is transacted. In this case, the `acknowledgeMode` parameter is ignored.
`acknowledgeMode`	An enumeration of the possible kinds of acknowledgement modes for the session. See Table 4-3 on page 80 for information on these values. After you have created a session, you can determine its acknowledgement mode by calling the `MQGetAcknowledgeMode` function.
`receiveMode`	An enumeration specifying whether this session will do synchronous or asynchronous message receives. Specify `MQ_SESSION_SYNC_RECEIVE` or `MQ_SESSION_ASYNC_RECEIVE`. If the session is only for producing messages, the `receiveMode` has no significance. In that case, specify `MQ_SESSION_SYNC_RECEIVE` to optimize the session's resource use.
`sessionHandle`	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

`sessionHandle`	The handle to the session with which you want to associate this destination.
`destinationType`	An enum specifying the destination type, either `MQ_QUEUE_DESTINATION` or `MQ_TOPIC_DESTINATION`.
`destinationHandle`	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

`sessionHandle`	The handle to the session whose acknowledgement mode you want to determine.
`ackMode`	Output parameter for the `ackMode`. The `ackMode` returned can be one of four enumeration values. See Table 4-3 on page 80 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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

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

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

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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

messageHandle	The message handle.
headersHandle	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
MQ_CORRELATION_ID_HEADER_PROPERTY	MQString	Client (optional)
MQ_MESSAGE_TYPE_HEADER_PROPERTY	MQString	Client (optional)
MQ_PERSISTENT_HEADER_PROPERTY	MQBool	Send function
MQ_EXPIRATION_HEADER_PROPERTY	MQInt64	Send function
MQ_PRIORITY_HEADER_PROPERTY	MQInt8	Send function
MQ_TIMESTAMP_HEADER_PROPERTY	MQInt64	Send function
MQ_MESSAGE_ID_HEADER_PROPERTY	MQString	Send function
MQ_REDELIVERED_HEADER_PROPERTY	MQBool	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

`messageHandle`	A handle to a message whose properties you want to get.
`propertiesHandle`	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

messageHandle	A handle to a message expecting a reply. This is the handle that is passed back to you when you receive the message.
destinationHandle	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

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

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

`connectionHandle`	The handle to the connection that you want the meta information about.
`propertiesHandle`	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

`propertiesHandle`	A properties handle that you want to access.
`key`	The property key for which you want to get the type of the property value.
`propertyType`	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

`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 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

`propertiesHandle`	A properties handle whose property value for the specified key you want to get.
`key`	The name of a property key.
`value`	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

messageHandle	A handle to an MQ_TEXT_MESSAGE message that is passed to you when you receive a message.
messageText	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. You must call this function once and only once before you create any connection that uses SSL.

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

Return Value

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

Parameters

`certificateDatabasePath`	An `MQString` specifying the path to the certificate data base.

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

Common Errors

`MQ_INCOMPATIBLE_LIBRARY`

`MQ_SSL_ALREADY_INITIALIZED`

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

`propertiesHandle`	A properties handle whose contents you want to access.
`key`	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

`consumerHandle`	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
`messageHandle`	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`.

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

`consumerHandle`	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
`messageHandle`	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.

consumerHandle	The handle to the message consumer. This handle is passed back to you when you create a synchronous message consumer.
timeoutMilliseconds	The number of milliseconds to wait for a message to arrive.
messageHandle	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 timoutMilliseconds 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 MQCRecoverSession 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_ACKNOWELDGE 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

producerHandle	The handle to the producer sending this message. This handle is passed back to you by the MQCreateMessageProducerForDestination function.
messageHandle	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

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

`msgDeliveryMode`	An enum
	`MQ_PERSISTENT_DELIVERY`
	`MQ_NONPERSISTENT_DELIVERY.`

`msgPriority`	A integer value of 0 through 9; 0 being the lowest priority and 9 the highest.
`msgTimeToLive`	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 he 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
msgDeliveryMode	MQ_PERSISTENT_DELIVERY
msgPriority	4
msgTimeToLive	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

producerHandle	The handle to the producer sending this message. This handle is passed back to you by the MQCreateMessageProducer function.
messageHandle	A handle to the message you want to send.
destinationHandle	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

producerHandle	The handle to the producer sending this message. This handle is passed back to you when you call the MQCreateMessageProducer function.
messageHandle	A handle to the message you want to send.
destinationHandle	A handle to the destination where you want to send the message.
msgDeliveryMode	An enum of either MQ_PERSISTENT_DELIVERY or MQ_NONPERSISTENT_DELIVERY.
msgPriority	A integer value of 0 through 9; 0 being the lowest priority and 9 the highest.
msgTimeToLive	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
msgDeliveryMode	MQ_PERSISTENT_DELIVERY
msgPriority	4
msgTimeToLive	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

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

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

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

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

`propertiesHandle`	A handle to the properties object whose property value for the specified key you want to set.
`key`	The name of a property key. The library makes a copy of the property key.
`value`	The `MQFloat64` 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

`propertiesHandle`	A handle to the properties object whose property value for the specified key you want to set.
`key`	The name of a property key. The library makes a copy of the property key.
`value`	The `MQInt16` 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

`propertiesHandle`	A handle to the properties object whose property value for the specified key you want to set.
`key`	The name of a property key. The library makes a copy of the property key.
`value`	The `MQInt32` 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

`propertiesHandle`	A handle to the properties object whose property value for the specified key you want to set.
`key`	The name of a property key. The library makes a copy of the property key.
`value`	The `MQInt64` 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

`propertiesHandle`	A handle to the properties object whose property value for the specified key you want to set
`key`	The name of a property key. The library makes a copy of the property key.
`value`	The `MQInt8` 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

messageHandle	A handle to a message.
headersHandle	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
MQ_CORRELATION_ID_HEADER_PROPERTY	MQString	Client (optional)
MQ_MESSAGE_TYPE_HEADER_PROPERTY	MQString	Client (optional)
MQ_PERSISTENT_HEADER_PROPERTY	MQBool	Send function
MQ_EXPIRATION_HEADER_PROPERTY	MQInt64	Send function
MQ_PRIORITY_HEADER_PROPERTY	MQInt8	Send function
MQ_TIMESTAMP_HEADER_PROPERTY	MQInt64	Send function
MQ_MESSAGE_ID_HEADER_PROPERTY	MQString	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

messageHandle	A handle to a message whose application-defined properties you want to set.
propertiesHandle	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 50 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

messageHandle	A handle to a message expecting a reply.
destinationHandle	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

propertiesHandle	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.
key	The name of a property key. The library makes a copy of the property key
value	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

messageHandle	A handle to a message whose text body you want to set.
messageText	An MQString 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

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

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

sessionHandle	The handle to the session to which this consumer belongs. This handle is created and passed back to you by the MQCreateSession function.
durableName	An MQString 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
`mqbasictypes.h`	Defines the types `MQBool`, `MQInt8`, `MQInt16`, `MQInt32`, `MQInt64`, `MQFloat32`, `MQFloat64`.
`mqbytes-message.h`	Function prototypes for creating, getting, setting bytes message.
`mqcallback-types.h`	Asynchronous receive and connection exception handling callback types.
`mqconnection.h`	Function prototypes for creating, managing, and closing connections. Function prototype for creating session.
`mqconnection-props.h`	Connection property constants
`mqconsumer.h`	Function prototypes for synchronous receives and closing the consumer.
`mqcrt.h`	All Message Queue C-API public header files.
`mqdestination.h`	Function prototypes to free destinations and get information about destinations.
`mqerrors.h`	Error codes
`mqheader-props.h`	Message header property constants
`mqmessage.h`	Function prototypes for getting and setting parts of message, freeing message, and acknowledging message.
`mqproducer.h`	Function prototypes for sending messages and closing the message producer.
`mqproperties.h`	Function prototypes for creating, setting, and getting properties
`mqsession.h`	Function prototypes for managing and closing sessions; for creating destinations, message producers and message consumers.
`mqssl.h`	Function declaration for initializing the SSL library.
`mqstatus.h`	Function prototypes for getting error information.

Table 4-7 Message Queue C-API Header Files *(Continued)*

`mqtext-message.h`	Function prototypes for creating, getting, setting text message.
`mqtypes.h`	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.
`mqversion.h`	Version information constant definitions.

Appendix A

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 68.

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_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_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_TIMEOUT		Broker: timeout
MQ_BROKER_UNAUTHORIZED		Broker: unauthorized
MQ_BROKER_UNAVAILABLE		Broker: unavailable
MQ_CANNOT_UNSUBSCRIBE_ACTIVE_CONSUMER	X	Cannot unsubscribe an active consumer.
MQ_CLIENTID_IN_USE	X	Client id already in use
MQ_CONCURRENT_ACCESS	X	Concurrent access
MQ_CONCURRENT_DEADLOCK	X	Operation may cause deadlock

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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).
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

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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
MQ_NO_CONNECTION		The Session's connection has been closed
MQ_NO_MESSAGE	X	There was no message to receive.

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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_INT16		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
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

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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_SERIALIZE_BAD_CLASS_UID		Serialize bad class UID
MQ_SERIALIZE_BAD_HANDLE		Serialize bad handle
MQ_SERIALIZE_BAD_MAGIC_NUMBER		Serialize bad magic number
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

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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
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.

Table A-1 Message Queue C Client Error Codes *(Continued)*

Code	Common	Description
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
  about 31
  broker, *See* broker acknowledgements
  client, *See* client acknowledgements
  data type for 74
  periodic 80
administration tools 23
administration, about 23
applications, example 17
applications, *See* JMS clients
AUTO_ACKNOWLEDGE mode 35

B

broker
  acknowledging consumed messages 31, 78
  acknowledging sent messages 77
  certificate for 79
  control messages 78
  host port for 77
  name for 77
  security 79
broker acknowledgements
  about 34
  automatic 80

C

C API
  header files 44
  runtime library 44
certificate database files 56
client acknowledgements 88
  about 35
  effect on performance 41
  explicit 80
  modes, *See* client acknowledgment modes
client acknowledgment modes
  AUTO_ACKNOWLEDGE 35
  CLIENT_ACKNOWLEDGE 35
  DUPS_OK_ACKNOWLEDGE 36
client identifier (ClientID) 30
client runtime
  about 22
  configurable properties 38
  message consumption, and 34
  message production, and 34
CLIENT_ACKNOWLEDGE mode 35
connection properties
  iterating through 148
  type of 140
connections
  about 27
  closing 90
  creating 55, 101
  creating properties for 51, 113
  exceptions 74, 82

connections *(continued)*
freeing 69, 118
freeing properties of 69, 121
handle to 74
orderly shutdown 57
properties of 75, 76
secure, initializing 56, 145
specifying 38
starting 179
stopping 181
timed out limit 77
transport protocol for 77
ConstMQString type 74
consumers 28
asynchronous 66
closing 91
creating asynchronous 98
creating asynchronous durable 95
creating durable 106
creating synchronous 109
handle to 74
synchronous 66, 149, 151, 153
type of 75
unsubscribing durable 182
working with 65

D

delivery modes
data type for 74
effects on performance 40
non-persistent 30
persistent 30
delivery, reliable 30
destinations
creating 59, 104
creating temporary 116
freeing 119
getting type of 126
handle to 74
type of 74
directory variables
IMQ_HOME 15
domains 29
DUPS_OK_ACKNOWLEDGE mode 36
durable subscribers, *See* durable subscriptions
durable subscriptions
about 29
ClientID, and 30

E

environment variables, *See* directory variables
error handling
error trace 127
error type 74
getting status code 141
MQStatus type 75
status string 142
example applications 17
exceptions
listener for 74

F

fixed integer type support 45
flow count, message 42

H

header files 44, 183

I

IMQ_HOME directory variable 15

J

JMS API 24

JMS clients
about 24
client runtime, and 22
performance, *See* performance
programming model 24
setup summary 48
JMS specification 17
JMSCorrelationID message header field 26
JMSDeliveryMode message header field 25
JMSDestination message header field 25
JMSExpiration message header field 25
JMSMessageID message header field 25
JMSPriority message header field 25
JMSRedelivered message header field 26
JMSReplyTo message header field 26
JMSTimestamp message header field 26
JMSType message header field 26

L

licenses for Message Queue editions 22
listeners, message
about 28
asynchronous consumption, and 35
data type for 75, 81
logging 72

M

memory management 69
message acknowledgements 58
message consumption
about 28, 34
asynchronous 36, 81
synchronous 36
message delivery models 29
message headers
fields 25
getting 134
properties 63
setting 173
message listeners, *See* listeners, message
message producers 28
message properties
default values for 63
getting 52, 136
handle to 75
introduced 26
iterating through 53, 148
setting 175
type of 140
Message Queue
fixed integer type support 45
header files 183
licenses for 22
meta data for 139
name of 79
product editions 22
version of 79
Message Queue message server 22
Message Queue programs, building 43
messages
about 24
acknowledging 88
body 26
composing 61
consumption of, *See* message consumption
correlation id 134
creating bytes type 100
creating text type 117
delivery models 29
delivery modes, *See* delivery modes
delivery of 33
duplicate sends 36
expiration of 134
filtering 67
flow count 42
freeing 120
getting text of 144
getting type of 138
handle to 75
headers, *See* message headers
limit of unconsumed 79
listeners for, *See* listeners, message
mode of 134

messages *(continued)*
ordering of 37
persistent 30
persistent storage 32
point-to-point delivery 29
prioritizing 37
priority of 134
processing 67
production of 34
publish/subscribe delivery 29
receiving 64
redelivered status 134
reliable delivery of 31
reply-to destination 137, 176
selection and filtering of 37
selector for 95, 98, 106
sending 62, 157, 159, 161, 163
set text of 178
type of 75, 134
messages properties
creating 113
freeing 121
messaging system, architecture 22
MQ_ACK_ON_ACKNOWLEDGE_PROPERTY 31, 58, 78
MQ_ACK_ON_PRODUCE_PROPERTY 77
MQ_ACK_TIMEOUT_PROPERTY 39, 77
MQ_AUTO_ACKNOWLEDGE enum 80
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 80
MQ_CONNECTION_FLOW_COUNT_PROPERTY 39, 78
MQ_CONNECTION_FLOW_LIMIT_ENABLED_PROPERTY 39
MQ_CONNECTION_FLOW_LIMIT_ENBABLED_PROPERTY 78
MQ_CONNECTION_FLOW_LIMIT_PROPERTY 39, 79
MQ_CONNECTION_TYPE_PROPERTY 77
MQ_CONNECTION_TYPEPROPERTY 56
MQ_CORRELATION_ID_HEADER_PROPERTY 134
MQ_DUPS_OK_ACKNOWLEDGE enum 80
MQ_EXPIRATION_HEADER_PROPERTY 63, 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 72
MQ_LOG_LEVEL 72
MQ_MESSAGE_ID_HEADER_PROPERTY 134
MQ_MESSAGE_TYPE_HEADER_PROPERTY 134
MQ_NAME_PROPERTY 79
MQ_PERSISTENT_HEADER_PROPERTY 63, 134
MQ_PRIORITY_HEADER_PROPERTY 63, 134
MQ_REDELIVERED_HEADER_PROPERTY 134
MQ_SESSION_ASYNC_RECEIVE 59
MQ_SESSION_ASYNC_RECEIVE consumer type 75
MQ_SESSION_SYNC_RECEIVE 59
MQ_SESSION_SYNC_RECEIVE consumer type 75
MQ_SESSION_TRANSACTED enum 81
MQ_SSL_BROKER_IS_TRUSTED 56, 79
MQ_SSL_CHECK_BROKER_FINGERPRINT 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 74, 82
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 45
mqcrt runtime library 44
MQDeliveryMode type 74
MQDestinationHandle type 74
MQDestinationType type 74
MQError type 74
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 56, 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 75
MQRecoverSession function 155
MQRollBackSession function 156
MQSendMessage function 157
MQSendMessageExt function 159
MQSendMessageToDestination function 161
MQSendMessageToDestinationExt function 163
MQSessionHandle type 75
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 75
MQStatusIsError function 180
MQStopConnection function 181
MQString type 75
MQType type 76
MQUnsubscribeDurableMessageConsumer function 182

N

NSPR library 45
NSS library 45

P

performance
  effect of delivery mode 40
  factors affecting 40
  message flow count 42
persistence
  about 32
  delivery modes, *See* delivery modes
  persistent messages 30
point-to-point delivery 29
producers 28
  closing 92
  creating 111
  creating for destination 112
  handle to 75
programming domains 29
programming examples
  build instructions 45
  running 46
properties, client runtime *See* client runtime
publish/subscribe delivery 29

Q

queue destinations 29

R

reliable delivery 30
runtime library 44

S

sample programs
  compiler options for 44
  running 46
selection, of messages 37
sessions
  about 28
  acknowledge mode of 123
  acknowledgement options for 31
  closing 93
  committing 94
  creating 58, 114
  handle to 75
  managing 59
  recovering 155
  rolling back 156
  transacted 31, 58, 81

T

thread management 70
topic destinations 29
transactions
  about 31
  committing 94
  working with 58