Content

Chapter 1: Introduction

The Native C++Runtime Library (C++RTL) allows C++ based application to interact with FioranoMQ. A C++ based client can thus seamlessly communicate with Java based Fiorano Clients.

This version of C++RTL is designed to run on both native and .NET platforms. Both the versions support secure and non-secure TCP and HTTP connections on Win32 platform for Point-to-Point and Publish/Subscribe communication models.

The C++ Runtime is designed to provide maximum conformance with the JMS specifications. All public APIs have similar signature as the corresponding java APIs specified by JMS. The classes have similar naming convention.

This guide shows you how to use all the ActiveX Runtime Library functions, to create Tifosi Services in VB and Delphi. This is a reference guide discussing all APIs of Tifosi ActiveX RTL in detail, assuming you know sufficient about VB and Delphi to read and understand the sample code.

The ActiveX Runtime Service APIs allow inter-operability between services, and provide the same functionality as the Java Services Development APIs.

The Native C++Runtime Library (C++RTL) allows C++ based application to interact with FioranoMQ. A C++ based client can thus seamlessly communicate with Java based Fiorano Clients.

This version of C++RTL is designed to run on both native and .NET platforms. Both the versions support secure and non-secure TCP and HTTP connections on Win32 platform for Point-to-Point and Publish/Subscribe communication models.

The C++ Runtime is designed to provide maximum conformance with the JMS specifications. All public APIs have similar signature as the corresponding java APIs specified by JMS. The classes have similar naming convention.

Chapter 2: Datatypes and Constants

This chapter contains an overview of the C++RTL specific data types and constants. A brief explanation of each data type along with sizes, is provided.

Basic Data Types and their Sizes

This section lists the basic data types used in C++RTL APIs, indicating the size of each.

- mqbyte Data defined to occupy 8 bits (unsigned).
- mqchar Data defined to occupy 8 bits.
- mqshort Data defined to occupy 16 bits.
- mqint Data defined to occupy 32 bits.
- mqlong Data defined to occupy 64 bits.
- mqfloat Data defined to occupy 32 bits.
- mqdouble Data defined to occupy 64 bits.

For more information on the data types, refer to section data types and constants of FioranoMQ C++RTL guide.

C++RTL Constants

All required public constants are categorized and declared in the related classes according to JMS specifications.

Class	Constant

Cmessage	`static const int CDEFAULT_DELIVERY_MODE`
	`static const int CDEFAULT_PRIORITY`
	`static const int CDEFAULT_TIME_TO_LIVE`

CdeliveryMode	`static const int CNON_PERSISTENT;` `static const int CPERSISTENT SET_CPERSISTENT;`

Class	Constant

CJMSSession	`static const int CAUTO_ACKNOWLEDGE`
	`static const int CCLIENT_ACKNOWLEDGE`
	`static const int CDUPS_OK_ACKNOWLEDGE`
	`static const int CSESSION_TRANSACTED`

For more information on these constants, refer to the Java docs of the containing class.

Naming convention

The C++RTL adheres to the following naming convention for you to easily identify the classes, constants and member functions with the corresponding definitions in JMS specifications.

Type	Naming Convention	Example
Class	C<JMSClass name>	`CTopicPublisher(JMS:TopicPub-lisher)`
Constant	C<JMS constants name>	`CNON_PERSISTENT()`
Function	same as in JMS spec	`publish`

This guide shows you how to use all the ActiveX Runtime Library functions, to create Tifosi Services in VB and Delphi. This is a reference guide discussing all APIs of Tifosi ActiveX RTL in detail, assuming you know sufficient about VB and Delphi to read and understand the sample code.

The ActiveX Runtime Service APIs allow inter-operability between services, and provide the same functionality as the Java Services Development APIs.

Chapter 3: Error Handling

The C++RTL uses exceptions to provide error handling. In event of an error in the C++RTL layer, CJMSException with a specific errorCode and description is thrown.

The Exception can be caught at the application level and the associated message can be read using the public API getMessage(). The exception handling is also exhaustive in that it provides the complete function stack trace at the moment of the error.

The exception stack is maintained in the Thread Local Storage, so that the excep- tion that occurred in one thread doesn't interfere with the flow of other threads. The stack trace can be printed on the console using the API call printStackTrace().

```
try
{
//Application code
}
catch(CJMSException *e)
                              {
                              cout << e->getMessage();
                              e->printStackTrace();
                              delete e;
                              }
```

For more information, read the CJMSException**Error! Reference source not found.** section.

Chapter 4: The DotNet Version

Microsoft's .NET™ is one of the emerging standards that is changing the way enterprises build Application software. The .NET technologies enable greater agility and increased flexibility for integrating business processes within internal organizations or with external business partners.

Fiorano's MQ provides support for .NET platform through the .NET version of C++ RTL. Any .NET application can be compiled with this rtl and deployed as a dotnet component. Such a component can communicate with any other component, written in any supported language and executing on any supported windows/non-windows platform.

The key point to note here is that these .NET applications execute in their native form without the need for conversion into web services. This is a unique feature that leads to high performance and lower maintenance costs and makes it ahead of all other EAI/BPM platforms in terms of interoperability.

The DotNet version of the native CppRTL has been implemented using the managed extensions provided with Microsoft Visual C++ ™. The dotNet version provides all the functionality of CppRTL with additional support for garbage collection provided by the dotnet platform.

Installation

The same set of header files can be used while compiling dotnet applications with FioranoMQ.

Make sure to compile with the DOTNET preprocessor definition. Follow the same set of instructions listed above.

Using DotNet samples

Organization of samples

The native C++ samples are located in the cpp\native\samples\ folder in FioranoMQ installation and organized under the following heads:

PubSub This directory contains the following sample programs, which illustrate basic JMS Publish/Subscribe functionality, using the C++ RTL.

PTP This directory contains two sample programs which illustrate JMS Request-Reply mechanism using the CppRTL.

Compiling the samples

To run the samples using FioranoMQ,

Compile each of the source files using the script files in the fmq\clients\cpp\native\scripts folder.

The scripts directory contains a script cppclientbuild.bat which compiles the samples using the Microsoft VC++ compiler.

Running the samples

Refer to the readme.txt file in the specific directory for information(if any) on the runtime arguments for the executable.

Chapter 5: API Reference

This document lists all FioranoMQ C++ Runtime APIs by category. The JMSInterface APIs are those that are common for both Publish/subscribe and PTP semantics, Naming and JNDI APIs are those that perform naming operations, Helper Function APIs are the utility functions, Publish/Subscribe APIs are those that implement the JMS Publish/Subscribe semantics and PTP APIs are those that implement the JMS Point To Point semantics.

The organization of this document is summarized as follows.

Helpers

CHashTable

This class implements a hashtable, which maps keys to values. Any non-null mqobject can be used as a value and any non-null mqcstring can be used as a key.

Constructor

Constructs a new, empty hashtable.

Declaration

```
CHashTable()
            throw (CJMSException *);
```

Put

Maps the specified key to the specified value in this hashtable. Neither the key nor the value can be null.

Declaration

```
mqobject Put(mqcstring key, mqobject value)

               throw (CJMSException *);
```

Get

Returns the value to which the specified key is mapped in this hashtable.

Declaration

```
mqobject Get(mqcstring key) FMQCONST

              throw (CJMSException *);
```

RemoveElement

Removes the key (and its corresponding value) from this hashtable. This method does nothing if the key is not in the hashtable.

Declaration

```
mqobject RemoveElement(mqcstring key)
                            throw (CJMSException *);
```

HashTableEnumerator

CHashTableEnumerator generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series.

hasMoreElemets

Tests if this enumeration contains more elements. Returns true if enumeration contains more elements otherwise it returns false

Declaration

```
mqboolean hasMoreElements();
```

nextKeyElement

Returns the nextKey value of this enumeration if this enumeration object has at least one more element to provide.

Declaration

```
mqcstring nextKeyElement();
```

Returns

Returns the next Key value

nextValueElement

Returns the next value element of this enumeration if thisenumeration object has at least one more element to provide

Declaration

```
mqobject nextValueElement();
```

Returns

Returns the next value Element

JMS Interfaces

CAdvisoryMessage

Base class for all advisory messages.

getAdvisoryMsgString

Returns a Unicode string for the advisory message.

Declaration

```
mqcstring_unicode getAdvisoryMsgString()
throw (CJMSException *);
```

Returns

The Unicode string value for the advisory message string

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getAMState

Returns a mqint object representing the Advisroy message state

Declaration

```
mqint getAMState()
throw (CJMSException *);
```

Returns

The advisory message state as mqint object

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isActive

Returns a mqboolean value representing the status of the advisory message object.

Declaration

```
mqboolean isActive()
throw (CJMSException *);
```

Returns

Returns True if advisory message object is active, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isDisconnected

Returns a mqboolean value representing the connection status of the advisory message object.

Declaration

```
mqboolean isDisconnected()
throw (CJMSException *);
```

Returns

Returns True if advisory message object is disconnected, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isRevalidating

Returns mqboolean value representing the revalidation status of the advisory message object.

Declaration

```
mqboolean isRevalidating()
throw (CJMSException *);
```

Returns

Returns TRUE if revalidating thread is active, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isFailed

Returns mqboolean value representing the failure staus of advisory message thread.

Declaration

```
mqboolean isFailed()
throw (CJMSException *);
```

Returns

Returns TRUE if is failed, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isTransferring

Returns mqboolean value if the advisory message thread is transferring data.

Declaration

```
mqboolean isTransferring()
throw (CJMSException *);
```

Returns

Returns TRUE if is transferring, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

isTransferComplete

Returns mqboolean value representing the transfer status.

Declaration

```
mqboolean isTransferComplete()
throw (CJMSException *);
```

Returns

Returns TRUE if the transfer is complete, FALSE otherwise.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

CAdvisoryMsgListener

A CAdvisoryMsgListener object is used to receive asynchronously delivered advisory messages when the reconnection thread is active.

onAdvisoryMessage

Passes a message to the listener.

Declaration

```
virtual void onAdvisoryMessage(CAdvisoryMessage *msg) = 0;
```

Parameters

msg

The message passed to the listener

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

CByteMessage

A CBytesMessage object is used to send a message containing a stream of uninterpreted bytes. It inherits from the CMessageinterface and adds a bytes message body. The receiver of the message supplies the interpretation of the bytes.

Derives

CMessage

getBodyLength

Gets the number of bytes of the message body when the message is in read-only mode. The value returned can be used to allocate a byte array. The value returned is the entire length of the message body, regardless of where the pointer for reading the message is currently located.

Declaration

```
mqlong getBodyLength() FMQCONST
throw (CJMSException  *);
```

Returns

Number of bytes in the message.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readBoolean

Reads a boolean from the bytes message stream.

Declaration

```
mqboolean readBoolean() FMQCONST
throw (CJMSException  *);
```

Returns

The boolean value read

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readByte

Reads a signed 8-bit value from the bytes message stream.

Declaration

```
mqbyte readByte() FMQCONST
throw (CJMSException  *);
```

Returns

The next byte from the bytes message stream as a signed 8-bit byte

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readBytes

Reads a byte array from the bytes message stream.If the length of array value is less than the number of bytes remaining to be read from the stream, the array should be filled. A subsequent call reads the next increment, and so on. If the number of bytes remaining in the stream is less than the length of array value, the bytes should be read into the array. The return value of the total number of bytes read will be less than the length of the array, indicating that there are no more bytes left to be read from the stream. The next read of the stream returns -1.

Declaration

```
mqint readBytes(mqbyteArray value, int length) FMQCONST
throw (CJMSException  *);
```

Parameters

value

The buffer into which the data is read.

Length

The length of the buffer.

Returns

The total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readChar

Reads a Unicode character value from the bytes message stream.

Declaration

```
mqchar readChar() FMQCONST
 throw (CJMSException  *);
```

Returns

The next two bytes from the bytes message stream as a Unicode character

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readDouble

Reads a double from the bytes message stream.

Declaration

```
mqdouble readDouble() FMQCONST
                            throw (CJMSException  *);
```

Returns

The next eight bytes from the bytes message stream, interpreted as a double.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readFloat

Reads a float from the bytes message stream.

Declaration

```
mqfloat readFloat() FMQCONST
                              throw (CJMSException  *);
```

Returns

The next four bytes from the bytes message stream, interpreted as a float

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readInt

Reads a signed 32-bit integer from the bytes message stream.

Declaration

```
mqint readInt() FMQCONST
                              throw (CJMSException  *);
```

Returns

The next four bytes from the bytes message stream, interpreted as an int

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readLong

Reads a signed 64-bit integer from the bytes message stream.

Declaration

```
mqlong readLong() FMQCONST
                              throw (CJMSException  *);
```

Returns

The next eight bytes from the bytes message stream, interpreted as a long.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readShort

Reads a signed 16-bit number from the bytes message stream.

Declaration

```
mqshort readShort() FMQCONST
                            throw (CJMSException  *);
```

Returns

The next two bytes from the bytes message stream, interpreted as a signed 16-bit number

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readUnsignedByte

Reads an unsigned 8-bit number from the bytes message stream.

Declaration

```
mqint readUnsignedByte() FMQCONST
                            throw (CJMSException  *);
```

Returns

The next byte from the bytes message stream, interpreted as an unsigned 8-bit number

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readUnsignedShort

Reads an unsigned 16-bit number from the bytes message stream.

Declaration

```
mqint readUnsignedShort() FMQCONST
                            throw (CJMSException  *);
```

Returns

The next two bytes from the bytes message stream, interpreted as an unsigned 16-bit integer.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

readUTF

Reads a mqcstring that has been encoded using a modified UTF-8 format from the bytesmessage stream.

For more information on the UTF-8 format, see "File System Safe UCS Transformation Format (FSS_UTF)", X/Open Preliminary Specification, X/Open Company Ltd., Document Number: P316. This information also appears in ISO/IEC 10646, Annex P.

Declaration

```
mqcstring readUTF() FMQCONST
                            throw (CJMSException  *);
```

Returns

A Unicode mqcstring from the bytes message stream

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

reset

Puts the message body in read-only mode and repositions the stream of bytes to the beginning.

Declaration

```
void reset()
                            throw (CJMSException  *);
```

Throws

CJMSException

If the JMS provider fails to reset the message due to some internal error.

writeBoolean

Writes a boolean to the bytes message stream as a 1-byte value. The value true is written as the value (byte)1; the value false is written as the value (byte)0.

Declaration

```
void writeBoolean(mqboolean value)
                            throw (CJMSException  *);
```

Parameters

value

The boolean value to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some

internal error.

writeByte

Writes a byte to the bytes message stream as a 1-byte value.

Declaration

```
void writeByte(mqbyte value)
                            throw (CJMSException  *);
```

Parameters:

value

The byte value to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeBytes

Writes a portion of a byte array to the bytes message stream.

Declaration

```
void writeBytes(mqbyteArray value, mqint offset, mqint length)
                            throw (CJMSException  *);
```

Parameters

value

The byte array value to be written
offset

The initial offset within the byte array
length

The number of bytes to use

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeChar

Writes a char to the bytes message stream as a 2-byte value, high byte first.

Declaration

```
void writeChar(mqchar value)
                            throw (CJMSException  *);
```

Parameters

value

The char value to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeDouble

Reads a double from the bytes message stream.

Declaration

```
void writeDouble(mqdouble value)
                            throw (CJMSException  *);
```

Returns

The next eight bytes from the bytes message stream, interpreted as a double

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

writeFloat

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the bytes message stream as a 4-byte quantity, high byte first.

Declaration

```
void writeFloat(mqfloat value)
                            throw (CJMSException  *);
```

Parameters

value

The float value to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeInt

Writes an int to the bytes message stream as four bytes, high byte first.

Declaration

```
void writeInt(mqint value)
                            throw (CJMSException  *);
```

Parameters

value

The int to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeLong

Writes a long to the bytes message stream as eight bytes, high byte first.

Declaration

```
void writeLong(mqlong value)
throw (CJMSException  *);
```

Parameters

value

The long to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeShort

Writes a short to the bytes message stream as two bytes, high byte first.

Declaration

```
void writeShort(mqshort value)
                            throw (CJMSException  *);
```

Parameters

value

The short to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

writeUTF

Writes a mqcstring to the bytes message stream using UTF-8 encoding in a machine-independent manner.

For more information on the UTF-8 format, refer "File System Safe UCS Transformation Format (FSS_UTF)", X/Open Preliminary Specification, X/OpenCompany Ltd., Document Number: P316. This information also appears in ISO/IEC 10646, Annexure P.

Declaration

```
void writeUTF(mqcstring value)
                            throw (CJMSException  *);
```

Parameters

value

The String value to be written

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

CConnection

A CConnection object is a client's active connection to its JMS provider.

Base for

CQueueConnection CTopicConnection

CConnectionConsumer

For application servers, CConnection objects provide a special facility for creating a ConnectionConsumer.

CConnectionFactory

A CConnectionFactory object encapsulates a set of connection configuration parameters that has been defined by an administrator. A client uses it to create a connection with a JMS provider.

Base for

CTopicConnectionFactory CQueueConnectionFactory

CDestination

A CDestination object encapsulates a provider-specific address.

Base for

Ctopic CQueue CTemporaryTopic CTemporaryQueue

CFioranoException

This is the root class of all exceptions.

public

checkForException

Utility Function that throws CJMSException if an exception had occurred.

Declaration

```
static void checkForException()
                            throw (CJMSException *);
```

CJMSException

Constructor

Constructs a CJMSException with the specified reason and with the error code.

Declaration

```
CJMSException(mqcstring errCode);
```

Parameters

errCode

The error code that identifies the error

Constructor

Constructs a CJMSException with the specified reason and with the error code and description.

Declaration

```
CJMSException(mqcstring errCode, mqcstring errDesc);
```

Parameters

errCode

The error code that identifies the error
errDesc

description of the error

checkForException

Utility Function that throws CJMSException if an exception had occurred.

Declaration

```
static void static void checkForException(mqcstring errCode);
                        throw (CJMSException *);
```

Parameters

errCode

The error code that identifies the error

Another over-loaded function

Utility Function that throws CJMSException if an exception had occurred.

Declaration

```
static void checkForException(mqcstring errCode, mqcstring errDesc)
                        throw (CJMSException *);
```

Parameters

errCode

The error code that identifies the error
errDesc

of the error

printStackTrace

Prints the Function stack trace on the console

Declaration

```
void printStackTrace();
```

getStackTrace

Gets the stack trace of the function

Declaration

```
const DN_STRING getStackTrace();
```

getErrorCode

Gets the vendor-specific error code.

Declaration

```
const mqcstring getErrorCode();
```

Returns

The vendor-specific error code

getLinkedException

Gets the exception linked to this one

Declaration

```
CJMSException *getLinkedException();
```

Returns

The linked exception

CExceptionListener

If FioranoMQ cpp detects a serious problem with a CConnection object, it informs the CConnection object's CExceptionListener, if one has been registered. It does this by calling the listener's onException method, passing it a CJMSException argument describing the problem.

onException

Notifies the user of a JMS exception. The user is expected to override this function with the required functionality

Declaration

```
virtual void onException(CFioranoException *msg) = 0;
```

Parameters

msg

Message handle

CMapMessage

The CMapMessage object is used to send a set of name-value pairs. The names must have a value that is not null, and not an empty mqcstring. The entries can be accessed sequentially or randomly by name. CMapMessage inherits from the CMessage interface and adds a message body that contains a Map.

Derives

CMessage

getBoolean

Returns the boolean value with the specified name.

Declaration

```
mqboolean getBoolean(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the boolean

Returns

The boolean value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getByte

Returns the byte value with the specified name.

Declaration

```
mqbyte getByte(mqcstring name) FMQCONST
                              throw (CJMSException *);
```

Parameters

name

The name of the byte

Returns

The byte value with the specified name.

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getChar

Returns the Unicode character value with the specified name.

Declaration

```
mqchar getChar(mqcstring name) FMQCONST
                              throw (CJMSException *);
```

Parameters

name

The name of the Unicode character

Returns

The Unicode character value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getDouble

Returns the double value with the specified name.

Declaration

```
mqdouble getDouble(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the double

Returns

The double value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getFloat

Returns the float value with the specified name.

Declaration

```
mqfloat getFloat(mqcstring name)FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the float

Returns

The float value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getInt

Returns the int value with the specified name.

Declaration

```
mqint getInt(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the int

Returns

The int value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getLong

Returns the long value with the specified name.

Declaration

```
mqlong getLong(mqcstring name) FMQCONST
                              throw (CJMSException *);
```

Parameters

name

The name of the long

Returns

The long value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getMapNames

Returns a pointer to mqcstrings of all the names in the MapMessage object.

Declaration

```
mqcstring *getMapNames() FMQCONST
                              throw (CJMSException *);
```

Returns

An a pointer to all the names (mqcstring) in this MapMessage

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getMapNamesHTEnum

Returns an Enumeration of all the names in the MapMessage object.

Declaration

```
CHashTableEnumerator *getMapNamesHTEnum() FMQCONST
                            throw (CJMSException *);
```

Returns

An enumeration of all the names in this MapMessage

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getShort

Returns the short value with the specified name.

Declaration

```
mqshort getShort(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the short

Returns

The short value with the specified name

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

getString

Returns the String value with the specified name.

Declaration

```
mqcstring getString(mqcstring name) FMQCONST
                            throws CJMSException;
```

Parameters

name

The name of the String

Returns

The String value with the specified name; if there is no item by this name, a null value is returned

Throws

CJMSException

If the JMS provider fails to read the message due to some internal error.

itemExists

Indicates whether an item exists in this MapMessage object.

Declaration

```
mqboolean itemExists(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the item to test

Returns

TRUE if the item exists

Throws

CJMSException

If the JMS provider fails to determine if the item exists due to some internal error.

setBoolean

Sets a boolean value with the specified name into the Map.

Declaration

```
void setBoolean(mqcstring name, mqboolean value)
                            throw (CJMSException *);
```

Parameters

name

The name of the boolean

value

The boolean value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setByte

Sets a byte value with the specified name into the Map.

Declaration

```
void setByte(mqcstring name, mqbyte value)
                            throw (CJMSException *);
```

Parameters

name

The name of the byte

value

The byte value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setBytes

Sets a byte array value with the specified name into the Map.

Declaration

```
void setBytes(mqcstring name, mqbyteArray value)
                            throw (CJMSException *);
```

Parameters

name

The name of the byte array

value

The byte array value to set in the Map; the array is copied so that the value for name will not be altered by future modifications

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setBytes

Sets a portion of the byte array value with the specified name into the Map.

Declaration

```
void setBytes(mqcstring name, mqbyteArray value, mqint offset, mqint length)
                            throw (CJMSException *);
```

Parameters

name

The name of the byte array
value

The byte array value to set in the Map
offset

The initial offset within the byte array
length

The number of bytes to use

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setChar

Sets a Unicode character value with the specified name into the Map.

Declaration

```
void setChar(mqcstring name, mqchar value)
                            throw (CJMSException *);
```

Parameters

name

The name of the Unicode character
value

The Unicode character value to set in the Map

Throws

MSException

If the JMS provider fails to write the message due to some internal error.

setDouble

Sets a double value with the specified name into the Map.

Declaration

```
void setDouble(mqcstring name, mqdouble value)
                              throw (CJMSException *);
```

Parameters

name

The name of the double
value

The double value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setFloat

Sets a float value with the specified name into the Map.

Declaration

```
void setFloat(mqcstring name, mqfloat value)
                              throw (CJMSException *);
```

Parameters

name

The name of the float
value

The float value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setInt

Sets an int value with the specified name into the Map.

Declaration

```
void setInt(mqcstring name, mqint value)
                            throw (CJMSException *);
```

Parameters

name

The name of the int
value

The int value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setLong

Sets a long value with the specified name into the Map.

Declaration

```
void setLong(mqcstring name, mqlong value)
                            throw (CJMSException *);
```

Parameters

name

The name of the long
value

The long value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setShort

Sets a short value with the specified name into the Map.

Declaration

```
void setShort(mqcstring name, mqshort value)
                            throw (CJMSException *);
```

Parameters

name

The name of the short
value

The short value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

setString

Sets a String value with the specified name into the Map.

Declaration

```
void setString(mqcstring name, mqcstring value)
                            throw (CJMSException *);
```

Parameters

name

The name of the String
value

The String value to set in the Map

Throws

CJMSException

If the JMS provider fails to write the message due to some internal error.

CMessage

The CMessage interface is the root interface of all JMS messages. It defines the message header and the acknowledge method used for all messages.

Base for

CTextMessage CBytesMessage CMapMessage CObjectMessage

and

CStreamMessage

Acknowledge

Acknowledges all consumed messages of the session of this consumed message. All consumed JMS messages support the acknowledge method for use when a client has specified that its JMS session's consumed messages are to be explicitly acknowledged. By invoking acknowledge on a consumed message, a client acknowledges all messages consumed by the session that the message was delivered.

Calls to acknowledge are ignored for both transacted sessions and sessions specified to use implicit acknowledgement modes. A client may individually acknowledge each message as it is consumed, or it may choose to acknowledge messages as an application-defined group (which is done by calling acknowledge on the last received message of the group, thereby acknowledging all messages consumed by the session.) Messages that have been received but not acknowledged may be redelivered.

Declaration

```
void acknowledge()
                        throw(CJMSException *);
```

Throws

CJMSException

If the JMS provider fails to acknowledge the messages due to some internal error.

ClearBody

Clears out the message body. Clearing a message's body does not clear its header values or property entries. If this message body was read-only, calling this method leaves the message body in the same state as an empty body in a newly created message.

Declaration

```
void clearBody()
                        throw (CJMSException *);
```

Throws

CJMSException

If the JMS provider fails to clear the message body due to some internal error.

clearProperties

Clears a message's properties. The message's header fields and body are not cleared.

Declaration

```
void clearProperties()
                            throw (CJMSException *);
```

Throws

CJMSException

If the JMS provider fails to clear the message properties due to some internal error.

getBooleanProperty

Returns the value of the boolean property with the specified name.

Declaration

```
mqboolean getBooleanProperty(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the boolean property

Returns

The boolean property value for the specified name

Throws

CJMSException

If the JMS provider fails to get the property value due to some internal error.

getByteProperty

Returns the value of the byte property with the specified name.

Declaration

```
mqbyte getByteProperty(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the byte property

Returns

The byte property value for the specified name

Throws

CJMSException

If the JMS provider fails to get the property value due to some internal error.

getDoubleProperty

Returns the value of the String property with the specified name.

Declaration

```
mqdouble getDoubleProperty(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the String property

Returns

The String property value for the specified name. If there is no property by this name, a null value is returned

Throws

CJMSException

If the JMS provider fails to get the property value due to some internal error.

getFloatProperty

Returns the value of the double property with the specified name.

Declaration

```
mqfloat getFloatProperty(mqcstring name)FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the double property

Returns

The double property value for the specified name

Throws

CJMSException

If the JMS provider fails to get the property value due to some internal error.

getIntProperty

Description

Returns the value of the int property with the specified name.

Declaration

```
mqint getIntProperty(mqcstring name) FMQCONST
                            throw (CJMSException *);
```

Parameters

name

The name of the int property

Returns

The int property value for the specified name

Throws

CJMSException

If the JMS provider fails to get the property value due to some internal error.

getJMSCorrelationID

Gets the correlation ID for the message. This method is used to return correlation ID values that are either provider-specific message IDs or application-specific String values.

Declaration

```
mqcstring getJMSCorrelationID() FMQCONST
                            throw (CJMSException *);
```

Returns

The correlation ID of a message as a String

Throws

CJMSException

If the JMS provider fails to get the correlation ID due to some internal error.

getJMSCorrelationIDAsBytes

Gets the correlation ID as an array of bytes for the message. The use of a byte[] value for JMSCorrelationID is non-portable.

Returns

The correlation ID of a message as an array of bytes

Throws

JMSException

If the JMS provider fails to get the correlation ID due to some internal error.

Declaration

```
mqcstring getJMSCorrelationIDAsBytes() FMQCONST
                            throw (CJMSException *);
```

getJMSDeliveryMode

Gets the DeliveryMode value specified for this message.

Declaration

```
mqint getJMSDeliveryMode() FMQCONST
                            throw (CJMSException *);
```

Returns

The delivery mode for this message

Throws

CJMSException

If the JMS provider fails to get the delivery mode due to some internal error.

getJMSDestination

Gets the Destination object for this message. The JMSDestination header field contains the destination to which the message is being sent. When a message is sent, this field is ignored. After completion of the send or publish method, the field holds the destination specified by the method. When a message is received, its JMSDestination value must be equivalent to the value assigned when it was sent.

Declaration

```
CDestination *getJMSDestination() FMQCONST
                            throw (CJMSException *);
```

Returns

The destination of this message

Throws

CJMSException

If the JMS provider fails to get the destination due tosome internal error.

getJMSExpiration

Gets the message's expiration value. When a message is sent, the JMSExpiration header field is left unassigned. After completion of the send or publish method, it holds the expiration time of the message. This is the sum of the time-to-live value specified by the client and the GMT at the time of the send or publish. If the time-to-live is specified as zero, JMSExpiration is set to zero to indicate that the message does not expire. When a message's expiration time is reached, a provider should discard it.The JMS API does not define any form of notification of message expiration. Clients should not receive messages that have expired; however, the JMS API does not guarantee that this will not happen.

Declaration

```
mqlong getJMSExpiration() FMQCONST
                              throw (CJMSException *);
```

Returns

The time the message expires, which is the sum of the time-to-live value specified by the client and the GMT at the time of the send

Throws

CJMSException

If the JMS provider fails to get the message expiration due to some internal error.

getJMSMessageID

Gets the message ID. The JMSMessageID header field contains a value that uniquely identifies each message sent by a provider. When a message is sent, JMSMessageID can be ignored. When the send or publish method returns, it contains a provider-assigned value. A JMSMessageID is a String value that should function as a unique key for identifying messages in a historical repository. The exact scope of uniqueness is provider-defined. It should at least cover all messages for a specific installation of a provider, where an installation is some connected set of message routers. All JMSMessageID values must start with the prefix 'ID:'. Uniqueness of message ID values across different providers is not required. Since message IDs take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that the messageID is not used by an application. By calling the MessageProducer. setDisableMessageID method, a JMS client enables this potential optimization for all messages sent by that message producer. If the JMS provider accepts this hint, these messages must have the message ID set to null. If the provider ignores the hint, the message ID must be set to its normal unique value.

Declaration

```
mqcstring getJMSMessageID() FMQCONST
                              throw (CJMSException *);
```

Returns

The message ID

Throws

CJMSException

If the JMS provider fails to get the message ID due to someinternal error.

getJMSPriority

Gets the message priority level. The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. In addition, clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. The JMS API does not require that a provider strictly implement priority ordering of messages; however, it should do its best to deliver expedited messages ahead of normal messages.

Declaration

```
mqint getJMSPriority() FMQCONST
                            throw (CJMSException *);
```

Returns

The default message priority

Throws

CJMSException

If the JMS provider fails to get the message priority due to some internal error.

getJMSRedelivered

Gets an indication of whether this message is being redelivered. If a client receives a message with the JMSRedelivered field set. It is likely, but not guaranteed, that this message was delivered earlier but that its receipt was not acknowledged at that time.

Declaration

```
mqboolean getJMSRedelivered() FMQCONST
                            throw (CJMSException *);
```

Returns

TRUE if this message is being redelivered

Throws

CJMSException

If the JMS provider fails to get the redelivered state due to some internal error.

getJMSReplyTo

Gets the Destination object to which a reply to this message should besent.

Declaration

```
CDestination *getJMSReplyTo() FMQCONST
                            throw (CJMSException *);
```

Returns

Destination to which to send a response to this message

Throws

CJMSException

If the JMS provider fails to get the JMSReplyTo destination due to some internal error.

getJMSTimestamp

Gets the message timestamp. The JMSTimestamp header field contains the time when the message was handed off to a provider to be sent. It is not the time when the message was actually transmitted, because the actual transmission may occur later due to transactions or other client-side queueing of messages. When a message is sent, JMSTimestamp is ignored.

Declaration

```
mqlong getJMSTimestamp() FMQCONST
                            throw (CJMSException *);
```

Returns

The timestamp of the message

Throws

CJMSException

If the JMS provider fails to get the timestamp due to someinternal error.

getJMSType

Gets the message type identifier supplied by the client when the message was sent.

Declaration

```
mqcstring getJMSType() FMQCONST
                            throw (CJMSException *);
```

Returns

The message type

Throws

CJMSException

If the JMS provider fails to get the message type due to some internal error.

getLongProperty

Returns the value of the long property with the specified name.