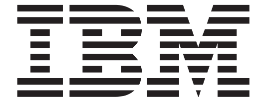
WebSphere Business Integration



Using Multi-Language Message Service

Version 1.0

SC34-6363-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices,” on page 243.

Second edition (July 2004)

This edition applies to IBM Multi-Language Message Service, Version 1.0 and to any subsequent releases and modifications until otherwise indicated in new editions.

Parts of the specification of Multi-Language Message Service are derived from the following sources:

The Java Message Service Specification, Version 1.1
Copyright 2002 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303, U.S.A.
All rights reserved.

Package javax.jms (JMS 1.1 API specification)
Copyright 2002 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303, U.S.A.
All rights reserved.

© Copyright International Business Machines Corporation 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

Tables vii

About this book ix
Who this book is for ix
What you need to know to understand this book . . ix
Terms used in this book ix
How to use this book x

Part 1. Getting started with XMS . . . 1

Chapter 1. Introducing XMS 3
What is Multi-Language Message Service? 3
Styles of messaging 3
Messaging transports 4
The XMS object model 5
Attributes and properties of objects 6
The first release of XMS 7
Supported operating environments 7
Function supported 7
Function not supported 8

Chapter 2. Installing XMS 9
Installing the XMS client 9
What is installed on Linux 10
What is installed on Windows 11
Uninstalling the XMS client 12
Uninstalling the XMS client on Windows using Add/Remove Programs 13

Chapter 3. Using XMS 15
The sample applications 15
The C sample applications 15
The C++ sample applications 16
The WebSphere MQ JMS sample applications . . 16
Running the C sample applications 17
Building the sample applications 18
Building your own applications 18
Building your own C applications 18
Building your own C++ applications 19
Problem determination 19

Part 2. Programming with XMS . . . 21

Chapter 4. Writing XMS applications . . 23
The threading model 23
Connections 23
Starting and stopping a connection 24
Closing a connection 24
Handling exceptions 24
Sessions 25
Asynchronous message delivery 25
Synchronous message delivery 25
Uniform resource identifiers (URIs) 26
Deleting objects 26
Setting the value of a property 27
Iterators 27
Coded character set identifiers (CCSIDs). 28

Chapter 5. Writing XMS applications in C 31
Object handles in C. 31
Using message and exception listener functions in C 31
Using message listener functions in C 31
Using exception listener functions in C 32
C functions that return a string or byte array by value 33
C functions that return a string or byte array by reference 34
Handling errors in C 34
Return codes 34
The error block 35

Chapter 6. Writing XMS applications in C++ 37
Using namespaces in C++ 37
Using the String class in C++ 38
C++ methods that return a byte array 38
Using the PropertyContext class in C++ 39
Handling errors in C++ 39
Using message and exception listeners in C++. . . 41
Using message listeners in C++. 41
Using exception listeners in C++ 43
Assigning XMS objects in C++ 43
Using the C API in a C++ application 45

Chapter 7. XMS messages 47
Header fields in an XMS message 47
Properties of an XMS message 48
JMS defined properties of a message 49
IBM defined properties of a message 49
Application defined properties of a message . . 50
The body of an XMS message 50
Bytes messages 52
Map messages 52
Message selectors 53

Part 3. XMS API reference 55

Chapter 8. XMS classes 59
BytesMessage. 61
Methods 61
Inherited methods in the C++ interface 75
Connection 76
Methods 76
Inherited methods in the C++ interface 81

© Copyright IBM Corp. 2004

ConnectionFactory . . . 83
Constructor . . . 83
Methods . . . 83
Inherited methods in the C++ interface . . . 91
ConnectionMetaData . . . 92
Methods . . . 92
Inherited methods in the C++ interface . . . 95
Destination . . . 96
Constructors . . . 96
Methods . . . 97
Inherited methods in the C++ interface . . . 104
ErrorBlock . . . 105
Methods . . . 105
Exception . . . 109
Methods . . . 109
ExceptionListener . . . 113
Methods . . . 113
IllegalStateException . . . 114
Inherited methods . . . 114
InvalidDestinationException . . . 115
Inherited methods . . . 115
InvalidSelectorException . . . 116
Inherited methods . . . 116
Iterator . . . 117
Methods . . . 117
MapMessage . . . 120
Methods . . . 120
Inherited methods in the C++ interface . . . 137
Message . . . 139
Methods . . . 139
Inherited methods in the C++ interface . . . 168
MessageConsumer . . . 169
Methods . . . 169
Inherited methods in the C++ interface . . . 175
MessageEOFException . . . 176
Inherited methods . . . 176
MessageFormatException . . . 177
Inherited methods . . . 177
MessageListener . . . 178
Methods . . . 178
MessageNotReadableException . . . 179
Inherited methods . . . 179
MessageNotWritableException . . . 180
Inherited methods . . . 180
MessageProducer . . . 181
Methods . . . 181
Inherited methods in the C++ interface . . . 192
Property . . . 193
Constructors . . . 193
Methods . . . 195
PropertyContext . . . 212
Methods . . . 212
ResourceAllocationException . . . 221
Inherited methods . . . 221
SecurityException . . . 222
Inherited methods . . . 222
Session . . . 223
Methods . . . 223
Inherited methods in the C++ interface . . . 231
String . . . 232
Constructors . . . 232
Methods . . . 233

Chapter 9. Properties of XMS objects 237

Properties of Connection . . . 237
Properties of ConnectionFactory . . . 237
Properties of ConnectionMetaData . . . 238
Properties of Destination . . . 239
Properties of MessageConsumer . . . 240
Properties of Session . . . 241

Appendix. Notices . . . 243

Trademarks . . . 244

Index . . . 247

Sending your comments to IBM . . . 251

Figures

1. XMS objects and their relationships 6

© Copyright IBM Corp. 2004

Tables

1. XMS client platforms and compilers 7
2. Installed directories on Linux and their contents 10
3. Installed directories on Windows and their contents 11
4. The names and valid values of properties that you can use in a topic URI 26
5. Objects that are deleted automatically 27
6. Object handle data types 31
7. Return codes from C function calls 34
8. The XMS classes on which the assignment operator is overloaded 43
9. JMS message header fields 47
10. JMS defined properties of a message 49
11. IBM defined properties of a message 49
12. XMS data types that are compatible with Java data types 51
13. A summary of the XMS classes 59
14. Exception codes and their corresponding C++ exceptions 60
15. Properties of Connection 237
16. Properties of ConnectionFactory 237
17. Properties of ConnectionMetaData 239
18. Properties of Destination 240
19. Properties of MessageConsumer 241
20. Properties of Session 241

© Copyright IBM Corp. 2004

About this book

This book is about IBM® Multi-Language Message Service (XMS), Version 1.0. The book has the following parts:

- Part 1, "Getting started with XMS," on page 1, which describes what XMS is, and how to install and use XMS
- Part 2, "Programming with XMS," on page 21, which describes how to write XMS applications
- Part 3, "XMS API reference," on page 55, which documents the XMS classes and their methods, and the properties of XMS objects

For the latest information about XMS, see the product readme.txt file, which is in the zipped file supplied with XMS.

Who this book is for

This book is primarily for application programmers who write XMS applications. Some of the information might also be useful to system administrators who manage systems on which XMS applications run, or who manage WebSphere® Business Integration Event Broker or WebSphere Business Integration Message Broker brokers to which XMS applications connect.

What you need to know to understand this book

To understand this book, you need the following skills, knowledge, and experience:

- C or C++ application programming skills. If you are not a C++ programmer, you need some knowledge of object oriented concepts and terminology.
- A working knowledge of the operating system that you are using.
- Experience in using TCP/IP as a communications protocol.
- Some knowledge of the concepts and terminology associated with WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker.
- Some knowledge of the *Java™ Message Service Specification, Version 1.1* and the WebSphere MQ implementation of JMS, WebSphere MQ classes for Java Message Service, might be beneficial, but is not absolutely necessary. You do not need to be a Java or JMS application programmer.

Terms used in this book

The term *XMS* is used as an abbreviation for Multi-Language Message Service.

The term *JMS* means Java Message Service.

The term *WebSphere MQ JMS* means WebSphere MQ classes for Java Message Service.

The term *Linux®* means SUSE LINUX Enterprise Server.

The term *Windows®* means Windows XP.

© Copyright IBM Corp. 2004

How to use this book

Certain sections in this book refer you to *WebSphere MQ Using Java* for more information. You can download the latest edition of *WebSphere MQ Using Java* from http://www.ibm.com/software/integration/mqfamily/library/manualsa/.

Part 1. Getting started with XMS

Chapter 1. Introducing XMS 3
What is Multi-Language Message Service? 3
Styles of messaging 3
Messaging transports 4
The XMS object model 5
Attributes and properties of objects 6
The first release of XMS 7
Supported operating environments 7
Function supported 7
Function not supported 8

Chapter 2. Installing XMS 9
Installing the XMS client 9
What is installed on Linux 10
What is installed on Windows 11
Uninstalling the XMS client 12
Uninstalling the XMS client on Windows using Add/Remove Programs 13

Chapter 3. Using XMS 15
The sample applications 15
The C sample applications 15
The C++ sample applications 16
The WebSphere MQ JMS sample applications . . 16
Running the C sample applications 17
Building the sample applications 18
Building your own applications 18
Building your own C applications 18
Building your own C++ applications 19
Problem determination 19

© Copyright IBM Corp. 2004

Chapter 1. Introducing XMS

This chapter introduces Multi-Language Message Service. The chapter contains the following sections:

- "What is Multi-Language Message Service?"
- "Styles of messaging"
- "Messaging transports" on page 4
- "The XMS object model" on page 5
- "The first release of XMS" on page 7

What is Multi-Language Message Service?

Multi-Language Message Service (XMS) is an application programming interface (API) that is based on the Java Message Service (JMS) API. With the first release of XMS, you can write XMS applications that use the publish/subscribe style of messaging. An XMS application acts as a client application to a broker of WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker, and uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport to communicate with the broker. You can write XMS applications in either the C or C++ programming language.

Styles of messaging

Point-to-point and *publish/subscribe* are two styles of messaging. Styles of messaging are also called *messaging domains*.

Point-to-point messaging

A common form of point-to-point messaging uses queuing. In the simplest case, an application sends a message to another application by identifying, implicitly or explicitly, a destination queue. The underlying messaging and queuing system receives the message from the sending application and routes the message to its destination queue. The receiving application can then retrieve the message from the queue.

If the underlying messaging and queuing system contains a message broker, the broker might replicate a message and route copies of the message to different queues so that more than one application can receive the message. The broker might also transform a message and add data to it.

A key characteristic of point-to-point messaging is that an application identifies a destination queue when it sends a message. The configuration of the underlying messaging and queuing system then determines precisely which queue the message is put on so that it can be retrieved by the receiving application.

Publish/subscribe messaging

In publish/subscribe messaging, there are two types of application: publisher and subscriber.

A *publisher* supplies information in the form of messages. When a publisher publishes a message, it specifies a topic, which identifies the subject of the information inside the message.

© Copyright IBM Corp. 2004

A *subscriber* is a consumer of the information that is published. A subscriber specifies the topics it is interested in by sending subscription requests to a publish/subscribe broker. The broker receives published messages from publishers and subscription requests from subscribers, and routes published messages to subscribers. A subscriber receives messages on all topics, and only those topics, to which it has subscribed.

A key characteristic of publish/subscribe messaging is that a publisher identifies a topic when it publishes a message, and a subscriber receives the message only if has subscribed to the topic. If a message is published on a topic for which there are no subscribers, no application receives the message.

An application can be both a publisher and a subscriber.

Note: The first release of XMS supports only the publish/subscribe messaging domain.

Messaging transports

A *messaging transport* is a way in which an application can exchange messages with a broker.

XMS supports two brokers for the publish/subscribe domain:

- WebSphere Business Integration Event Broker
- WebSphere Business Integration Message Broker

Each broker provides the following three transports:

WebSphere MQ Enterprise Transport
All communication between a publisher and a broker, or between a subscriber and a broker, uses WebSphere MQ.

If a publisher uses this transport, the publisher publishes messages by putting them on a queue that is monitored by the broker.

If a subscriber uses this transport, the subscriber sends subscription requests to the broker by putting messages on a queue that is monitored by the broker. In turn, the broker puts messages, published on topics to which the subscriber has subscribed, on a queue that is monitored by the subscriber.

WebSphere MQ Real-Time Transport
All communication between a publisher and a broker, or between a subscriber and a broker, uses a TCP connection.

If a publisher uses this transport, the publisher publishes messages by sending them directly to the broker over a TCP connection.

If a subscriber uses this transport, the subscriber sends subscription requests directly to the broker over a TCP connection. In turn, the broker sends messages, published on topics to which the subscriber has subscribed, directly to the subscriber over a TCP connection.

WebSphere MQ Multicast Transport
A subscriber can use this transport to receive published messages from a broker. The transport cannot be used for any other purpose.

The transport works by associating a multicast IP address with a topic. When a message is published, the broker transmits one copy of the

message to the multicast IP address associated with the topic. IP then routes the message to all subscribers that have subscribed to the topic.

WebSphere MQ Multicast Transport is a high performance transport. Using this transport, a broker transmits only one copy of a published message over the network. Using WebSphere MQ Real-Time Transport, by comparison, a broker must transmit a copy of a published message to each subscriber.

Note: The first release of XMS supports only WebSphere MQ Real-Time Transport and WebSphere MQ Multicast Transport.

The XMS object model

The XMS API is an object oriented interface. The XMS object model is based on the JMS 1.1 object model.

The following list summarizes the main XMS classes, or types of object:

ConnectionFactory
A ConnectionFactory object encapsulates a set of configuration parameters for a connection. An application uses a connection factory to create a connection.

Connection
A Connection object encapsulates an application's active connection to a broker. An application uses a connection to create sessions.

Destination
A destination is where an application sends messages, or it is a source from which an application receives messages, or both. In the publish/subscribe domain, a Destination object encapsulates a topic.

Session
A session is a single threaded context for sending and receiving messages. An application uses a session to create messages, message producers, and message consumers.

Message
A Message object encapsulates a message that an application sends or receives.

MessageProducer
An application uses a message producer to send messages to a destination.

MessageConsumer
An application uses a message consumer to receive messages sent to a destination.

Figure 1 on page 6 shows these objects and their relationships.

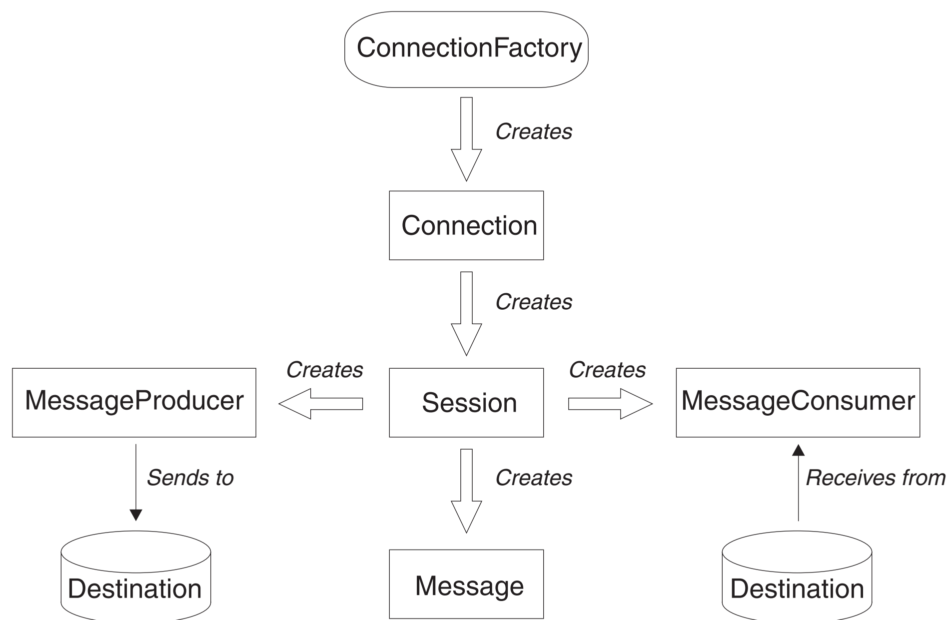


Figure 1. XMS objects and their relationships

XMS applications written in C++ use these classes and their methods. XMS applications written in C use the same object model even though C is not an object oriented language. When a C application creates an object, XMS stores the object internally and returns a handle for the object to the application. The application can then use the handle subsequently to reference the object. For example, if a C application creates a connection factory, XMS returns a handle for the connection factory to the application. In general, for each C++ method in the C++ interface, there is an equivalent C function in the C interface.

The XMS object model is based upon the domain independent interfaces that are described in the *Java Message Service Specification, Version 1.1*. Domain specific classes, such as Topic, TopicPublisher, and TopicSubscriber, are not provided.

Attributes and properties of objects

An XMS object can have attributes and properties, which are characteristics of the object. Attributes and properties, however, are implemented in different ways.

Attributes
An attribute of an object is always present and occupies storage, even if the attribute does not have a value. In this respect, an attribute is similar in concept to a field in a fixed length data structure. Another distinguishing feature is that each attribute has its own methods for setting and getting its value.

Properties
A property of an object is present and occupies storage only after its value is set. However, a property cannot be deleted, and the storage recovered, after its value has been set, although you can change its value. A property does not have its own methods for setting and getting its value. Instead, XMS provides a set of generic methods for setting and getting the values of properties.

The first release of XMS

This section specifies the supported operating environments for the first release of XMS. It also summarizes the function supported in the first release, and the function that is not supported. If you need an explanation of any of the function mentioned in this section, see the following sources of information:

- The WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker Information Center.
- *WebSphere MQ Using Java*
- *Java Message Service Specification, Version 1.1*

Supported operating environments

An XMS client is supplied for each of the operating systems listed in Table 1. The table also lists the supported C and C++ compiler for each client platform.

Table 1. XMS client platforms and compilers

Supported operating system	Supported C and C++ compiler
Microsoft® Windows XP Professional with Service Pack 1	Microsoft Visual C++, Version 6.0 with Service Pack 5
SUSE LINUX Enterprise Server 8 (Intel™ only)	gcc 3.2 (supplied with the operating system)

Function supported

The following function is supported in the first release of XMS:

- An XMS publisher can publish messages using WebSphere MQ Real-Time Transport. To receive the messages and forward them to subscribers, the broker must be configured with a message flow that contains a Real-timeOptimizedFlow message processing node, or a message flow that contains a Real-timeInput message processing node and a Publication message processing node.
- An XMS subscriber can send subscription requests using WebSphere MQ Real-Time Transport. The subscriber can then receive messages, published on the topics to which it has subscribed, using WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport.
- The XMS message model is the same as the WebSphere MQ JMS message model. In particular, XMS implements the same message header fields and message properties that WebSphere MQ JMS implements:
  - JMS header fields. These are fields whose names commence with the prefix JMS.
  - JMS defined properties. These are properties whose names commence with the prefix JMSX.
  - IBM defined properties. These are the properties whose names commence with the prefix JMS_IBM_.

  As a result, XMS subscribers can receive messages published by WebSphere MQ JMS publishers, and WebSphere MQ JMS subscribers can receive messages published by XMS publishers. For each message that is published, some of the header fields and properties are set by the publisher and others are set by the XMS or WebSphere MQ JMS client when the message is sent. Where appropriate, these header fields and properties are propagated with a message through the broker and are made available to each subscriber that receives the

message. This level of interoperability is also available if a WebSphere MQ JMS publisher or subscriber uses WebSphere MQ Enterprise Transport.

Function not supported

The following function is not supported in the first release of XMS:

- An XMS application cannot connect to a WebSphere MQ queue manager and perform messaging and queuing operations.
- Administered objects are not supported. ConnectionFactory and Destination objects are administered objects in JMS but, in XMS, only applications can create these objects and set their attributes and properties.
- Object messages, stream messages, and text messages are not supported. Only bytes messages, map messages, and messages without bodies are supported.
- Durable topic subscribers are not supported. Only nondurable message consumers are supported.
- Temporary topics are not supported.
- Persistent messages are not supported. Only nonpersistent messages are supported.
- Transacted sessions are not supported.
- When an application connects to a broker, the application can supply a user identifier and a password, which the broker can use to authenticate the application. In the WebSphere Business Integration Event Broker and WebSphere Business Integration Message Broker Information Centers, this is referred to as "simple telnet-like password authentication". This is the only form of authentication supported by XMS, and it means that you cannot use the message protection facilities provided by the broker. Authentication using Secure Sockets Layer (SSL) is not supported.
- A TopicRequestor class is not provided, which means that the request/reply style of messaging is not directly supported.

Chapter 2. Installing XMS

This chapter describes how to install the Multi-Language Message Service (XMS) client. For the latest information about installing the product, see the product readme.txt file, which is in the zipped file supplied with XMS.

On all platforms, XMS is installed using an InstallShield MultiPlatform 5 installer. The procedures in this chapter describe how to use the installer in the form of a Wizard with a graphical user interface. However, if you invoke the installer from a command prompt using the runtime command line option **-silent**, you can perform an unattended, or silent, installation, which requires no interaction with the Wizard. Other runtime command line options allow you to have more control over the installation. For general information about MultiPlatform 5, see the InstallShield Web site at http://www.installshield.com/. For more specific information about the runtime command line options, see the *InstallShield MultiPlatform 5 User's Guide*, which you can download from the same Web site.

This chapter contains the following sections:

- "Installing the XMS client"
- "What is installed on Linux" on page 10
- "What is installed on Windows" on page 11
- "Uninstalling the XMS client" on page 12

Installing the XMS client

To install the XMS client on Linux or Windows, follow this procedure. The installed client requires 50 MB of disk space on Linux and 40 MB of disk space on Windows.

1. On Linux, log in as root. On Windows, log on as an administrator.
2. Create a temporary directory and extract the contents of the zipped file supplied with XMS into the directory.

   A subdirectory of the temporary directory is created. The subdirectory is called gxixms_install and contains the files needed for the installation.
3. On Linux, run the file called setuplinuxia32 that is in the gxixms_install directory. On Windows, run the file setup.exe that is in the gxixms_install directory.

   Messages in the command prompt window inform you that the installer is searching for, and preparing, a Java Virtual Machine (JVM). XMS provides a JVM for the installer; you do not need to provide one.

   The Installer window opens and displays the following message:

   ```
   Welcome to the InstallShield Wizard for IBM Multi-Language Message Service
   ```
4. Click **Next**.

   The Installer window asks you to read the licence agreement.
5. If you accept the terms of the licence agreement, click **I accept the terms of the licence agreement**, and then click **Next**.

   The Installer windows asks you where you want to install XMS. If you do not want to install XMS in the directory suggested, you can choose another directory. If you choose to install XMS in a directory that does not currently exist, the installer creates the directory for you.

© Copyright IBM Corp. 2004

6. Click **Next**.

   The Installer window asks you which features you want to install. Ensure that **IBM Multi-Language Message Service Toolkit Feature** is selected.

   The toolkit contains the documentation, the sample applications, and the libraries and header files needed to compile C and C++ applications. If you do not select this feature, only the files needed to run XMS applications are installed.
7. Click **Next**.

   The Installer window displays details of what is about to be installed.
8. Click **Next** to start the installation.

   The Installer windows displays a bar showing the progress of the installation. Wait for the progress bar to complete. When the installation completes successfully, the window displays the following message:

   ```
   The InstallShield Wizard has successfully installed IBM Multi-Language Message
   Service. Choose Finish to exit the wizard.
   ```
9. Click **Finish** to close the Installer window.

You have now successfully installed the XMS client, which is ready to use.

What is installed on Linux

On Linux, XMS is installed in the /opt/IBM/gxixms directory unless you choose to install it in a different directory. Table 2 lists the installed directories, relative to the installation directory, and describes their contents.

Table 2. Installed directories on Linux and their contents

Installed directory	Contents
	The readme.txt file for the product and the licence agreement
_jvm	The Java Virtual Machine (JVM) required by the uninstaller
_uninst	The files required to uninstall the XMS client
doc	This book as a PDF file
lib	The shared object libraries required to compile and run XMS applications, and a symbolic link to the shared object library in the lib/3.2 directory
lib/3.2	The shared object library required to compile XMS applications written in C++ using the gcc 3.2 compiler, and to run the applications
tools/c/include	The XMS header files for C
tools/cpp/include	The XMS header files for C++
tools/samples	The readme.txt file for the sample applications
tools/samples/bin	The compiled sample applications and the command file to run them
tools/samples/c/RTTconsumer	The source and makefile for the C message consumer sample application that uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport

Table 2. Installed directories on Linux and their contents (continued)

Installed directory	Contents
tools/samples/c/RTTconsumersync	The source and makefile for the C message consumer application that uses synchronous message delivery and WebSphere MQ Real-Time Transport
tools/samples/c/RTTproducer	The source and makefile for the C message producer sample application that uses WebSphere MQ Real-Time Transport
tools/samples/cpp/RTTcons	The source and makefile for the C++ message consumer sample application that uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport
tools/samples/cpp/RTTprod	The source and makefile for the C++ message producer sample application that uses WebSphere MQ Real-Time Transport
tools/samples/java/RTTpublisher	The source for the WebSphere MQ JMS message producer sample application that uses WebSphere MQ Real-Time Transport
tools/samples/java/RTTsubscriber	The source for the WebSphere MQ JMS message consumer sample application that uses WebSphere MQ Real-Time Transport
tools/samples/java/RTTsubscribersync	The source for the WebSphere MQ JMS message consumer application that uses synchronous message delivery and WebSphere MQ Real-Time Transport

What is installed on Windows

On Windows, XMS is installed in the C:\Program Files\IBM\gxixms directory unless you choose to install it in a different directory. Table 3 lists the installed directories, relative to the installation directory, and describes their contents.

Table 3. Installed directories on Windows and their contents

Installed directory	Contents
	The readme.txt file for the product and the licence agreement
_jvm	The Java Virtual Machine (JVM) required by the uninstaller
_uninst	The files required to uninstall the XMS client
bin	The *.dll and *.pdb files required to run XMS applications
doc	This book as a PDF file
tools\c\include	The XMS header files for C
tools\cpp\include	The XMS header files for C++
tools\lib	The XMS link libraries for C and C++
tools\samples	The readme.txt file for the sample applications
tools\samples\bin	The compiled sample applications and the command file to run them

Table 3. Installed directories on Windows and their contents (continued)

Installed directory	Contents
tools\samples\c\RTTconsumer	The source and makefile for the C message consumer sample application that uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport
tools\samples\c\RTTconsumersync	The source and makefile for the C message consumer application that uses synchronous message delivery and WebSphere MQ Real-Time Transport
tools\samples\c\RTTproducer	The source and makefile for the C message producer sample application that uses WebSphere MQ Real-Time Transport
tools\samples\cpp\RTTcons	The source and makefile for the C++ message consumer sample application that uses WebSphere MQ Real-Time Transport
tools\samples\cpp\RTTprod	The source and makefile for the C++ message producer sample application that uses WebSphere MQ Real-Time Transport
tools\samples\java\RTTpublisher	The source for the WebSphere MQ JMS message producer sample application that uses WebSphere MQ Real-Time Transport
tools\samples\java\RTTsubscriber	The source for the WebSphere MQ JMS message consumer sample application that uses WebSphere MQ Real-Time Transport
tools\samples\java\RTTsubscribersync	The source for the WebSphere MQ JMS message consumer application that uses synchronous message delivery and WebSphere MQ Real-Time Transport

Uninstalling the XMS client

To remove the XMS client from your Linux or Windows system, follow this procedure:

1. On Linux, log in as root. On Windows, log on as an administrator.
2. On Linux, run the file called uninstaller that is in the directory *install_dir*/_uninst. On Windows, run the file uninstaller.exe that is in the directory *install_dir*_uninst.

   install_dir is the directory where you have installed XMS.

   The Uninstaller window opens and displays the following message:

   ```
   Welcome to the InstallShield Wizard for IBM Multi-Language Message Service
   ```
3. Click **Next**.

   The Uninstaller window displays details of what is about to be uninstalled.
4. Click **Next** to start the removal of the XMS.

   The Uninstaller window confirms that XMS is being uninstalled. When XMS has been removed successfully, the window displays the following message:

   ```
   The InstallShield Wizard has successfully uninstalled IBM Multi-Language Message
   Service. Choose Finish to exit the wizard.
   ```
5. Click **Finish** to close the Uninstaller window.

You have now successfully removed the XMS client from your system.

Uninstalling the XMS client on Windows using Add/Remove Programs

You can remove the XMS client from your Windows system using Add/Remove Programs. Follow this procedure:

1. Log on to Windows as an administrator.
2. From the Windows task bar, click **Start —> Settings —> Control Panel**.

   The Control Panel window opens.
3. Double-click **Add/Remove Programs**.

   The Add/Remove Programs window opens.
4. Click **IBM Multi-Language Message Service** to select it.
5. Click **Change/Remove**.

   The Uninstaller window opens and displays the following message:

   ```
   Welcome to the InstallShield Wizard for IBM Multi-Language Message Service
   ```
6. Click **Next**.

   The Uninstaller window provides information about what is about to be uninstalled.
7. Click **Next** to start the removal of the XMS.

   The Uninstaller window confirms that XMS is being uninstalled. When XMS has been removed successfully, the window displays the following message:

   ```
   The InstallShield Wizard has successfully uninstalled IBM Multi-Language Message
   Service. Choose Finish to exit the wizard.
   ```
8. Click **Finish** to close the Uninstaller window.

You have now successfully removed the XMS client from your system.

Chapter 3. Using XMS

This chapter provides information about how to use XMS after you have installed it. It describes the sample applications provided with XMS, and how to use them to verify your installation. It explains how to build the sample applications and your own applications. The chapter ends by describing how to produce a trace to help diagnose a problem.

The chapter contains the following sections:

- “The sample applications”
- “Running the C sample applications” on page 17
- “Building the sample applications” on page 18
- “Building your own applications” on page 18
- “Problem determination” on page 19

The sample applications

This section describes the sample applications supplied with XMS. Both the source and a compiled version are provided for each application. To find out where the applications are installed on Linux, see Table 2 on page 10. For Windows, see Table 3 on page 11 instead.

Three sets of sample applications are supplied with XMS:

- “The C sample applications”
- “The C++ sample applications” on page 16
- “The WebSphere MQ JMS sample applications” on page 16

The C sample applications

The following C sample applications are supplied with XMS:

RTTproducer
: This application publishes a bytes message every 2 seconds. Each bytes message has a string property and a body that contains a string encoded in UTF-8 format.

RTTconsumer
: This application receives bytes messages asynchronously using WebSphere MQ Real-Time Transport. For each message received, the application reads a string, encoded in UTF-8 format, from the body of the message and gets the value of a string property of the message. The application then displays the two strings on the screen.

  If you start the application with one of the following optional arguments, and the topic to which the application subscribes is configured for multicast in the broker, the application uses WebSphere MQ Multicast Transport instead:

```
multicast:XMSC_MULTICAST_ENABLED
multicast:XMSC_MULTICAST_NOT_RELIABLE
multicast:XMSC_MULTICAST_RELIABLE
```

© Copyright IBM Corp. 2004

RTTconsumersync
This application receives bytes messages synchronously using WebSphere MQ Real-Time Transport. The application calls the xmsMsgConsumerReceiveWithWait() function with a specified wait interval. For each message received, the application reads a string, encoded in UTF-8 format, from the body of the message and gets the value of a string property of the message. The application then displays the two strings on the screen.

The C++ sample applications

The following C++ sample applications are supplied with XMS:

RTTprod
This application publishes a bytes message every 2 seconds. Each bytes message has a string property and a body that contains a string encoded in UTF-8 format.

The application has two classes:

RTTprod
This class contains the main method.

SampleExpListener
This is an application defined exception listener class, which encapsulates the onException() method.

RTTcons
This application receives bytes messages asynchronously using WebSphere MQ Real-Time Transport. For each message received, the application reads a string, encoded in UTF-8 format, from the body of the message and gets the value of a string property of the message. The application then displays the two strings on the screen.

If you start the application with one of the following optional arguments, and the topic to which the application subscribes is configured for multicast in the broker, the application uses WebSphere MQ Multicast Transport instead:

```
multicast:XMSC_MULTICAST_ENABLED
multicast:XMSC_MULTICAST_NOT_RELIABLE
multicast:XMSC_MULTICAST_RELIABLE
```

The application has three classes:

RTTcons
This class contains the main method.

SampleExpListener
This is an application defined exception listener class, which contains the onException() method.

SampleMsgListener
This is an application defined message listener class, which contains the onMessage() method.

The WebSphere MQ JMS sample applications

The following WebSphere MQ JMS sample applications are supplied with XMS. You can use these applications to demonstrate XMS applications exchanging messages with WebSphere MQ JMS applications. For information about how to prepare and run WebSphere MQ JMS applications, see *WebSphere MQ Using Java*.

RTTpublisher
: This application publishes a bytes message every 2 seconds. Each bytes message has a string property and a body that contains a string encoded in UTF-8 format.

RTTsubscriber
: This application receives bytes messages asynchronously using WebSphere MQ Real-Time Transport. For each message received, the application reads a string, encoded in UTF-8 format, from the body of the message and gets the value of a string property of the message. The application then displays the two strings on the screen.

RTTsubscribersync
: This application receives bytes messages synchronously using WebSphere MQ Real-Time Transport. The application calls the receive() method with a specified timeout interval. For each message received, the application reads a string, encoded in UTF-8 format, from the body of the message and gets the value of a string property of the message. The application then displays the two strings on the screen.

Running the C sample applications

This section describes how to run the compiled versions of the RTTproducer and RTTconsumer applications on Linux or Windows. You can use these applications to verify that you have installed XMS correctly.

The RTTproducer and RTTconsumer applications connect to a broker of WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker using WebSphere MQ Real-Time Transport. Before you can run the applications, you must create a message flow that contains a Real-timeOptimizedFlow message processing node, or a message flow that contains a Real-timeInput message processing node and a Publication message processing node, and deploy the message flow to a broker that is running on your system. If you need more information about how to do this, see the WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker Information Center.

After you have prepared the broker, follow this procedure to run the applications:

1. Open a command prompt window.
2. On Linux, change to the directory *install_dir*/tools/samples/bin. On Windows, change to the directory *install_dir*\tools\samples\bin.

   install_dir is the directory where you have installed XMS.
3. At the command prompt, enter the following command:

   ```
   xmsdemo port_number
   ```

   where `port_number` is the port number on which the Real-timeOptimizedFlow message processing node, or the Real-timeInput message processing node, listens for publish and subscribe requests.

The command xmsdemo starts the RTTconsumer application and then the RTTproducer application, passing the following arguments to each application:

```
host:localhost topic:xms/test port:port_number
```

RTTconsumer subscribes to the topic xms/test, and then receives each message published on that topic. RTTproducer publishes a message on the topic xms/test every 2 seconds.

If you want to verify that the RTTproducer and RTTconsumer applications can connect to a broker that is running on a different system, you can edit the command file xmsdemo on Linux, or xmsdemo.cmd on Windows. In the command file, replace both occurrences of the string `localhost` by the IP address or host name of the system on which the broker is running.

Building the sample applications

This section describes how to build the C and C++ sample applications on Linux or Windows.

Before you can build the sample applications on Windows, you must ensure that you have set up the Microsoft Visual C++ build environment. You can do this by running vcvars32.

To build a sample application, follow this procedure:

1. Open a command prompt window.
2. Change to the directory that contains the source and makefile for the sample application you want to build.
   To find out where the source for the sample application is installed on Linux, see Table 2 on page 10. For Windows, see Table 3 on page 11 instead.
3. At the command prompt, enter the following command on Linux:
   ```
   gmake
   ```
   On Windows, enter the following command instead:
   ```
   nmake
   ```
   The command builds an executable version of the application in the current directory.

Building your own applications

This section provides the information you need to build your own XMS applications on Linux or Windows. To find out where the files and libraries mentioned in this section are installed on Linux, see Table 2 on page 10. For Windows, see Table 3 on page 11 instead.

Building your own C applications

Your C applications must include the file xms.h, which defines the XMS function prototypes. The file also includes the file xmsc.h, which defines the data types, enumerated data types, and constants used by the XMS API.

The makefile called Makefile, which is provided for each of the C sample applications, shows you how to build your applications. Note, in particular, that you must link your applications using the shared object library libgxi.so on Linux, or the library gxi.lib on Windows.

Before running your applications on Linux, make sure that the directory *install_dir*/lib is in the path specified by the LD_LIBRARY_PATH environment variable. On Windows, make sure that the directory *install_dir*\bin is in the path specified by the PATH environment variable. *install_dir* is the directory where you have installed XMS.

Building your own C++ applications

Your C++ applications must include the file xms.hpp, which defines the XMS classes and their methods. The file also includes the file xmsc.h, which defines the data types, enumerated data types, and constants used by the XMS API.

The makefile called Makefile, which is provided for each of the C++ sample applications, shows you how to build your applications. Note, in particular, that you must link your applications using the shared object library libgxi01.so on Linux, or the library gxi01vn.lib on Windows.

Before running your applications on Linux, make sure that the directory *install_dir*/lib is in the path specified by the LD_LIBRARY_PATH environment variable. On Windows, make sure that the directory *install_dir*\bin is in the path specified by the PATH environment variable. *install_dir* is the directory where you have installed XMS.

Problem determination

If you experience a problem with XMS, your IBM Support Center might ask you to produce a trace to help diagnose the problem.

To enable tracing for an application, set the environment variable XMS_TRACE_ON to 1 and then start the application.

To disable tracing for an application, clear the environment variable XMS_TRACE_ON. Tracing ends only after the application ends.

XMS creates a trace file in the current working directory unless you specify an alternative location by setting the environment variable XMS_TRACE_FILE_PATH to the fully qualified path name of the directory where you want XMS to create the trace file. You must set the environment variable before you start the application that you want to trace, and you must make sure that the user identifier under which the application runs has the authority to write to the directory where XMS creates the trace file. The name of the trace file starts with the prefix xms, and the file has the extension .trc.

Part 2. Programming with XMS

Chapter 4. Writing XMS applications 23
The threading model 23
Connections 23
Starting and stopping a connection 24
Closing a connection 24
Handling exceptions 24
Sessions 25
Asynchronous message delivery 25
Synchronous message delivery 25
Uniform resource identifiers (URIs) 26
Deleting objects 26
Setting the value of a property 27
Iterators 27
Coded character set identifiers (CCSIDs). 28

Chapter 5. Writing XMS applications in C . . . 31
Object handles in C. 31
Using message and exception listener functions in C 31
Using message listener functions in C 31
Using exception listener functions in C 32
C functions that return a string or byte array by value 33
C functions that return a string or byte array by reference 34
Handling errors in C 34
Return codes 34
The error block 35

Chapter 6. Writing XMS applications in C++ . . 37
Using namespaces in C++ 37
Using the String class in C++ 38
C++ methods that return a byte array 38
Using the PropertyContext class in C++ 39
Handling errors in C++ 39
Using message and exception listeners in C++. . . 41
Using message listeners in C++. 41
Using exception listeners in C++ 43
Assigning XMS objects in C++ 43
Using the C API in a C++ application 45

Chapter 7. XMS messages 47
Header fields in an XMS message 47
Properties of an XMS message 48
JMS defined properties of a message 49
IBM defined properties of a message 49
Application defined properties of a message . . 50
The body of an XMS message 50
Bytes messages 52
Map messages 52
Message selectors 53

© Copyright IBM Corp. 2004

Chapter 4. Writing XMS applications

This chapter provides information that you might find useful when writing Multi-Language Message Service (XMS) applications. If you are writing applications in C, see also Chapter 5, "Writing XMS applications in C," on page 31. If you are writing applications in C++, see also Chapter 6, "Writing XMS applications in C++," on page 37.

The chapter contains the following sections:

- "The threading model"
- "Connections"
- "Sessions" on page 25
- "Uniform resource identifiers (URIs)" on page 26
- "Deleting objects" on page 26
- "Setting the value of a property" on page 27
- "Iterators" on page 27
- "Coded character set identifiers (CCSIDs)" on page 28

The threading model

The following general rules govern how a multithreaded application can use XMS objects:

- Only objects of the following types can be used concurrently on different threads:
  - ConnectionFactory
  - Connection
  - ConnectionMetaData
  - Destination
- A Session object can be used only on the thread on which it is created.
- All other objects can be used only on the same thread as the session in which they are created.

Exceptions to these rules are indicated by entries labelled "Thread context" in the interface definitions of the methods in Chapter 8, "XMS classes," on page 59.

Connections

An application uses a ConnectionFactory object to create a Connection object, and uses a Connection object to create a Session object. Creating a connection is relatively expensive in terms of system resources because it involves establishing a communications connection, and it might also involve authenticating the application.

An XMS application can create multiple connections, and a multithreaded application can use a single Connection object concurrently on multiple threads.

A connection serves several purposes:

- A Connection object encapsulates a communications connection. If a connection uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast

© Copyright IBM Corp. 2004

Transport, the Connection object encapsulates a TCP connection between the application and a broker of WebSphere Business Integration Event Broker or WebSphere Business Integration Message Broker.
- When an application creates a connection, the application can be authenticated.
- A C application can register an exception listener function and context data with a connection. A C++ application can register an exception listener with a connection.

An XMS application typically creates a connection, one or more sessions, and a number of message producers and message consumers.

Starting and stopping a connection

A connection can operate in either started or stopped mode.

When an application creates a connection, the connection is in stopped mode. When the connection is in stopped mode, the application can send messages but not receive messages, either synchronously or asynchronously. The application can use the time while the connection is in stopped mode to initialize sessions.

An application can start a connection by calling the Start Connection method. When the connection is in started mode, the application can send and receive messages. The application can then stop and restart the connection by calling the Stop Connection and Start Connection methods.

Closing a connection

An application closes a connection by calling the Close Connection method. When an application closes a connection, XMS takes the following actions:
- XMS closes all the sessions associated with the connection, and deletes certain objects associated with these sessions. For more information about which objects are deleted, see “Deleting objects” on page 26.
- XMS ends the communications connection with the broker.
- XMS releases the memory and other internal resources used by the connection.

Handling exceptions

If a C application registers an exception listener function and context data with a connection, or if a C++ application registers an exception listener with a connection, XMS notifies the application asynchronously if a serious problem occurs with the connection. XMS notifies a C application by calling the exception listener function, passing a pointer to the context data as one parameter and the handle for the error block as the other parameter. XMS notifies a C++ application by calling the onException() method of the exception listener, passing a pointer to the exception as a parameter.

If an application uses a connection only to consume messages asynchronously, and for no other purpose, then the only way the application can learn about a problem with the connection is by using an exception listener.

For more information about using exception listener functions in a C application, see “Using exception listener functions in C” on page 32. If you are using C++, see “Using exception listeners in C++” on page 43 instead.

Sessions

A session is a single threaded context for sending and receiving messages. Using a Session object, an application can create MessageProducer and MessageConsumer objects, and dynamically create Destination objects that represent topics.

An application can create multiple sessions, where each session produces and consumes messages independently of the other sessions. If two message consumers in separate sessions, or even in the same session, subscribe to the same topic, they each receive a copy of any message published on that topic.

Unlike a Connection object, a Session object cannot be shared across threads. Only the Close Session method of a Session object can be called from a thread other than the one on which the Session object was created. The Close Session method ends a session and releases any system resources allocated to the session.

If an application needs to process messages on more than one thread, the application must first create the additional threads, and then create a session on each thread.

Asynchronous message delivery

If a C application registers a message listener function and context data with a message consumer, or if a C++ application registers a message listener with a message consumer, the application can receive messages asynchronously. When a message arrives for a message consumer, XMS delivers the message to a C application by calling the message listener function, passing a pointer to the context data as one parameter and the handle for the message as the other parameter. XMS delivers the message to a C++ application by calling the onMessage() method of the message listener, passing a pointer to the message as a parameter.

XMS uses one thread to handle all asynchronous message delivery for a session. This means that only one message listener function or one onMessage() method can run at a time. If more than one message consumer in a session is receiving messages asynchronously, and a message listener function or onMessage() method is currently delivering a message to one message consumer, then any other message consumers that are waiting for the same message must continue to wait. Other messages that are waiting to be delivered to the session must also continue to wait.

If an application needs concurrent delivery of messages, it must create more than one session, so that XMS uses more than one thread to handle asynchronous message delivery. In this way, more than one message listener function or onMessage() method can run concurrently.

For more information about using message listener functions in a C application, see “Using message listener functions in C” on page 31. If you are using C++, see “Using message listeners in C++” on page 41 instead.

Synchronous message delivery

Messages are delivered synchronously to an application if the application uses the Receive methods of a MessageConsumer object. Using the Receive methods, an application can wait a specified period of time for a message, or it can wait indefinitely. Alternatively, if an application does not want to wait for a message, it can use the Receive with No Wait method.

Uniform resource identifiers (URIs)

A *uniform resource identifier (URI)* is a string that identifies a destination and, optionally, specifies one or more properties of the destination.

In its simplest form, a URI for a topic has the following format:

```
topic://topic_name
```

where *topic_name* is the name of the topic. If you want to specify one or more properties of a topic, a URI has the following extended format:

```
topic://topic_name?prop_name1=prop_value1&prop_name2=prop_value2& ...
```

where *prop_name* is the name of a property and *prop_value* is the value of a property.

In a URI, you cannot use named constants for the names and values of properties. Table 4 shows, for each property of a topic, the name and valid values that you can use in a URI. For more information about the properties of a topic, see "Properties of Destination" on page 239.

Table 4. The names and valid values of properties that you can use in a topic URI

Name of property	The name that you can use in a URI	The valid values that you can use in a URI (default value in bold)
XMSC_MULTICAST	multicast	**-1 (= XMSC_MULTICAST_ASCF)** 0 (= XMSC_MULTICAST_DISABLED) 3 (= XMSC_MULTICAST_NOT_RELIABLE) 5 (= XMSC_MULTICAST_RELIABLE) 7 (= XMSC_MULTICAST_ENABLED)
XMSC_PRIORITY	priority	**-2 (= XMSC_PROPERTY_AS_APP)** An integer in the range 0 to 9

An application can use a topic URI as a parameter when it calls the Create Destination method of the Destination class. Here is an example in a fragment of C code:

```
rc = xmsDestCreate("topic://Sport/Football/Results?multicast=0&priority=9",
                   &topic,
                   xmsError);
```

A C++ application can also use a topic URI as a parameter when it calls the Create Topic method of the Session class. Here is an example in a fragment of C++ code:

```
topic = session.createTopic("topic://Sport/Football/Results?multicast=7");
```

Deleting objects

When an application creates an XMS object, XMS allocates memory and other internal resources to the object. XMS retains these internal resources until the application explicitly deletes the object by calling the object's close or delete method, at which point XMS releases the internal resources. In a C++ application, an object is also deleted when it goes out of scope. If an application tries to delete an object that is already deleted, the call is ignored.

When an application deletes a Connection or Session object, XMS deletes certain associated objects automatically and releases their internal resources. These are objects that were created by the Connection or Session object and depend for their

existence upon the connection or session. These objects are shown in Table 5. Note that, if an application closes a connection with dependent sessions, all objects dependent on those sessions are also deleted. Only a Connection or Session object can have dependent objects.

Table 5. Objects that are deleted automatically

Deleted object	Method	Dependent objects that are deleted automatically
Connection	Close Connection	ConnectionMetaData and Session objects
Session	Close Session	MessageConsumer and MessageProducer objects

Setting the value of a property

When an application sets the value of a property of an XMS object, the value replaces any previous value the property had. Similarly, when an application sets the property value attribute of a Property object, the property value replaces any previous value the attribute had.

Iterators

An iterator encapsulates a list of Property objects and a cursor that maintains the current position in the list. When an iterator is created, the position of the cursor is before the first Property object.

An application uses an iterator to retrieve each Property object in turn. To retrieve the Property objects, the application uses the following three methods of the Iterator class:

- Check for More Properties
- Get Next Property
- Reset Iterator

The Iterator class is equivalent to the Enumerator class in Java.

An application can use an iterator to retrieve the properties of a message, or to retrieve the name-value pairs in the body of a map message. The following code fragment shows how a C application can use an iterator to print out all properties of a message:

```
/********************************************************/
/* XMS Sample using an iterator to browse properties    */
/********************************************************/
rc = xmsMsgGetProperties(hMsg, &it, xmsError);
if (rc == XMS_OK)
{
  rc = xmsIteratorHasNext(it, &more, xmsError);
  while (more)
  {
    rc = xmsIteratorGetNextProperty(it, &p, xmsError);
    if (rc == XMS_OK)
    {
      xmsPropertyGetName(p, name, 100, &len, xmsError);
      printf("Property name=\"%s\"\n", name);
      xmsPropertyGetType(p, &type, xmsError);
      switch (type)
      {
        case XMS_PROPERTY_TYPE_INT:
        {
          xmsINT value=0;
          xmsPropertyGetInt(p, &value, xmsError);
```

```
        printf("Property value=%d\n", value);
        break;
      }
      case XMS_PROPERTY_TYPE_STRING:
      {
        xmsSIZE len=0;
        char value[100];
        xmsPropertyGetString(p, value, 100, &len, xmsError);
        printf("Property value=\"%s\"\n", value);
        break;
      }
      default:
      {
        printf("Unhandled property type (%d)\n", (int)type);
      }
    }
    xmsPropertyDispose(&p, xmsError);
  }
  rc = xmsIteratorHasNext(it, &more, xmsError);
 }
 printf("Finished iterator....\n");
 xmsIteratorDispose(&it, xmsError);
}
/*********************************************************/
```

Coded character set identifiers (CCSIDs)

When a message is transported over WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport, the character data in the message is in Unicode. If required therefore, XMS converts the character data in an incoming message from Unicode into the code page used by the receiving application, and converts the character data in an outgoing message from the code page used by the sending application into Unicode.

The character data in transmitted messages is in Unicode so that XMS applications can exchange messages with WebSphere MQ JMS applications. Character data that requires conversion might be in any of the following parts of a message:

- Header fields (see “Header fields in an XMS message” on page 47)
- Properties (see “Properties of an XMS message” on page 48)
- The body (see “The body of an XMS message” on page 50)

The XMSC_CLIENT_CCSID property of a ConnectionFactory, Connection, or Session object specifies which code page an application is using. The value of the XMSC_CLIENT_CCSID property is a coded character set identifier (CCSID), which identifies a code page. XMS sets the initial value of this property when the application creates one of these objects, but the application can change the value subsequently.

When an application creates a connection factory, XMS derives an appropriate CCSID for the application from information provided by the operating system. On Windows, the derived CCSID depends on whether the application is a console application or a Windows application with a graphical user interface. On Linux, XMS derives the CCSID from the code set name that XMS extracts from the language information defined in the locale of the application. The derived CCSID becomes the initial value of the XMSC_CLIENT_CCSID property of the ConnectionFactory object.

When an application uses a ConnectionFactory object to create a connection, XMS copies the XMSC_CLIENT_CCSID property of the ConnectionFactory object to the

newly created Connection object. XMS copies the property only at the time the application creates the connection. If the application subsequently changes the value of the property of the ConnectionFactory object, XMS does not propagate the change to the Connection object.

In a similar way, when an application uses a Connection object to create a session, XMS copies the XMSC_CLIENT_CCSID property of the Connection object to the newly created Session object. XMS copies the property only at the time the application creates the session.

An application can change the value of the XMSC_CLIENT_CCSID property of a ConnectionFactory, Connection, or Session object by calling the Set Property method. Before the application can call Set Property, however, it must first create a Property object and set an integer property value in the object by calling the Set Integer Property Value method. The application can specify one of the following values for the propertyValue parameter of the Set Integer Property Value method:

XMSC_USE_PROCESS_CCSID
When the application calls Set Property, XMS derives an appropriate CCSID for the application from information provided by the operating system, as described previously. The new value of the XMSC_CLIENT_CCSID property is the derived CCSID.

XMSC_DEFAULT_RTT_CCSID
When the application calls Set Property, the new value of the XMSC_CLIENT_CCSID property is the CCSID that identifies the code page used by the WebSphere MQ Real-Time Transport and WebSphere MQ Multicast Transport services of the broker to which the application is connected.

A coded character set identifier (CCSID)
When the application calls Set Property, the new value of the XMSC_CLIENT_CCSID property is the specified CCSID. If the application specifies a CCSID that is not valid, or specifies a CCSID for which the platform does not support code page conversion, the call fails and XMS returns error code XMS_E_ILLEGAL_PROPERTY_VALUE.

Alternatively, for a ConnectionFactory object only, an application can change the value of the XMSC_CLIENT_CCSID property by calling the Set Integer Property method. The propertyValue parameter of the Set Integer Property method can also have any of the values just described.

When XMS needs to convert the character data in a message, XMS uses the code page associated with the session in which the message is being sent or received. XMS determines the code page from the value of the XMSC_CLIENT_CCSID property of the Session object. If the session is using the same code page as the WebSphere MQ Real-Time Transport and WebSphere MQ Multicast Transport services of the broker to which the session it is connected, XMS does not attempt to perform any code page conversion for messages sent to, or received from, the broker.

Chapter 5. Writing XMS applications in C

This chapter provides information that you might find useful when writing Multi-Language Message Service (XMS) applications in C. The chapter contains the following sections:

- "Object handles in C"
- "Using message and exception listener functions in C"
- "C functions that return a string or byte array by value" on page 33
- "C functions that return a string or byte array by reference" on page 34
- "Handling errors in C" on page 34

Object handles in C

When writing an XMS application in C, every object handle has a data type, where the data type of the handle relates specifically to the type of object. Table 6 shows the handle data type for each type of object.

Note, however, that BytesMessage, MapMessage, and Message objects all have handles with data type xmsHMsg. For more information about how to use handles for messages, see "The body of an XMS message" on page 50.

Table 6. Object handle data types

Type of object	Object handle data type
BytesMessage	xmsHMsg
Connection	xmsHConn
ConnectionFactory	xmsHConnFact
ConnectionMetaData	xmsHConnMetaData
Destination	xmsHDest
ErrorBlock	xmsHErrorBlock
Iterator	xmsHIterator
MapMessage	xmsHMsg
Message	xmsHMsg
MessageConsumer	xmsHMsgConsumer
MessageProducer	xmsHMsgProducer
Property	xmsHProperty
Session	xmsHSess

Using message and exception listener functions in C

A C application uses a message listener function to receive messages asynchronously, and an exception listener function to be notified asynchronously of a problem with a connection.

Using message listener functions in C

To receive messages asynchronously, a C application must register a message listener function and context data with one or more message consumers. The

© Copyright IBM Corp. 2004

application does this by calling the xmsMsgConsumerSetMessageListener() function for each message consumer, passing pointers to the message listener function and context data as parameters.

A message listener function is a callback function written by the user. When a message arrives for a message consumer, XMS calls the message listener function to deliver the message, passing a pointer to the context data as one parameter and the handle for the message as the other parameter.

The format and content of the context data is defined by the application, and the data itself occupies memory owned by the application. For example, the context data might be a structure allocated on the heap. The context data contains all the information that the message listener function needs to refer to when processing a message. XMS does not make a copy of the context data, and so the application must ensure that the context data is still available when XMS calls the message listener function.

For more information about the message listener function, including its signature, see “MessageListener” on page 178.

Note that it is the responsibility of the application to release the resources used by a message that is received asynchronously. XMS does not release these resources.

To stop the asynchronous delivery of messages to a message consumer, the application can call the xmsMsgConsumerSetMessageListener() function again, this time passing a null pointer as a parameter instead of a pointer to a message listener function.

Using exception listener functions in C

Using an exception listener function is similar in principle to using a message listener function.

A C application must register an exception listener function with a connection by calling the xmsConnSetExceptionListener() function, passing pointers to the exception listener function and context data as parameters. An exception listener function is a callback function written by the user. If XMS detects a problem with the connection, XMS calls the exception listener function, passing a pointer to the context data as one parameter and the handle for an error block as the other parameter.

The context data contains all the information that the exception listener function needs to refer to when processing an error block. In all other respects, the way that context data is used with an exception listener function is the same as the way it is used with a message listener function.

For more information about the exception listener function, including its signature, see “ExceptionListener” on page 113.

Note that it is the responsibility of the application to release the resources used by an error block received in this way. XMS does not release these resources.

To stop the asynchronous reporting of problems with a connection, the application can call the xmsConnSetExceptionListener() function again, this time passing a null pointer as a parameter instead of a pointer to an exception listener function.

C functions that return a string or byte array by value

In the C API, certain functions return a string or byte array as a parameter. Each of these functions provides essentially the same interface for retrieving a string or a byte array.

Here is an example of one of these functions. The function implements the Get String Property method in the Message class and returns a string.

```
xmsRC xmsMsgGetStringProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsCHAR *propertyValue,
                              xmsSIZE length,
                              xmsSIZE *actualLength,
                              xmsHErrorBlock errorBlock);
```

Three parameters control the retrieval of the string:

propertyValue
: This parameter is a pointer to a buffer provided by the application into which XMS copies the characters in the string. If data conversion is required, XMS converts the characters into the code page used by the application before copying them into the buffer.

length
: This parameter is the length of the buffer in bytes. This is an input parameter that must be set by the application before the call.

actualLength
: This output parameter is the length of the string that XMS stores in the buffer. The length is measured in bytes. If data conversion is required, this is the length after conversion.

If the buffer is not large enough to store the whole string, XMS returns the string truncated to the length of the buffer, sets the actualLength parameter to the actual length of the string, and returns error code XMS_E_DATA_TRUNCATED.

If the length parameter is zero, XMS returns the length of the string in the actualLength parameter but does not copy the string into the buffer. XMS returns error code XMS_E_DATA_TRUNCATED in this case as well.

If the length of the buffer is larger than the length of the string that XMS copies into the buffer, XMS appends a null character to the end of the string. If the length of the buffer is less than or equal to the length of the string, XMS does not append a null character. The length of the string in the buffer, as reported by XMS in the actualLength parameter, does not include this null character. If a function returns a byte array instead of a string, XMS still appends a null character to the end of the byte array, if there is room in the buffer.

Note that, if an XMS application receives a message sent by a WebSphere MQ JMS application, strings in the header fields, properties, and body of the message might contain embedded null characters. You cannot use the standard C string manipulation functions to process strings containing embedded nulls because these functions assume that the first null character in a string marks the end of the string.

C functions that return a string or byte array by reference

When a C application calls one of the functions described in "C functions that return a string or byte array by value" on page 33, XMS must copy the string or byte array into the buffer provided by the application. If an application is processing a large volume of messages, and the strings or byte arrays in the messages are very large, then the time taken to copy them might have an impact on performance.

To deliver better performance in this situation, the C API provides another set of functions. When an application calls one of these functions, one parameter returns a pointer to a string or byte array that is stored in memory owned by XMS, and another parameter returns the length of the string or byte array.

Here are some examples of these functions:

- xmsMsgGetStringPropertyByRef(), which implements the Get String Property by Reference method in the Message class
- xmsBytesMsgReadBytesByRef(), which implements the Read Bytes by Reference method in the BytesMessage class

If data conversion is required for a string, XMS converts the characters into the code page of the application and returns a pointer to the converted string. The length returned to the application is the length of the converted string.

If data conversion is required, the first time an application retrieves a string by reference might take as long as retrieving the string by value. However, XMS caches the converted string and so subsequent calls to retrieve the same string do not take as long.

Because these functions return a pointer to memory owned by XMS, the application must not attempt to free or modify the contents of this memory. Attempting to do so might cause unpredictable results.

The pointer returned to the application remains valid until the XMS object, with which the string or byte array is associated, is deleted. The application must copy the string or byte array if it needs the data after the object is deleted.

Handling errors in C

Most functions in the C API return a value that is a return code, and have an optional input parameter that is a handle for an error block. This section describes the respective roles of the return code and the error block.

Return codes

The return code from a C function call indicates whether the call was successful. The return code has data type xmsRC. Table 7 shows the possible return codes and their meaning.

Table 7. Return codes from C function calls

Return code	Meaning
XMS_OK	The call completed successfully.
Any other value	The call failed. The error block contains more details about why the call failed.

The error block

When an application calls a C function, the application can include a handle for an error block as an input parameter on the call. If the call fails, XMS stores information in the error block about why the call failed. The application can then retrieve this information from the error block.

An error block contains the following information:

Exception code
An integer representing the exception. The exception code provides a high level indication of why the call failed, but does not indicate precisely which error has occurred. The header file xmsc.h defines a named constant for each exception code.

The exception code matches the JMS exception that is thrown by a JMS method in the same circumstances.

Error code
An integer representing the error. The error code provides a more precise indication of which error has occurred. The header file xmsc.h defines a named constant for each error code.

Error string
A null terminated string of characters that describes the error. The characters in the string are the same as those in the named constant that represents the error code.

Error data
A null terminated string of characters that provides additional information about the error. The information is free format.

Linked error
The handle for an linked error block. If XMS needs to report more information about a call that has failed, XMS can create one or more additional error blocks and chain them from the error block provided by the application.

XMS provides a set of helper functions to create an error block and extract information from it. An application must use a helper function to create an error block and obtain a handle for it before calling the first function that can accept the handle as an input parameter. If the function call fails, the application can then use other helper functions to extract information about the error that XMS has stored in the error block. For details of these helper functions, see “ErrorBlock” on page 105.

Chapter 6. Writing XMS applications in C++

This chapter provides information that you might find useful when writing Multi-Language Message Service (XMS) applications in C++. The chapter contains the following sections:

- "Using namespaces in C++"
- "Using the String class in C++" on page 38
- "C++ methods that return a byte array" on page 38
- "Using the PropertyContext class in C++" on page 39
- "Handling errors in C++" on page 39
- "Using message and exception listeners in C++" on page 41
- "Assigning XMS objects in C++" on page 43
- "Using the C API in a C++ application" on page 45

Using namespaces in C++

All the C++ classes supplied with XMS are declared in a namespace called xms. A C++ application can therefore adopt one of the following approaches when referring to the names of XMS classes:

- The application can qualify the names of XMS classes with the name of the namespace, xms. See the following fragment of code, for example:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  xms::ConnectionFactory connFact;
  xms::Connection        conn;

  conn = connFact.createConnection();

  // Other code here

  cout << "Exiting..." << endl;

  return(0);
}
```

- The application can use a using directive to make the names of XMS classes available without having to qualify them. See the following fragment of code, for example:

```
#include <xms.hpp>

using namespace std;
using namespace xms;

int main(int argc, char *argv[])
{
  ConnectionFactory connFact;
  Connection        conn;

  conn = connFact.createConnection();

  // Other code here
```

© Copyright IBM Corp. 2004

```
    cout << "Exiting..." << endl;

    return(0);
  }
```

Using the String class in C++

In the C++ API, a String object encapsulates a string. When called, certain methods accept a String object as a parameter or return a String object.

A String object can encapsulate a null terminated character array. Alternatively, a String object can encapsulate a byte array with embedded null characters, where the byte array might, or might not be, null terminated. Therefore, when an application creates a String object from a byte array, the application must specify the length of the byte array. The following code fragment creates both types of String object:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  xms::String strA("Normal character string");
  xms::String strB("This\0string\0contains\0nulls", 26);

  // The overloaded assignment operator can be used to create
  // a String object from a null terminated character array.

  xms::String strC = "Another character string";

  // Other code here

  return(0);
}
```

To make it easier to create and manipulate String objects, certain operators and constructors are overloaded on the String class. If an application calls a method that requires a String object as an input parameter, it is not necessary to create the String object first. The application can simply pass a null terminated character array to the method as a parameter, and XMS automatically creates a String object on the stack.

In addition, the String class encapsulates methods to create and manipulate String objects. For the definitions of these methods, see “String” on page 232.

C++ methods that return a byte array

In the C++ API, certain methods return a byte array as a parameter. Each of these methods provides essentially the same interface for retrieving a byte array.

Here is an example of one of these methods, Get Object Property in the Message class:

```
xmsOBJECT_TYPE getObjectProperty(const String & propertyName,
                                 xmsBYTE *propertyValue,
                                 const xmsSIZE length,
                                 xmsSIZE *actualLength) const;
```

The way that the parameters propertyValue, length, and actualLength control the retrieval of the byte array is exactly the same as the way described in “C functions that return a string or byte array by value” on page 33.

Other examples of these methods are getBytes() and getObject() in the MapMessage class, getByteArray() in the Property class, getBytesProperty() in the PropertyContext class, and get() in the String class.

Using the PropertyContext class in C++

The PropertyContext class is an abstract superclass that encapsulates methods that get and set properties. These methods are inherited, directly or indirectly, by the following XMS classes:

- BytesMessage
- Connection
- ConnectionFactory
- ConnectionMetaData
- Destination
- MapMessage
- Message
- MessageConsumer
- MessageProducer
- Session

For the definitions of the methods in the PropertyContext class, see "PropertyContext" on page 212.

Handling errors in C++

If XMS detects an error while processing a call to a method, XMS throws an exception. An XMS exception is an object of one of the following types:

- Exception
- IllegalStateException
- InvalidDestinationException
- InvalidSelectorException
- MessageEOFException
- MessageFormatException
- MessageNotReadableException
- MessageNotWritableException
- ResourceAllocationException
- SecurityException

The Exception class is a superclass of each of the remaining classes in this list. As a result, an application can include the calls to XMS methods in a try block and, to catch all types of XMS exception, the application can simply specify the Exception class in the exception declaration of the catch construct. The following code fragment illustrates this technique:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  int nRC = 0;

  try
  {
```

```
    xms::ConnectionFactory connFact;
    xms::Connection        conn;

    connFact.setStringProperty(XMSC_HOST_NAME, "localhost");
    connFact.setIntProperty(XMSC_PORT, 1506);

    conn = connFact.createConnection();

    // Other code here
  }
  catch(xms::Exception & ex)
  {
    cerr << ex;

    // Error handling code here

    nRC = -1;
  }

  return(0);
}
```

Note that, if an application uses this technique to catch XMS exceptions, the application must catch an exception by reference, and not by value. This ensures that an exception is not sliced and valuable data about the error is not lost.

The Exception class itself is a subclass of the std::exception class. Therefore, to catch all exceptions, including those thrown by the C++ run-time environment, an application can simply specify the std::exception class in the exception declaration of the catch construct. The following code fragment illustrates this point:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  int nRC = 0;

  try
  {
    xms::ConnectionFactory connFact;

    connFact.setStringProperty(XMSC_HOST_NAME, "localhost");
    connFact.setIntProperty(XMSC_PORT, 1506);

    // Additional code here
  }
  catch(exception & ex)
  {
    // Error handling code here

    nRC = -1;
  }

  return(0);
}
```

After an application catches an XMS exception, the application can use the methods of the Exception class to find out information about the error. For the definitions of these methods, see “Exception” on page 109. The information encapsulated by an XMS exception is essentially the same as the information provided to a C application in an error block. For details of this information, see “The error block” on page 35.

If XMS detects more than one error during a call, XMS can create an exception for each error and link the exceptions to form a chain. After an application has caught the first exception, the application can call the getLinkedException() method to get a pointer to the next exception in the chain. The application can continue to call the getLinkedException() method on each exception in the chain until a null pointer is returned, indicating that there are no more exceptions in the chain.

Note that, because the getLinkedException() method returns a pointer to a linked exception, it is the responsibility of the application to release the object using the C++ delete operator.

The Exception class provides the dump() method, which an application can use to dump an exception, as formatted text, to a specified C++ output stream. The operator << is overloaded on the Exception class and can be used for the same purpose.

Using message and exception listeners in C++

A C++ application uses a message listener to receive messages asynchronously, and uses an exception listener to be notified asynchronously of a problem with a connection.

Using message listeners in C++

To receive messages asynchronously, a C++ application must define a message listener class that is based on the abstract class MessageListener. The message listener class must provide an implementation of the onMessage() method. The application can then instantiate the class to create a message listener, and register the message listener with one or more message consumers by calling the setMessageListener() method for each message consumer. Subsequently, when a message arrives for a message consumer, XMS calls the onMessage() method to deliver the message. XMS does not make a copy the message listener, and so the application must ensure that the message listener is still available when XMS calls the onMessage() method.

Note that, if more than one message consumer in a session has a registered message listener, only one onMessage() method can run at a time. For more information about this situation, and what to do if your application needs concurrent delivery of messages, see “Asynchronous message delivery” on page 25.

To stop the asynchronous delivery of messages to a message consumer, the application can call the setMessageListener() method again, this time passing a null pointer as the parameter instead of a pointer to a message listener. Unless the registration of a message listener is cancelled in this way, the message listener must exist for as long as the message consumer exists.

The following code fragment provides an example of how to implement a message listener class with an onMessage() method:

```
#include <xms.hpp>

using namespace std;

class MyMsgListener : public xms::MessageListener
{
public:

  virtual xmsVOID onMessage(const Message * pMsg);
```

```
};

  -----------------------------------------------------

xmsVOID MyMsgListener::onMessage(const Message * pMsg)
{
  if (pMsg != NULL)
  {
    cout << pMsg->getJMSCorrelationID() << endl;
    cout << pMsg->getJMSMessageID() << endl;

    if (pMsg->getType() == XMSC_T_BYTES_MSG)
    {
      xms::BytesMessage * pBytes = (xms::BytesMessage *) pMsg;

      cout << pBytes->readUTF() << endl;
    }

    delete pMsg;
  }
}
```

Note that, because XMS delivers a pointer to a message when it calls the onMessage() method, it is the responsibility of the application to release the message using the C++ delete operator.

The following code fragment now shows how an application can use this message listener class to implement the asynchronous delivery of messages to a message consumer:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  int                    nRC = 0;
  xms::ConnectionFactory cf;
  xms::Connection        conn;
  xms::Session           sess;
  xms::Destination       dest;
  xms::MessageConsumer   msgConn;
  MyMsgListener          msgLst;

  try
  {
    cf.setIntProperty(XMSC_TRANSPORT_TYPE, XMSC_TP_DIRECT_TCPIP);
    cf.setStringProperty(XMSC_HOST_NAME, "localhost");
    cf.setIntProperty(XMSC_PORT, 1506);
    cf.setIntProperty(XMSC_MULTICAST, XMSC_MULTICAST_DISABLED);

    conn    = cf.createConnection();
    sess    = conn.createSession();
    dest    = xms::Destination(XMSC_TOPIC, "test");
    msgConn = sess.createConsumer(dest);

    msgConn.setMessageListener(&msgLst);

    conn.start();

    while(xmsTRUE)
    {
      Sleep(1000);
      cout << "Waiting..." << endl;
    }
  }
```

```
  catch(exception & ex)
  {
    nRC = -1;
  }

  return(nRC);
}
```

Using exception listeners in C++

Using an exception listener is similar in principle to using a message listener.

A C++ application must define an exception listener class that is based on the abstract class ExceptionListener. The exception listener class must provide an implementation of the onException() method. The application can then instantiate the class to create an exception listener, and register the exception listener with a connection by calling the setExceptionListener() method. Subsequently, if XMS detects a problem with the connection, XMS calls the onException() method to pass an exception to the application. XMS does not make a copy the exception listener, and so the application must ensure that the exception listener is still available when XMS calls the onException() method.

To stop the asynchronous reporting of problems with a connection, the application can call the setExceptionListener() method again, this time passing a null pointer as the parameter instead of a pointer to an exception listener. Unless the registration of an exception listener is cancelled in this way, the exception listener must exist for as long as the connection exists.

Note that, because XMS passes a pointer to an exception when it calls the onException() method, it is the responsibility of the application to release the exception by using the C++ delete operator.

Assigning XMS objects in C++

The assignment operator is overloaded on each of the XMS classes listed in Table 8. If an object is already assigned to one variable, and an application assigns the value of that variable to another variable of the same type, the precise action of the overloaded assignment operator depends on the type of the object being assigned:

- For some types of object, a copy of the object is assigned to the second variable. This is called a *deep copy*. When a deep copy is made, the original object and its copy become two completely separate objects, which can be used independently of each other.
- For the other types of object, only a reference to the object is copied and assigned to the second variable. This is called a *shallow copy*. When a shallow copy is made, the two variables reference the same object. If an application makes changes to the object by accessing the object through one variable, the application can see those changes if it accesses the object through the other variable.

Table 8 indicates, for each type of object, whether the overloaded assignment operator makes a shallow or a deep copy of an object.

Table 8. The XMS classes on which the assignment operator is overloaded

Class	Shallow copy	Deep copy
BytesMessage	✓	
Connection	✓	

Table 8. The XMS classes on which the assignment operator is overloaded (continued)

Class	Shallow copy	Deep copy
ConnectionFactory	✓	
ConnectionMetaData	✓	
Destination	✓	
Exception		✓
IllegalStateException		✓
InvalidDestinationException		✓
InvalidSelectorException		✓
Iterator	✓	
MapMessage	✓	
Message	✓	
MessageConsumer	✓	
MessageEOFException		✓
MessageFormatException		✓
MessageNotReadableException		✓
MessageNotWritableException		✓
MessageProducer	✓	
Property		✓
ResourceAllocationException		✓
SecurityException		✓
Session	✓	
String		✓

When a shallow copy of an object is made, the object is deleted only when all the variables that reference the object go out of scope. If the application closes or deletes the object before the variables that reference the object go out of scope, the application can no longer access the object through any of the variables.

The following code fragment illustrates these points:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  xms::ConnectionFactory cf;
  xms::Connection        conn;
  xms::Session           sess;
  xms::Session           sess2;

  conn = cf.createConnection();
  sess = conn.createSession();

  // Make a shallow copy of the Session object.

  sess2 = sess;

  // Set a property in the Session object using the sess2 variable.

  sess2.setStringProperty("property", "test");
```

```
  // Make another shallow copy of the Session object.

  if (sess2.isNull() != xmsTRUE)
  {
    xms::Session sess3 = sess2;

    // Set another property in the Session object, this time using
    // the sess3 variable.

    sess3.setStringProperty("another property", "test");
  }

  // The sess3 variable is now out of scope, but the second property
  // is still set in the Session object.

  // Close the Session object.

  sess.close();

  // The Session object is now closed and can no longer be accessed
  // through the sess2 variable.  As a result, the following statement
  // causes "invalid session" to be written to the standard output
  // stream.

  if (sess2.isNull() == xmsTRUE)
  {
    cout << "invalid session" << endl;
  }

  return(0);
}
```

Using the C API in a C++ application

Most C++ classes supplied with XMS provide a getHandle() method. A C++ application can call the getHandle() method of an object to retrieve the handle that a C application would use to access the object. The C++ application can then use the handle to access the object by calling functions in the C API. The following code fragment illustrates how this is done:

```
#include <xms.hpp>

using namespace std;

int main(int argc, char *argv[])
{
  xms::ConnectionFactory cf;
  xms::Connection        conn;
  xmsHConn               hConn;

  conn = cf.createConnection();

  // Retrieve the handle for the connection.

  hConn = conn.getHandle();

  // Using the retrieved handle, call a C API function.

  xmsConnStart(hConn, NULL);

  // Other code here

  return(0);
}
```

Using the handle for an object, a C++ application can close or delete the object by calling the appropriate C API function. However, if a C++ application closes or deletes an object in this way, the application can no longer use the object using the C++ API.

Being able to use the C API is useful if a C++ application needs to use function that is available only in the C API. The function described in "C functions that return a string or byte array by reference" on page 34 is an example of such function.

Chapter 7. XMS messages

This chapter describes the structure and content of Multi-Language Message Service (XMS) messages and how applications process XMS messages.

An XMS message has the following parts:

A header
The header of a message contains fields, and all messages contain the same set of header fields. XMS and applications use the values of the header fields to identify and route messages. For more information about header fields, see "Header fields in an XMS message."

A set of properties
The properties of a message specify additional information about the message. Although all messages have the same set of header fields, every message can have a different set of properties. For more information about the properties of a message, see "Properties of an XMS message" on page 48.

A body
The body of a message contains application data. For more information about the body of a message, see "The body of an XMS message" on page 50.

An application can select which messages it wants to receive. It does this by using message selectors, which specify the selection criteria. The criteria can be based on the values of certain header fields and the values of any of the properties of a message. For more information about message selectors, see "Message selectors" on page 53.

Header fields in an XMS message

To allow an XMS application to exchange messages with a WebSphere MQ JMS application, the header of an XMS message contains the JMS message header fields. The names of these header fields commence with the prefix JMS. For a description of the JMS message header fields, see the *Java Message Service Specification, Version 1.1*.

XMS implements the JMS message header fields as attributes of a Message object. Each header field has its own methods for setting and getting its value. For a description of these methods, see "Message" on page 139. A header field is always readable and writable.

Table 9 lists the JMS message header fields and indicates how the value of each field is set for a transmitted message. Note that some of the fields are set automatically by XMS when an application sends a message or, in the case of JMSRedelivered, when an application receives a message.

Table 9. JMS message header fields

Name of the JMS message header field	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMSCorrelationID	Set JMSCorrelationID [Message]

© Copyright IBM Corp. 2004

Table 9. JMS message header fields (continued)

Name of the JMS message header field	How the value is set for a transmitted message (in the format *method* [*class*])
JMSDeliveryMode	Send [MessageProducer]
JMSDestination	Send [MessageProducer]
JMSExpiration	Send [MessageProducer]
JMSMessageID	Send [MessageProducer]
JMSPriority	Send [MessageProducer]
JMSRedelivered	Receive [MessageConsumer]
JMSReplyTo	Set JMSReplyTo [Message]
JMSTimestamp	Send [MessageProducer]
JMSType	Set JMSType [Message]

Properties of an XMS message

To allow an XMS application to exchange messages with a WebSphere MQ JMS application, XMS supports the following predefined properties of a Message object:

- The same JMS defined properties that WebSphere MQ JMS supports. The names of these properties commence with the prefix JMSX.
- The same IBM defined properties that WebSphere MQ JMS supports. The names of these properties commence with the prefix JMS_IBM_.

Each predefined property has two names:

- A JMS name, for a JMS defined property, or a WebSphere MQ JMS name, for an IBM defined property.

  This is the name by which the property is known in JMS or WebSphere MQ JMS, and is also the name that is transmitted with a message that has this property. An XMS application uses this name to identify the property in a message selector expression.
- An XMS name.

  An XMS application uses this name to identify a property in all situations except in a message selector expression. Each XMS name is defined as a named constant in the header file, xmsc.h. The value of the named constant is the corresponding JMS or WebSphere MQ JMS name.

In addition to the predefined properties, an XMS application can create and use its own set of message properties. These properties are called *application defined properties*.

A message property does not have its own methods for setting and getting its value. Instead, the Message class provides the following generic methods:

- A set and a get method for each of the following data types: xmsBOOL, xmsBYTE, xmsSHORT, xmsINT, xmsLONG, xmsFLOAT, xmsDOUBLE, and String (or character array, if you are using the C interface)
- A Set Property method and a Get Property method

For more information about these methods, see "Message" on page 139.

After an application creates a message, the properties of the message are readable and writable. The properties remain readable and writable after the application

sends the message. When an application receives a message, the properties of the message are read-only. If an application calls the Clear Properties method of the Message class when the properties of a message are read-only, the properties become readable and writable. The method also clears the properties.

To determine the values of all the properties of a message, an application can call the Get Properties method of the Message class. The method creates an iterator that encapsulates a list of Property objects, where each Property object represents a property of the message. The application can then use the methods of the Iterator class to retrieve each Property object in turn, and use the methods of the Property class to retrieve the name, data type, and value of each property. For a sample fragment of C code that performs a similar function, see "Iterators" on page 27.

JMS defined properties of a message

Table 10 lists the JMS defined properties of a message that are supported by both XMS and WebSphere MQ JMS. For a description of the JMS defined properties, see the *Java Message Service Specification, Version 1.1*.

The table specifies the data type of each property and indicates how the value of the property is set for a transmitted message. Note that some of the properties are set automatically by XMS when an application sends a message or, in the case of JMSXDeliveryCount, when an application receives a message.

Table 10. JMS defined properties of a message

XMS name of the JMS defined property	JMS name	Data type	How the value is set for a transmitted message (in the format *method* [*class*])
JMSX_APPID	JMSXAppID	String[1]	Send [MessageProducer]
JMSX_DELIVERY_COUNT	JMSXDeliveryCount	xmsINT	Receive [MessageConsumer]
JMSX_GROUPID	JMSXGroupID	String[1]	Set String Property [Message]
JMSX_GROUPSEQ	JMSXGroupSeq	xmsINT	Set Integer Property [Message]
JMSX_USERID	JMSXUserID	String[1]	Send [MessageProducer]

Notes:

1. This is the data type if you are using C++. If you are programming in C, it is a character array.

IBM defined properties of a message

Table 11 lists the IBM defined properties of a message that are supported by both XMS and WebSphere MQ JMS. For more information about the IBM defined properties, see *WebSphere MQ Using Java*.

The table specifies the data type of each property and indicates how the value of the property is set for a transmitted message. Note that some of the properties are set automatically by XMS when an application sends a message.

Table 11. IBM defined properties of a message

XMS name of the IBM defined property	WebSphere MQ JMS name	Data type	How the value is set for a transmitted message *method* [*class*])
JMS_IBM_CHARACTER_SET	JMS_IBM_Character_Set	xmsINT	Set Integer Property [Message]
JMS_IBM_ENCODING	JMS_IBM_Encoding	xmsINT	Set Integer Property [Message]
JMS_IBM_FEEDBACK	JMS_IBM_Feedback	xmsINT	Set Integer Property [Message]

Table 11. IBM defined properties of a message (continued)

XMS name of the IBM defined property	WebSphere MQ JMS name	Data type	How the value is set for a transmitted message *method* [*class*])
JMS_IBM_FORMAT	JMS_IBM_Format	String[1]	Set String Property [Message]
JMS_IBM_LAST_MSG_IN_GROUP	JMS_IBM_Last_Msg_In_Group	xmsINT	Set Integer Property [Message]
JMS_IBM_MSGTYPE	JMS_IBM_MsgType	xmsINT	Set Integer Property [Message]
JMS_IBM_PUTAPPLTYPE	JMS_IBM_PutApplType	xmsINT	Send [MessageProducer]
JMS_IBM_PUTDATE	JMS_IBM_PutDate	String[1]	Send [MessageProducer]
JMS_IBM_PUTTIME	JMS_IBM_PutTime	String[1]	Send [MessageProducer]
JMS_IBM_REPORT_COA	JMS_IBM_Report_COA	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_COD	JMS_IBM_Report_COD	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_DISCARD_MSG	JMS_IBM_Report_Discard_Msg	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_EXCEPTION	JMS_IBM_Report_Exception	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_EXPIRATION	JMS_IBM_Report_Expiration	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_NAN	JMS_IBM_Report_NAN	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_PAN	JMS_IBM_Report_PAN	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_PASS_CORREL_ID	JMS_IBM_Report_Pass_Correl_ID	xmsINT	Set Integer Property [Message]
JMS_IBM_REPORT_PASS_MSG_ID	JMS_IBM_Report_Pass_Msg_ID	xmsINT	Set Integer Property [Message]

Notes:

1. This is the data type if you are using C++. If you are programming in C, it is a character array.

Application defined properties of a message

An XMS application can create and use its own set of message properties. When an application sends a message, these properties are also transmitted with the message. A receiving application, using message selectors, can then select which messages it wants to receive based on the values of these properties.

To allow a JMS application to select and process messages sent by an XMS application, the name of an application defined property must conform to the rules documented in *WebSphere MQ Using Java*. The value of an application defined property must have one of the following data types: xmsBOOL, xmsBYTE, xmsSHORT, xmsINT, xmsLONG, xmsFLOAT, xmsDOUBLE, or String (or character array, if you are using the C interface).

The body of an XMS message

The body of a message contains application data. However, a message can have no body, and comprise only the header fields and properties.

The first release of XMS supports two types of message body:

Bytes The body contains a stream of bytes. A message with this type of body is called a *bytes message*. The BytesMessage class contains the methods to process the body of a bytes message. For more information about bytes messages, see “Bytes messages” on page 52.

Map The body contains a set of name-value pairs. A message with this type of

body is called a *map message*. The MapMessage class contains the methods to process the body of a map message. For more information about map messages, see "Map messages" on page 52.

In the C interface, XMS returns a message handle to an application when the application creates a bytes message or map message. The application can use this handle to call any of the methods of the Message class, and any of the methods of the BytesMessage or MapMessage class, whichever is appropriate for the type of message body. However, if an application tries to call a method that is inappropriate for the type of message body, the call fails and XMS returns error code XMS_E_BAD_PARAMETER.

A C application can use the Get Type method of the Message class to determine the body type of a message. The Get Type method returns one of the following values:

XMSC_T_BYTES_MSG
: If the message is a bytes message

XMSC_T_MAP_MSG
: If the message is a map message

XMSC_T_MSG
: If the message has no body.

See the following fragment of C code, for example:

```
xmsMESSAGE_TYPE msgtype;
xmsMsgConsumerReceive(messageConsumer, &msg, errorBlock);
xmsMsgGetTypeId(msg, &msgtype, errorBlock);
if (msgtype == XMSC_T_BYTES_MSG)
{
  xmsBytesMsgGetBodyLength(msg, &length, errorBlock);
}
```

In the C++ interface, BytesMessage and MapMessage are subclasses of the Message class.

To ensure that XMS applications can exchange messages with WebSphere MQ JMS applications, an XMS application and a WebSphere MQ JMS application must be able to interpret the application data in the body of a message in the same way. For this reason, each element of application data written in the body of a message by an XMS application must have one of the data types listed in Table 12. For each XMS data type, the table shows the compatible Java data type. XMS provides the methods to write elements of application data with these data types, and only these data types.

Table 12. XMS data types that are compatible with Java data types

XMS data type	Represents	Size	Compatible Java data type
xmsBOOL	The boolean value xmsTRUE or xmsFALSE	32 bits	boolean
xmsCHAR16	Double byte character	16 bits	char
xmsBYTE	Signed 8-bit integer	8 bits	byte
xmsSHORT	Signed 16-bit integer	16 bits	short
xmsINT	Signed 32-bit integer	32 bits	int
xmsLONG	Signed 64-bit integer	64 bits	long
xmsFLOAT	Signed floating point number	32 bits	float

Table 12. XMS data types that are compatible with Java data types (continued)

XMS data type	Represents	Size	Compatible Java data type
xmsDOUBLE	Signed double precision floating point number	64 bits	double
String[1]	String of characters	-	String

Notes:

1. This is the data type if you are using C++. If you are programming in C, it is a character array.

Bytes messages

The body of a bytes message contains a stream of bytes. The body contains no data names or data type information, only the actual data, and it is entirely the responsibility of the sending and receiving applications to interpret this data. Bytes messages are particularly useful if an XMS application needs to exchange messages with applications that are not using the XMS or JMS application programming interface.

After an application creates a bytes message, the body of the message is write-only. The application assembles the application data into the body by calling the appropriate write methods of the BytesMessage class. Each time the application writes a value to the bytes message stream, the value is assembled immediately after the previous value written by the application. XMS maintains an internal cursor to remember the position of the last byte that was assembled.

When the application sends the message, the body of the message becomes read-only. In this mode, the application can send the message multiple times.

When an application receives a bytes message, the body of the message is read-only. The application can use the appropriate read methods of the BytesMessage class to read the contents of the bytes message stream. The application reads the bytes in sequence, and XMS maintains an internal cursor to remember the position of the last byte that was read.

Using the C interface only, an application can skip over bytes without reading them by calling a read function with a null pointer for the `value` parameter, or for the `buffer` parameter if the application calls Read Bytes. For information about how to skip over bytes when calling Read UTF String, see "Read UTF String" on page 68.

If an application calls the Reset method of the BytesMessage class when the body of a bytes message is write-only, the body becomes read-only. The method also repositions the cursor at the beginning of the bytes message stream.

If an application calls the Clear Body method of the Message class when the body of a bytes message is read-only, the body becomes write-only. The method also clears the body.

Map messages

The body of a map message contains a set of name-value pairs. In each name-value pair, the name is a string that identifies the value, and the value is an element of application data that has one of the XMS data types listed in Table 12 on page 51

on page 51. The order of the name-value pairs is not defined. The MapMessage class contains the methods to set and get name-value pairs.

An application can access a name-value pair randomly by specifying its name. Alternatively, the application can access the name-value pairs sequentially using an iterator. The application can call the Get Name-Value Pairs method of the MapMessage class to create an iterator that encapsulates a list of Property objects, where each Property object encapsulates a name-value pair. The application can then use the methods of the Iterator class to retrieve each Property object in turn, and use the methods of the Property class to retrieve the name, data type, and value of each name-value pair. Although a name-value pair is not a property, the methods of the Property class treat a name-value pair like a property.

After an application creates a map message, the body of the message is readable and writable. The body remains readable and writable after the application sends the message. When an application receives a map message, the body of the message is read-only. If an application calls the Clear Body method of the Message class when the body of a map message is read-only, the body becomes readable and writable. The method also clears the body.

Message selectors

An XMS application uses messages selectors to select which messages it wants to receive.

When an application creates a message consumer, it can associate a message selector expression with the consumer. The message selector expression specifies the selection criteria. XMS determines whether each incoming message satisfies the selection criteria. If a message satisfies the selection criteria, XMS delivers the message to the message consumer. If a message does not satisfy the selection criteria, XMS does not deliver the message.

An application can create more than one message consumer, each with its own message selector expression. If an incoming message satisfies the selection criteria of more than one message consumer, XMS delivers the message to each of these consumers.

A message selector expression can reference the following properties of a message:
- JMS defined properties
- IBM defined properties
- Application defined properties

It can also reference the following message header fields:
- JMSCorrelationID
- JMSDeliveryMode
- JMSMessageID
- JMSPriority
- JMSTimestamp
- JMSType

A message selector expression, however, cannot reference data in the body of a message.

Here is an example of a message selector expression:

```
JMSPriority > 3 AND manufacturer = 'Jaguar' AND model in ('xj6','xj12')
```

XMS delivers a message to a message consumer with this message selector expression only if the message has a priority greater than 3, an application defined property, manufacturer, with a value of Jaguar, and another application defined property, model, with a value of xj6 or xj12.

The syntax rules for forming a message selector expression in XMS are the same as those in WebSphere MQ JMS. For information about how to construct a message selector expression therefore, see *WebSphere MQ Using Java*. Note, in particular, that, in a message selector expression, the names of JMS defined properties must be the JMS names, and the names of IBM defined properties must be the WebSphere MQ JMS names. You cannot use the XMS names in a message selector expression.

Part 3. XMS API reference

Chapter 8. XMS classes 59
BytesMessage. 61
Methods 61
Get Body Length 61
Read Boolean Value 61
Read Byte 62
Read Bytes 62
Read Bytes by Reference 63
Read Character 64
Read Double Precision Floating Point Number 65
Read Floating Point Number 65
Read Integer 66
Read Long Integer 66
Read Short Integer 67
Read Unsigned Byte 67
Read Unsigned Short Integer 68
Read UTF String. 68
Reset 69
Write Boolean Value 70
Write Byte 70
Write Bytes 71
Write Character 71
Write Double Precision Floating Point Number 72
Write Floating Point Number 72
Write Integer 73
Write Long Integer 73
Write Short Integer 74
Write UTF String 74
Inherited methods in the C++ interface 75
Connection 76
Methods 76
Check Whether Null 76
Close Connection 76
Create Session 77
Get Exception Listener. 77
Get Handle 78
Get Metadata 78
Get Property 79
Set Exception Listener 79
Set Property 80
Start Connection. 81
Stop Connection 81
Inherited methods in the C++ interface 81
ConnectionFactory 83
Constructor 83
Create Connection Factory 83
Methods 83
Check Whether Null 83
Create Connection (using the default user identity) 84
Create Connection (using a specified user identity) 84
Delete Connection Factory 85
Get Boolean Property 85
Get Handle 86
Get Integer Property 86
Get Property 87
Get String Property. 87
Get String Property by Reference 88
Set Boolean Property 89
Set Integer Property 89
Set Property 90
Set String Property 90
Inherited methods in the C++ interface 91
ConnectionMetaData 92
Methods 92
Check Whether Null 92
Delete Connection Metadata. 92
Get Handle 93
Get Integer Property 93
Get Property 93
Get String Property. 94
Get String Property by Reference 95
Inherited methods in the C++ interface 95
Destination 96
Constructors 96
Create Destination (using a URI) 96
Create Destination (specifying a type and name) 96
Methods 97
Check Whether Null 97
Delete Destination 97
Get Destination Name. 98
Get Destination Name as URI 98
Get Destination Type 99
Get Handle 100
Get Integer Property 100
Get Property 100
Get String Property 101
Get String Property by Reference. 102
Set Integer Property 102
Set Property 103
Set String Property 103
Inherited methods in the C++ interface. . . . 104
ErrorBlock 105
Methods 105
Clear Error Block 105
Create Error Block. 105
Delete Error Block. 105
Get Error Code 106
Get Error Data 106
Get Error Module 107
Get Error String 107
Get Exception Code 108
Get Linked Error 108
Exception. 109
Methods 109
Check Whether Null 109
Delete Exception 109
Dump Exception 110
Get Error Code 110
Get Error Data 110

© Copyright IBM Corp. 2004

Get Error String 111
Get Exception Code 111
Get Handle 111
Get Linked Exception. 111
ExceptionListener 113
Methods 113
On Exception 113
IllegalStateException 114
Inherited methods 114
InvalidDestinationException 115
Inherited methods 115
InvalidSelectorException. 116
Inherited methods 116
Iterator 117
Methods 117
Check for More Properties 117
Check Whether Null 117
Delete Iterator 118
Get Handle 118
Get Next Property 119
Reset Iterator 119
MapMessage 120
Methods 120
Check Name-Value Pair Exists. 120
Get Boolean Value. 120
Get Byte 121
Get Bytes. 122
Get Bytes by Reference 123
Get Character 123
Get Double Precision Floating Point Number 124
Get Floating Point Number. 124
Get Integer 125
Get Long Integer 126
Get Name-Value Pairs 126
Get Object 127
Get Short Integer 128
Get String 129
Get String by Reference 130
Set Boolean Value 130
Set Byte 131
Set Bytes 131
Set Character 132
Set Double Precision Floating Point Number 133
Set Floating Point Number 133
Set Integer 134
Set Long Integer 135
Set Object 135
Set Short Integer 136
Set String. 137
Inherited methods in the C++ interface. . . . 137
Message 139
Methods 139
Check Property Exists 139
Check Whether Null 139
Clear Body 140
Clear Properties 140
Delete Message. 141
Get Boolean Property. 141
Get Byte Property 142
Get Double Precision Floating Point Property 142
Get Floating Point Property 143
Get Handle 143
Get Integer Property 144
Get JMSCorrelationID 144
Get JMSDeliveryMode 145
Get JMSDestination 145
Get JMSExpiration. 146
Get JMSMessageID 147
Get JMSPriority 148
Get JMSRedelivered 148
Get JMSReplyTo 149
Get JMSTimestamp 149
Get JMSType 150
Get Long Integer Property 151
Get Object Property 151
Get Properties 153
Get Property 153
Get Short Integer Property 154
Get String Property 154
Get String Property by Reference. 155
Get Type 156
Set Boolean Property 156
Set Byte Property 157
Set Double Precision Floating Point Property 157
Set Floating Point Property 158
Set Integer Property 158
Set JMSCorrelationID. 159
Set JMSDeliveryMode 159
Set JMSDestination 160
Set JMSExpiration 161
Set JMSMessageID. 161
Set JMSPriority 162
Set JMSRedelivered 162
Set JMSReplyTo 163
Set JMSTimestamp. 164
Set JMSType. 164
Set Long Integer Property 165
Set Object Property 165
Set Property 166
Set Short Integer Property 167
Set String Property 167
Inherited methods in the C++ interface. . . . 168
MessageConsumer 169
Methods 169
Check Whether Null 169
Close Message Consumer 169
Get Handle 170
Get Message Listener. 170
Get Message Selector 171
Get Property 171
Receive 172
Receive (with a wait interval) 173
Receive with No Wait 173
Set Message Listener 174
Set Property 175
Inherited methods in the C++ interface. . . . 175
MessageEOFException 176
Inherited methods. 176
MessageFormatException 177
Inherited methods. 177
MessageListener 178
Methods 178

On Message 178
MessageNotReadableException 179
Inherited methods. 179
MessageNotWritableException. 180
Inherited methods. 180
MessageProducer 181
Methods 181
Check Whether Null 181
Close Message Producer. 181
Get Default Delivery Mode. 182
Get Default Priority 182
Get Default Time to Live 183
Get Destination. 183
Get Disable Message ID Flag 184
Get Disable Timestamp Flag 184
Get Handle 185
Get Property 185
Send 186
Send (specifying a delivery mode, priority, and time to live) 186
Send (to a specified destination) 187
Send (to a specified destination, specifying a delivery mode, priority, and time to live) . . 188
Set Default Delivery Mode 189
Set Default Priority 190
Set Default Time to Live. 190
Set Disable Message ID Flag 191
Set Disable Timestamp Flag 191
Set Property 192
Inherited methods in the C++ interface. . . . 192
Property 193
Constructors. 193
Copy Property 193
Create Property 193
Create Property (with no property value or property type) 194
Methods 195
Check Whether Null 195
Check Property Type 195
Delete Property. 196
Get Boolean Property Value 197
Get Byte Array Property Value 197
Get Byte Array Property Value by Reference 198
Get Byte Property Value. 199
Get Character Property Value 199
Get Double Precision Floating Point Property Value 200
Get Floating Point Property Value 200
Get Handle 201
Get Integer Property Value 201
Get Long Integer Property Value 202
Get Property Name 202
Get Property Type. 203
Get Short Integer Property Value 204
Get String Property Value 204
Get String Property Value by Reference. . . 205
Set Boolean Property Value. 205
Set Byte Array Property Value. 206
Set Byte Property Value 206
Set Character Property Value 207
Set Double Precision Floating Point Property Value 208
Set Floating Point Property Value. 208
Set Integer Property Value 209
Set Long Integer Property Value 209
Set Short Integer Property Value 210
Set String Property Value 210
PropertyContext 212
Methods 212
Get Boolean Property. 212
Get Byte Property 212
Get Byte Array Property. 213
Get Character Property 213
Get Double Precision Floating Point Property 214
Get Floating Point Property 214
Get Integer Property 214
Get Long Integer Property 215
Get Short Integer Property 215
Get String Property 215
Set Boolean Property 216
Set Byte Property 216
Set Byte Array Property 217
Set Character Property 217
Set Double Precision Floating Point Property 218
Set Floating Point Property 218
Set Integer Property 218
Set Long Integer Property 219
Set Short Integer Property 219
Set String Property 220
ResourceAllocationException 221
Inherited methods. 221
SecurityException 222
Inherited methods. 222
Session 223
Methods 223
Check Whether Null 223
Close Session 223
Create Bytes Message. 224
Create Consumer 224
Create Consumer (with message selector) . . 225
Create Consumer (with message selector and local message flag) 226
Create Map Message 227
Create Message. 227
Create Producer 228
Create Topic. 228
Get Acknowledgement Mode 229
Get Handle 229
Get Property 230
Set Property 230
Inherited methods in the C++ interface. . . . 231
String 232
Constructors. 232
Create String 232
Create String (from a byte array) 232
Create String (from a character array) . . . 232
Methods 233
Check Whether Null 233
Compare Strings 233
Concatenate Strings 233
Delete String 234

Get Pointer to String 234
Get String 234

Chapter 9. Properties of XMS objects 237
Properties of Connection 237
Properties of ConnectionFactory 237
Properties of ConnectionMetaData 238
Properties of Destination 239
Properties of MessageConsumer 240
Properties of Session 241

Chapter 8. XMS classes

This chapter documents the XMS classes and their methods. Table 13 summarizes all the classes.

Table 13. A summary of the XMS classes

Class	Description	Page
BytesMessage	A bytes message is a message whose body comprises a stream of bytes.	61
Connection	A Connection object represents an application's active connection to a broker.	76
ConnectionFactory	An application uses a connection factory to create a connection.	83
ConnectionMetaData	A ConnectionMetaData object provides information about a connection.	92
Destination	A destination is where an application sends messages, or it is a source from which an application receives messages, or both.	96
ErrorBlock	If a C function call fails, XMS can store information in an error block about why the call failed.	105
Exception	If XMS detects an error while processing a call to a C++ method, XMS throws an exception. An exception is an object that encapsulates information about the error. There are different types of XMS exception, and an Exception object is just one type of exception. However, the Exception class is a superclass of the other XMS exception classes. XMS throws an Exception object in situations where none of the other types of exception are appropriate.	109
ExceptionListener	An application uses an exception listener to be notified asynchronously of a problem with a connection.	113
IllegalStateException	XMS throws this exception if an application calls a method at an incorrect or inappropriate time, or if XMS is not in an appropriate state for the requested operation.	114
InvalidDestinationException	XMS throws this exception if an application specifies a destination that is not valid.	115
InvalidSelectorException	XMS throws this exception if an application provides a message selector expression whose syntax is not valid.	116
Iterator	An iterator encapsulates a list of Property objects. An application uses an iterator to retrieve each Property object in turn.	117
MapMessage	A map message is a message whose body comprises a set of name-value pairs.	120
Message	A Message object represents a message that an application sends or receives.	139
MessageConsumer	An application uses a message consumer to receive messages sent to a destination.	169
MessageEOFException	XMS throws this exception if XMS encounters the end of a bytes message stream when an application is reading the body of a bytes message.	176
MessageFormatException	XMS throws this exception if XMS encounters a message with a format that is not valid.	177
MessageListener	An application uses a message listener to receive messages asynchronously.	178
MessageNotReadableException	XMS throws this exception if an application attempts to read the body of a message that is write-only.	179

© Copyright IBM Corp. 2004

Table 13. A summary of the XMS classes (continued)

Class	Description	Page
MessageNotWritableException	XMS throws this exception if an application attempts to write to the body of a message that is read-only.	180
MessageProducer	An application uses a message producer to send messages to a destination.	181
Property	A Property object represents a property of an object.	193
PropertyContext	PropertyContext is an abstract superclass that encapsulates methods that get and set properties. These methods are inherited by other classes.	212
ResourceAllocationException	XMS throws this exception if XMS cannot allocate the resources required by a method.	221
SecurityException	XMS throws this exception if the user identifer and password provided to authenticate an application are rejected. XMS also throws this exception if an authority check fails and prevents a method from completing.	222
Session	A session is a single threaded context for sending and receiving messages.	223
String	A String object encapsulates a string.	232

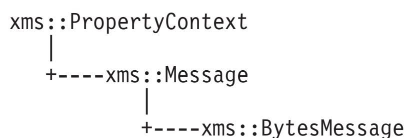
The definition of each method lists the exception codes that XMS might return if it detects an error while processing a call to the C function or the C++ method. Each exception code is represented by its named constant. Table 14 lists the exception codes and their corresponding C++ exceptions.

Table 14. Exception codes and their corresponding C++ exceptions

Exception code	Corresponding C++ exception
XMS_X_GENERAL_EXCEPTION	Exception
XMS_X_ILLEGAL_STATE_EXCEPTION	IllegalStateException
XMS_X_INVALID_DESTINATION_EXCEPTION	InvalidDestinationException
XMS_X_INVALID_SELECTOR_EXCEPTION	InvalidSelectorException
XMS_X_MESSAGE_EOF_EXCEPTION	MessageEOFException
XMS_X_MESSAGE_FORMAT_EXCEPTION	MessageFormatException
XMS_X_MESSAGE_NOT_READABLE_EXCEPTION	MessageNotReadableException
XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION	MessageNotWritableException
XMS_X_RESOURCE_ALLOCATION_EXCEPTION	ResourceAllocationException
XMS_X_SECURITY_EXCEPTION	SecurityException

BytesMessage

C++ inheritance hierarchy:



A bytes message is a message whose body comprises a stream of bytes.

Methods

Get Body Length

C interface:
```
xmsRC xmsBytesMsgGetBodyLength(xmsHMsg message,
                               xmsLONG *bodyLength,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsLONG getBodyLength() const;
```

Get the length of the body of the message when the body of the message is read-only.

Parameters:

message (input)
The handle for the message.

bodyLength (output)
The length of the body of the message in bytes. The call returns the length of the whole body regardless of where the cursor for reading the message is currently positioned.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the bodyLength parameter.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

Read Boolean Value

C interface:
```
xmsRC xmsBytesMsgReadBoolean(xmsHMsg message,
                             xmsBOOL *value,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBOOL readBoolean() const;
```

Read a boolean value from the bytes message stream.

Parameters:

message (input)
The handle for the message.

value (output)
The boolean value that is read. If you specify a null pointer on input, the call skips over the boolean value without reading it.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The boolean value that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Byte

C interface:
```
xmsRC xmsBytesMsgReadByte(xmsHMsg message,
                          xmsBYTE *value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBYTE readByte() const;
```

Read the next byte from the bytes message stream as a signed 8-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The byte that is read. If you specify a null pointer on input, the call skips over the byte without reading it.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The byte that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Bytes

C interface:
```
xmsRC xmsBytesMsgReadBytes(xmsHMsg message,
                           xmsBYTE *buffer,
                           xmsSIZE bufferLength,
                           xmsSIZE *returnedLength,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT readBytes(xmsBYTE *buffer,
                 const xmsSIZE bufferLength,
                 xmsSIZE *returnedLength) const;
```

Read an array of bytes from the bytes message stream starting from the current position of the cursor.

Parameters:

message (input)
The handle for the message.

buffer (output)
The buffer to contain the array of bytes that is read. If the number of bytes remaining to be read from the stream before the call is greater than or equal to the length of the buffer, the buffer is filled. Otherwise, the buffer is partially filled with all the remaining bytes.

If you specify a null pointer on input, the call skips over the bytes without reading them. If the number of bytes remaining to be read from the stream before the call is greater than or equal to the length of the buffer, the number of bytes skipped is equal to the length of the buffer. Otherwise, all the remaining bytes are skipped.

bufferLength (input)
The length of the buffer in bytes.

returnedLength (output)
The number of bytes that are read into the buffer. If the buffer is partially filled, the value is less than the length of the buffer, indicating that there are no more bytes remaining to be read. If there are no bytes remaining to be read from the stream before the call, the value is XMSC_END_OF_STREAM.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the returnedLength parameter.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

Read Bytes by Reference

C interface:

```
xmsRC xmsBytesMsgReadBytesByRef(xmsHMsg message,
                                xmsBYTE **stream,
                                xmsSIZE *length,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the start of the bytes message stream and get the length of the stream.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

message (input)
The handle for the message.

stream (output)
A pointer to the start of the bytes message stream.

length (output)
The number of bytes in the bytes message stream.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Character

C interface:

```
xmsRC xmsBytesMsgReadChar(xmsHMsg message,
                          xmsCHAR16 *value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsCHAR16 readChar() const;
```

Read the next 2 bytes from the bytes message stream as a character.

Parameters:

message (input)
The handle for the message.

value (output)
The character that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The character that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Double Precision Floating Point Number

C interface:

```
xmsRC xmsBytesMsgReadDouble(xmsHMsg message,
                            xmsDOUBLE *value,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsDOUBLE readDouble() const;
```

Read the next 8 bytes from the bytes message stream as a double precision floating point number.

Parameters:

message (input)
The handle for the message.

value (output)
The double precision floating point number that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The double precision floating point number that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Floating Point Number

C interface:

```
xmsRC xmsBytesMsgReadFloat(xmsHMsg message,
                           xmsFLOAT *value,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsFLOAT readFloat() const;
```

Read the next 4 bytes from the bytes message stream as a floating point number.

Parameters:

message (input)
The handle for the message.

value (output)
The floating point number that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The floating point number that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Integer

C interface:

```
xmsRC xmsBytesMsgReadInt(xmsHMsg message,
                         xmsINT *value,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT readInt() const;
```

Read the next 4 bytes from the bytes message stream as a signed 32-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The integer that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Long Integer

C interface:

```
xmsRC xmsBytesMsgReadLong(xmsHMsg message,
                          xmsLONG *value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsLONG readLong() const;
```

Read the next 8 bytes from the bytes message stream as a signed 64-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The long integer that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The long integer that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Short Integer

C interface:
```
xmsRC xmsBytesMsgReadShort(xmsHMsg message,
                           xmsSHORT *value,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsSHORT readShort() const;
```

Read the next 2 bytes from the bytes message stream as a signed 16-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The short integer that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The short integer that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Unsigned Byte

C interface:
```
xmsRC xmsBytesMsgReadUnsignedByte(xmsHMsg message,
                                  xmsUINT8 *value,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsUINT8 readUnsignedByte() const;
```

Read the next byte from the bytes message stream as an unsigned 8-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The byte that is read. If you specify a null pointer on input, the call skips over the byte without reading it.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The byte that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read Unsigned Short Integer

C interface:

```
xmsRC xmsBytesMsgReadUnsignedShort(xmsHMsg message,
                                   xmsUINT16 *value,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsUINT16 readUnsignedShort() const;
```

Read the next 2 bytes from the bytes message stream as an unsigned 16-bit integer.

Parameters:

message (input)
The handle for the message.

value (output)
The unsigned short integer that is read. If you specify a null pointer on input, the call skips over the bytes without reading them.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The unsigned short integer that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Read UTF String

C interface:

```
xmsRC xmsBytesMsgReadUTF(xmsHMsg message,
                         xmsCHAR *buffer,
                         xmsSIZE bufferLength,
                         xmsSIZE *actualLength,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
String readUTF() const;
```

Read a string, encoded in UTF-8, from the bytes message stream. If required, XMS converts the characters in the string from UTF-8 into the local code page.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33.

Parameters:

message (input)
The handle for the message.

buffer (output)
The buffer to contain the string that is read. If data conversion is required, this is the string after conversion.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, or the value `XMSC_QUERY_LENGTH`, the string is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the string in bytes. If data conversion is required, this is the length of the string after conversion.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the string that is read. If data conversion is required, this is the string after conversion.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Notes:
1. To skip over a string without reading it, call Read UTF String with the following parameter settings:
   - Set the buffer parameter to the null pointer.
   - Set the bufferLength parameter to the value `XMSC_SKIP`.
   - Set the actualLength parameter to the null pointer.
2. If the buffer is not large enough to store the whole string, XMS returns the string truncated to the length of the buffer, sets the actualLength parameter to the actual length of the string, and returns error code XMS_E_DATA_TRUNCATED. XMS does not advance the internal cursor.
3. If any other error occurs while attempting to read the string, XMS reports the error but does not set the actualLength parameter or advance the internal cursor.

Reset

C interface:
```
xmsRC xmsBytesMsgReset(xmsHMsg message,
                       xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID reset() const;
```

Put the body of the message into read-only mode and reposition the cursor at the beginning of the bytes message stream.

Parameters:

message (input)
The handle for the message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Write Boolean Value

C interface:

```
xmsRC xmsBytesMsgWriteBoolean(xmsHMsg message,
                              xmsBOOL value,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeBoolean(const xmsBOOL value);
```

Write a boolean value to the bytes message stream.

Parameters:

message (input)
The handle for the message.

value (input)
The boolean value to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Byte

C interface:

```
xmsRC xmsBytesMsgWriteByte(xmsHMsg message,
                           xmsBYTE value,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeByte(const xmsBYTE value);
```

Write a byte to the bytes message stream.

Parameters:

message (input)
The handle for the message.

value (input)
The byte to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Bytes

C interface:

```
xmsRC xmsBytesMsgWriteBytes(xmsHMsg message,
                            xmsBYTE *value,
                            xmsSIZE length,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeBytes(const xmsBYTE *value,
                   const xmsSIZE length);
```

Write an array of bytes to the bytes message stream.

Parameters:

message (input)
The handle for the message.

value (input)
The array of bytes to be written.

length (input)
The number of bytes in the array.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Character

C interface:

```
xmsRC xmsBytesMsgWriteChar(xmsHMsg message,
                           xmsCHAR16 value,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeChar(const xmsCHAR16 value);
```

Write a character to the bytes message stream as 2 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The character to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Double Precision Floating Point Number

C interface:

```
xmsRC xmsBytesMsgWriteDouble(xmsHMsg message,
                             xmsDOUBLE value,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeDouble(const xmsDOUBLE value);
```

Convert a double precision floating point number to a long integer and write the long integer to the bytes message stream as 8 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The double precision floating point number to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Floating Point Number

C interface:

```
xmsRC xmsBytesMsgWriteFloat(xmsHMsg message,
                            xmsFLOAT value,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeFloat(const xmsFLOAT value);
```

Convert a floating point number to an integer and write the integer to the bytes message stream as 4 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The floating point number to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Integer

C interface:

```
xmsRC xmsBytesMsgWriteInt(xmsHMsg message,
                          xmsINT value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeInt(const xmsINT value);
```

Write an integer to the bytes message stream as 4 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The integer to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Long Integer

C interface:

```
xmsRC xmsBytesMsgWriteLong(xmsHMsg message,
                           xmsLONG value,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeLong(const xmsLONG value);
```

Write a long integer to the bytes message stream as 8 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The long integer to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write Short Integer

C interface:

```
xmsRC xmsBytesMsgWriteShort(xmsHMsg message,
                            xmsSHORT value,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeShort(const xmsSHORT value);
```

Write a short integer to the bytes message stream as 2 bytes, high order byte first.

Parameters:

message (input)
The handle for the message.

value (input)
The short integer to be written.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Write UTF String

C interface:

```
xmsRC xmsBytesMsgWriteUTF(xmsHMsg message,
                          xmsCHAR *value,
                          xmsSIZE length,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID writeUTF(const String & value);
```

Write a string, encoded in UTF-8, to the bytes message stream. If required, XMS converts the characters in the string from the local code page into UTF-8.

Parameters:

message (input)
The handle for the message.

value (input)
In the C interface, this parameter is the string to be written.

In the C++ interface, this parameter is a String object encapsulating the string to be written.

length (input)
The length of the string in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the Message class:

clearBody, clearProperties, getHandle, getJMSCorrelationID, getJMSDeliveryMode, getJMSDestination, getJMSExpiration, getJMSMessageID, getJMSPriority, getJMSRedelivered, getJMSReplyTo, getJMSTimestamp, getJMSType, getObjectProperty, getProperties, getProperty, isNull, propertyExists, setJMSCorrelationID, setJMSDeliveryMode, setJMSDestination, setJMSExpiration, setJMSMessageID, setJMSPriority, setJMSRedelivered, setJMSReplyTo, setJMSTimestamp, setJMSType, setObjectProperty, setProperty

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

Connection

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::Connection
```

A Connection object represents an application's active connection to a broker.

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the Connection object is a null object.

Parameters:
None

C++ method returns:
- xmsTRUE, if the Connection object is a null object.
- xmsFALSE, if the Connection object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Close Connection

C interface:
```
xmsRC xmsConnClose(xmsHConn *connection,
                   xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID close();
```

Close the connection.

If an application tries to close a connection that is already closed, the call is ignored.

Parameters:

connection (input/output)
On input, the handle for the connection. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Session

C interface:

```
xmsRC xmsConnCreateSession(xmsHConn connection,
                           xmsBOOL transacted,
                           xmsACKNOWLEDGE_MODE acknowledgeMode,
                           xmsHSess *session,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
Session createSession(const xmsBOOL transacted = xmsFALSE,
                      const xmsACKNOWLEDGE_MODE
                      acknowledgeMode = XMSC_AUTO_ACKNOWLEDGE);
```

Create a session.

Parameters:

connection (input)
The handle for the connection.

transacted (input)
Indicates whether the session is transacted. The value must be xmsFALSE.

acknowledgeMode (input)
Indicates how messages received by an application are acknowledged. The value must be XMSC_AUTO_ACKNOWLEDGE.

session (output)
The handle for the session.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Session object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Exception Listener

C interface:

```
xmsRC xmsConnGetExceptionListener(xmsHConn connection,
                                  fpXMS_EXCEPTION_CALLBACK *lsr,
                                  xmsCONTEXT *context,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
ExceptionListener * getExceptionListener() const;
```

In a C application, get pointers to the exception listener function and context data that are registered with the connection.

In a C++ application, get a pointer to the exception listener that is registered with the connection.

For more information about using exception listener functions in a C application, see “Using exception listener functions in C” on page 32. If you are using C++, see “Using exception listeners in C++” on page 43 instead.

Parameters:

connection (input)
The handle for the connection.

lsr (output)
A pointer to the exception listener function. If no exception listener function is registered with the connection, the call returns a null pointer.

context (output)
A pointer to the context data. If no exception listener function is registered with the connection, the call returns a null pointer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to the exception listener. If no exception listener is registered with the connection, the call returns a null pointer.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:

```
xmsHConn getHandle() const;
```

Get the handle that a C application would use to access the connection.

Parameters:
None

C++ method returns:
The handle for the connection.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Metadata

C interface:

```
xmsRC xmsConnGetMetaData(xmsHConn connection,
                         xmsHConnMetaData *connectionMetaData,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
ConnectionMetaData getMetaData() const;
```

Get the metadata for the connection.

Parameters:

connection (input)
The handle for the connection.

connectionMetaData (output)
The handle for the connection metadata.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The ConnectionMetaData object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:

```
xmsRC xmsConnGetProperty(xmsHConn connection,
                         xmsCHAR *propertyName,
                         xmsHProperty *property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

connection (input)
The handle for the connection.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Exception Listener

C interface:

```
xmsRC xmsConnSetExceptionListener(xmsHConn connection,
                                  fpXMS_EXCEPTION_CALLBACK lsr,
                                  xmsCONTEXT context,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setExceptionListener(const ExceptionListener *lsr);
```

In a C application, register an exception listener function and context data with the connection.

In a C++ application, register an exception listener with the connection.

For more information about using exception listener functions in a C application, see "Using exception listener functions in C" on page 32. If you are using C++, see "Using exception listeners in C++" on page 43 instead.

Parameters:

connection (input)
The handle for the connection.

lsr (input)
In the C interface, this parameter is a pointer to the exception listener function.

In the C++ interface, this parameter is a pointer to the exception listener.

If an exception listener function or exception listener is already registered with the connection, you can cancel the registration by specifying a null pointer instead.

context (input)
A pointer to the context data.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:

```
xmsRC xmsConnSetProperty(xmsHConn connection,
                         xmsHProperty property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

connection (input)
The handle for the connection.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Start Connection

C interface:
```
xmsRC xmsConnStart(xmsHConn connection,
                   xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID start() const;
```

Start, or restart, the delivery of incoming messages for the connection. The call is ignored if the connection is already started.

Parameters:

connection (input)
The handle for the connection.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Stop Connection

C interface:
```
xmsRC xmsConnStop(xmsHConn connection,
                  xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID stop() const;
```

Stop temporarily the delivery of incoming messages for the connection. The call is ignored if the connection is already stopped.

Parameters:

connection (input)
The handle for the connection.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

ConnectionFactory

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::ConnectionFactory
```

An application uses a connection factory to create a connection.

For a list of the XMS defined properties of a ConnectionFactory object, see "Properties of ConnectionFactory" on page 237.

Constructor

Create Connection Factory

C interface:

```
xmsRC xmsConnFactCreate(xmsHConnFact *factory,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
ConnectionFactory();
```

Create a connection factory with the default properties.

Parameters:

factory (output)
The handle for the connection factory.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:

```
xmsBOOL isNull() const;
```

Determine whether the ConnectionFactory object is a null object.

Parameters:
None

C++ method returns:

- xmsTRUE, if the ConnectionFactory object is a null object.
- xmsFALSE, if the ConnectionFactory object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Connection (using the default user identity)

C interface:

```
xmsRC xmsConnFactCreateConnection(xmsHConnFact factory,
                                  xmsHConn *connection,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
Connection createConnection();
```

Create a connection using the default user identity. The connection factory properties XMSC_USER and XMSC_PASSWORD, if they are set, are used to authenticate the application. If these properties are not set, the connection is created without authenticating the client, provided the broker permits a connection without authentication.

The connection is created in stopped mode. No messages are delivered until Start Connection is called explicitly.

Parameters:

factory (input)
The handle for the connection factory.

connection (output)
The handle for the connection.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Connection object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_SECURITY_EXCEPTION

Create Connection (using a specified user identity)

C interface:

```
xmsRC xmsConnFactCreateConnectionForUser(xmsHConnFact factory,
                                         xmsCHAR *userID,
                                         xmsCHAR *password,
                                         xmsHConn *connection,
                                         xmsHErrorBock errorBlock);
```

C++ interface:

```
Connection createConnection(const String & userID,
                            const String & password);
```

Create a connection using a specified user identity. The specified user identifier and password are used to authenticate the application. The connection factory properties XMSC_USER and XMSC_PASSWORD, if they are set, are ignored.

The connection is created in stopped mode. No messages are delivered until Start Connection is called explicitly.

Parameters:

factory (input)
The handle for the connection factory.

userID (input)
The user identifier to be used to authenticate the application.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

password (input)
The password to be used to authenticate the application. The password is in the format of a null terminated string.

connection (output)
The handle for the connection.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Connection object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_SECURITY_EXCEPTION

Delete Connection Factory

C interface:

```
xmsRC xmsConnFactDispose(xmsHConnFact *factory,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual ~ConnectionFactory();
```

Delete the connection factory.

If an application tries to delete a connection factory that is already deleted, the call is ignored.

Parameters:

factory (input/output)
On input, the handle for the connection factory. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Boolean Property

C interface:

```
xmsRC xmsConnFactGetBooleanProperty(xmsHConnFact factory,
                                    xmsCHAR *propertyName,
                                    xmsBOOL *propertyValue,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the boolean property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:

```
xmsHConnFact getHandle() const;
```

Get the handle that a C application would use to access the connection factory.

Parameters:
None

C++ method returns:
The handle for the connection factory.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property

C interface:

```
xmsRC xmsConnFactGetIntProperty(xmsHConnFact factory,
                                xmsCHAR *propertyName,
                                xmsINT *propertyValue,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the integer property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsConnFactGetProperty(xmsHConnFact factory,
                             xmsCHAR *propertyName,
                             xmsHProperty *property,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
```
Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property

C interface:
```
xmsRC xmsConnFactGetStringProperty(xmsHConnFact factory,
                                   xmsCHAR *propertyName,
                                   xmsCHAR *propertyValue,
                                   xmsSIZE length,
                                   xmsSIZE *actualLength,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the string property identified by name.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
The buffer to contain the value of the property.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the value of the property is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property by Reference

C interface:

```
xmsRC xmsConnFactGetStringPropertyByRef(xmsHConnFact factory,
                                        xmsCHAR *propertyName,
                                        xmsCHAR **propertyValue,
                                        xmsSIZE *length,
                                        xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the value of the string property identified by name.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
A pointer to the value of the property.

length (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Boolean Property

C interface:

```
xmsRC xmsConnFactSetBooleanProperty(xmsHConnFact factory,
                                    xmsCHAR *propertyName,
                                    xmsBOOL propertyValue,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the boolean property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Integer Property

C interface:

```
xmsRC xmsConnFactSetIntProperty(xmsHConnFact factory,
                                xmsCHAR *propertyName,
                                xmsINT propertyValue,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the integer property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:

```
xmsRC xmsConnFactSetProperty(xmsHConnFact factory,
                             xmsHProperty property,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

factory (input)
The handle for the connection factory.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set String Property

C interface:

```
xmsRC xmsConnFactSetStringProperty(xmsHConnFact factory,
                                   xmsCHAR *propertyName,
                                   xmsCHAR *propertyValue,
                                   xmsSIZE length,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the string property identified by name.

Parameters:

factory (input)
The handle for the connection factory.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(input)
The value of the property as a character array.

length (input)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

ConnectionMetaData

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::ConnectionMetaData
```

A ConnectionMetaData object provides information about a connection.

For a list of the XMS defined properties of a ConnectionMetaData object, see "Properties of ConnectionMetaData" on page 238.

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:

```
xmsBOOL isNull() const;
```

Determine whether the ConnectionMetaData object is a null object.

Parameters:
None

C++ method returns:

- `xmsTRUE`, if the ConnectionMetaData object is a null object.
- `xmsFALSE`, if the ConnectionMetaData object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Connection Metadata

C interface:

```
xmsRC xmsConnMetaDataDispose(xmsHConnMetaData *connectionMetaData,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual ~ConnectionMetaData();
```

Delete the metadata for the connection.

If an application tries to delete connection metadata that is already deleted, the call is ignored.

Parameters:

connectionMetaData (input/output)
On input, the handle for the connection metadata. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHConnMetaData getHandle() const;
```

Get the handle that a C application would use to access the connection metadata.

Parameters:
None

C++ method returns:
The handle for the connection metadata.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property

C interface:
```
xmsRC xmsConnMetaDataGetIntProperty(xmsHConnMetaData connectionMetaData,
                                    xmsCHAR *propertyName,
                                    xmsINT *propertyValue,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the integer property identified by name.

Parameters:

connectionMetaData (input)
The handle for the connection metadata.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsConnMetaDataGetProperty(xmsHConnMetaData connectionMetaData,
                                 xmsCHAR *propertyName,
                                 xmsHProperty *property,
                                 xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

connectionMetaData (input)
The handle for the connection metadata.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property

C interface:

```
xmsRC xmsConnMetaDataGetStringProperty(xmsHConnMetaData connectionMetaData,
                                       xmsCHAR *propertyName,
                                       xmsCHAR *propertyValue,
                                       xmsSIZE length,
                                       xmsSIZE *actualLength,
                                       xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the string property identified by name.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

connectionMetaData (input)
The handle for the connection metadata.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
The buffer to contain the value of the property.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the value of the property is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property by Reference

C interface:
```
xmsRC
xmsConnMetaDataGetStringPropertyByRef(xmsHConnMetaData connectionMetaData,
                                      xmsCHAR *propertyName,
                                      xmsCHAR **propertyValue,
                                      xmsSIZE *length,
                                      xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the value of the string property identified by name.

For more information about how to use this function, see "C functions that return a string or byte array by reference" on page 34.

Parameters:

connectionMetaData (input)
The handle for the connection metadata.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
A pointer to the value of the property.

length (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

Destination

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::Destination
```

A destination is where an application sends messages, or it is a source from which an application receives messages, or both.

For a list of the XMS defined properties of a Destination object, see “Properties of Destination” on page 239.

Constructors

Create Destination (using a URI)

C interface:

```
xmsRC xmsDestCreate(xmsCHAR *URI,
                    xmsHDest *destination,
                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
Destination(const String & URI);
```

Create a destination using the specified uniform resource identifier (URI). Properties of the destination that are not specified by the uniform resource identifier take the default values.

Parameters:

URI (input)
The uniform resource identifier.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

destination (output)
The handle for the destination.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Destination (specifying a type and name)

C interface:

```
xmsRC xmsDestCreateByType(xmsDESTINATION_TYPE destinationType,
                          xmsCHAR *destinationName,
                          xmsHDest *destination,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
Destination(const xmsDESTINATION_TYPE destinationType,
            const String & destinationName);
```

Create a destination using the specified destination type and name.

Parameters:

destinationType (input)
The type of the destination. The value must be `XMSC_TOPIC`.

destinationName (input)
The name of the destination, which can be the name of a topic or a topic URI.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

destination (output)
The handle for the destination.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the Destination object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the Destination object is a null object.
- `xmsFALSE`, if the Destination object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Destination

C interface:
```
xmsRC xmsDestDispose(xmsHDest *destination,
                     xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual ~Destination();
```

Delete the destination.

If an application tries to delete a destination that is already deleted, the call is ignored.

Parameters:

destination (input/output)
On input, the handle for the destination. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Destination Name

C interface:
```
xmsRC xmsDestGetName(xmsHDest destination,
                     xmsCHAR *destinationName,
                     xmsSIZE length,
                     xmsSIZE *actualLength,
                     xmsHErrorBlock errorBlock);
```

C++ interface:
```
String getDestinationName() const;
```

Get the name of the destination.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33.

Parameters:

destination (input)
The handle for the destination.

destinationName (output)
The buffer to contain the name of the destination.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the name of the destination is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the name of the destination in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the name of the destination.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Destination Name as URI

C interface:
```
xmsRC xmsDestToString(xmsHDest destination,
                      xmsCHAR *destinationName,
                      xmsSIZE length,
                      xmsSIZE *actualLength,
                      xmsHErrorBlock errorBlock);
```

C++ interface:

```
String toString() const;
```

Get the name of the destination in the format of a uniform resource identifier (URI).

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

destination (input)
The handle for the destination.

destinationName (output)
The buffer to contain the uniform resource identifier.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the URI is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the URI in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the uniform resource identifier.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Destination Type

C interface:

```
xmsRC xmsDestGetTypeId(xmsHDest destination,
                       xmsDESTINATION_TYPE *destinationType,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsDESTINATION_TYPE GetType();
```

Get the type of the destination.

Parameters:

destination (input)
The handle for the destination.

destinationType (output)
The type of the destination. The value is always `XMSC_TOPIC`.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the destinationType parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHDest getHandle() const;
```

Get the handle that a C application would use to access the destination.

Parameters:
None

C++ method returns:
The handle for the destination.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property

C interface:
```
xmsRC xmsDestGetIntProperty(xmsHDest destination,
                            xmsCHAR *propertyName,
                            xmsINT *propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the integer property identified by name.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsDestGetProperty(xmsHDest destination,
                         xmsCHAR *propertyName,
                         xmsHProperty *property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property

C interface:

```
xmsRC xmsDestGetStringProperty(xmsHDest destination,
                               xmsCHAR *propertyName,
                               xmsCHAR *propertyValue,
                               xmsSIZE length,
                               xmsSIZE *actualLength,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the string property identified by name.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
The buffer to contain the value of the property.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the value of the property is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property by Reference

C interface:
```
xmsRC xmsDestGetStringPropertyByRef(xmsHDest destination,
                                    xmsCHAR *propertyName,
                                    xmsCHAR **propertyValue,
                                    xmsSIZE *length,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the value of the string property identified by name.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue(output)
A pointer to the value of the property.

length (output)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Integer Property

C interface:
```
xmsRC xmsDestSetIntProperty(xmsHDest destination,
                            xmsCHAR *propertyName,
                            xmsINT propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the integer property identified by name.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:

```
xmsRC xmsDestSetProperty(xmsHDest destination,
                         xmsHProperty property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

destination (input)
The handle for the destination.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set String Property

C interface:

```
xmsRC xmsDestSetStringProperty(xmsHDest destination,
                               xmsCHAR *propertyName,
                               xmsCHAR *propertyValue,
                               xmsSIZE length,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the string property identified by name.

Parameters:

destination (input)
The handle for the destination.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property as a character array.

length (input)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

ErrorBlock

If a C function call fails, XMS can store information in an error block about why the call failed. For more information about the error block and its contents, see "The error block" on page 35.

This class is a helper class, and only the C interface uses the class.

Methods

Clear Error Block

C interface:

```
xmsRC xmsErrorClear(xmsHErrorBlock errorBlock);
```

Clear the contents of the error block.

Note that XMS automatically clears the contents of an error block that is passed by a C API function call.

Parameters:

errorBlock (input)
The handle for the error block.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Error Block

C interface:

```
xmsRC xmsErrorCreate(xmsHErrorBlock *errorBlock);
```

Create an error block.

Parameters:

errorBlock (output)
The handle for the error block.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Error Block

C interface:

```
xmsRC xmsErrorDispose(xmsHErrorBlock *errorBlock);
```

Delete the error block and any linked error blocks.

If an application tries to delete an error block that is already deleted, the call is ignored.

Parameters:

errorBlock (input/output)
On input, the handle for the error block. On output the call returns a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error Code

C interface:
```
xmsRC xmsErrorGetErrorCode(xmsHErrorBlock errorBlock,
                           xmsINT *errorCode);
```

Get the error code.

Parameters:

errorBlock (input)
The handle for the error block.

errorCode (output)
The error code.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error Data

C interface:
```
xmsRC xmsErrorGetErrorData(xmsHErrorBlock errorBlock,
                           xmsCHAR *buffer,
                           xmsSIZE bufferLength,
                           xmsSIZE *actualLength);
```

Get the error data.

For more information about how to use this method, see "C functions that return a string or byte array by value" on page 33.

Parameters:

errorBlock (input)
The handle for the error block.

buffer (output)
The buffer to contain the error data.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, the error data is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the error data in bytes.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error Module

C interface:

```
xmsRC xmsErrorGetModule(xmsHErrorBlock errorBlock,
                        xmsMODULE_TYPE *errorModule);
```

Get the identifier of the XMS module where the error originated. This information might be useful to your IBM Support Center if an unexpected error occurs.

Parameters:

errorBlock (input)
The handle for the error block.

errorModule (output)
The identifier of the XMS module.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error String

C interface:

```
xmsRC xmsErrorGetErrorString(xmsHErrorBlock errorBlock,
                             xmsCHAR *buffer,
                             xmsSIZE bufferLength,
                             xmsSIZE *actualLength);
```

Get the error string.

For more information about how to use this method, see “C functions that return a string or byte array by value” on page 33.

Parameters:

errorBlock (input)
The handle for the error block.

buffer (output)
The buffer to contain the error string.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, the error string is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the error string in bytes.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Exception Code

C interface:

```
xmsRC xmsErrorGetJMSException(xmsHErrorBlock errorBlock,
                              xmsJMSEXP_TYPE *exceptionCode);
```

Get the exception code.

Parameters:

errorBlock (input)
The handle for the error block.

exceptionCode (output)
The exception code.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Linked Error

C interface:

```
xmsRC xmsErrorGetLinkedError(xmsHErrorBlock errorBlock,
                             xmsHErrorBlock *linkedError);
```

Get the handle for the next error block in the chain of error blocks.

Parameters:

errorBlock (input)
The handle for the error block.

linkedError (output)
The handle for the next error block in the chain. The call returns a null handle if there are no more error blocks in the chain.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Exception

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
```

If XMS detects an error while processing a call to a C++ method, XMS throws an exception. An exception is an object that encapsulates information about the error.

There are different types of XMS exception, and an Exception object is just one type of exception. However, the Exception class is a superclass of the other XMS exception classes. XMS throws an Exception object in situations where none of the other types of exception are appropriate.

Only the C++ interface uses this class.

Methods

Check Whether Null

C++ interface:

```
xmsBOOL isNull() const;
```

Determine whether the Exception object is a null object.

Parameters:
None

C++ method returns:

- `xmsTRUE`, if the Exception object is a null object.
- `xmsFALSE`, if the Exception object is not a null object.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Exception

C++ interface:

```
virtual ~Exception() throw();
```

Delete the exception and any linked exceptions.

Parameters:
None

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Dump Exception

C++ interface:

```
xmsVOID dump(std::ostream outputStream) const;
```

Dump the exception to the specified C++ output stream as formatted text.

Parameters:

outputStream (input)
The C++ output stream.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error Code

C++ interface:

```
xmsINT getErrorCode() const;
```

Get the error code.

Parameters:
None

C++ method returns:
The error code.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error Data

C++ interface:

```
String getErrorData() const;
```

Get the free format data that provides additional information about the error.

Parameters:
None

C++ method returns:
A String object encapsulating the error data.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Error String

C++ interface:

```
String getErrorString() const;
```

Get the string of characters that describes the error. The characters in the string are the same as those in the named constant that represents the error code.

Parameters:
None

C++ method returns:
A String object encapsulating the error string.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Exception Code

C++ interface:

```
xmsJMSEXP_TYPE getJMSException() const;
```

Get the exception code.

Parameters:
None

C++ method returns:
The exception code.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C++ interface:

```
xmsHErrorBlock getHandle() const;
```

Get the handle for the internal error block that XMS creates for the exception.

Parameters:
None

C++ method returns:
The handle for the error block.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Linked Exception

C++ interface:

```
Exception * getLinkedException() const;
```

Get a pointer to the next exception in the chain of exceptions.

Parameters:
None

C++ method returns:
A pointer to an exception. The call returns a null pointer if there are no more exceptions in the chain.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

ExceptionListener

C++ inheritance hierarchy:
None

An application uses an exception listener to be notified asynchronously of a problem with a connection.

If an application uses a connection only to consume messages asynchronously, and for no other purpose, then the only way the application can learn about a problem with the connection is by using an exception listener.

Methods

On Exception

C interface:

```
xmsVOID onException(xmsCONTEXT context,
                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID onException(const Exception *exception);
```

Notify the application of a problem with a connection.

In a C application, onException() is the exception listener function that is registered with the connection. The name of the function does not have to be onException.

In a C++ application, onException() is a method of the exception listener that is registered with the connection. The name of the method must be onException.

For more information about using exception listener functions in a C application, see “Using exception listener functions in C” on page 32. If you are using C++, see “Using exception listeners in C++” on page 43 instead.

Parameters:

context (input)
A pointer to the context data that is registered with the connection.

errorBlock (input)
The handle for an error block created by XMS.

exception (input)
A pointer to an exception created by XMS.

C++ method returns:
Void

IllegalStateException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::IllegalStateException
```

XMS throws this exception if an application calls a method at an incorrect or inappropriate time, or if XMS is not in an appropriate state for the requested operation.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

InvalidDestinationException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::InvalidDestinationException
```

XMS throws this exception if an application specifies a destination that is not valid.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

InvalidSelectorException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::InvalidSelectorException
```

XMS throws this exception if an application provides a message selector expression whose syntax is not valid.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

Iterator

C++ inheritance hierarchy:
None

An iterator encapsulates a list of Property objects and a cursor that maintains the current position in the list. When an iterator is created, the position of the cursor is before the first Property object.

An application uses an iterator to retrieve each Property object in turn.

This class is a helper class.

Methods

Check for More Properties

C interface:

```
xmsRC xmsIteratorHasNext(xmsHIterator iterator,
                         xmsBOOL *moreProperties,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBOOL hasNext();
```

Check whether there are any more Property objects beyond the current position of the cursor. The call does not move the cursor.

Parameters:

iterator (input)
The handle for the iterator.

moreProperties (output)
If the value is `xmsTRUE`, more Property objects are available for retrieval beyond the current position of the cursor. If the value is `xmsFALSE`, no more Property objects are available for retrieval beyond the current position of the cursor.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the moreProperties parameter.

Thread context:
Session

Exceptions:
XMS_X_GENERAL_EXCEPTION

Check Whether Null

C interface:
Not applicable

C++ interface:

```
xmsBOOL isNull() const;
```

Determine whether the Iterator object is a null object.

Parameters:
None

C++ method returns:
- xmsTRUE, if the Iterator object is a null object.
- xmsFALSE, if the Iterator object is not a null object.

Thread context:
Session

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Iterator

C interface:
```
xmsRC xmsIteratorDispose(xmsHIterator *iterator,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual ~Iterator();
```

Delete the iterator.

If an application tries to delete an iterator that is already deleted, the call is ignored.

Parameters:

iterator (input/output)
On input, the handle for the iterator. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

Thread context:
Session

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHIterator getHandle() const;
```

Get the handle that a C application would use to access the iterator.

Parameters:
None

C++ method returns:
The handle for the iterator.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Next Property

C interface:

```
xmsRC xmsIteratorGetNextProperty(xmsHIterator iterator,
                                 xmsHProperty *property,
                                 xmsHErrorBlock errorBlock);
```

C++ interface:

```
Property getNextProperty();
```

Move the cursor to the next Property object and get the Property object at the new position of the cursor.

Parameters:

iterator (input)
The handle for the iterator.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Thread context:
Session

Exceptions:
XMS_X_GENERAL_EXCEPTION

Reset Iterator

C interface:

```
xmsRC xmsIteratorReset(xmsHIterator iterator,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID reset();
```

Move the cursor back to a position before the first Property object.

Parameters:

iterator (input)
The handle for the iterator.

errorBlock (input)
The handle for an error block or a null handle.

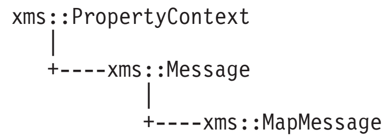
C++ method returns:
Void

Thread context:
Session

Exceptions:
XMS_X_GENERAL_EXCEPTION

MapMessage

C++ inheritance hierarchy:


```
xms::PropertyContext
   |
   +----xms::Message
           |
           +----xms::MapMessage
```

A map message is a message whose body comprises a set of name-value pairs.

Methods

Check Name-Value Pair Exists

C interface:
```
xmsRC xmsMapMsgItemExists(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL *pairExists,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBOOL itemExists(const String & name) const;
```

Check whether the body of the map message contains a name-value pair with the specified name.

Parameters:

message (input)
The handle for the message.

name (input)
The name of the name-value pair.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

pairExists (output)
If the value is xmsTRUE, the body of the map message contains a name-value pair with the specified name. If the value is xmsFALSE, the body of the map message does not contain a name-value pair with the specified name.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the pairExists parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Boolean Value

C interface:
```
xmsRC xmsMapMsgGetBoolean(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL *value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBOOL getBoolean(const String & name) const;
```

Get the boolean value identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the boolean value.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The boolean value retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The boolean value retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte

C interface:
```
xmsRC xmsMapMsgGetByte(xmsHMsg message,
                       xmsCHAR *name,
                       xmsBYTE *value,
                       xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBYTE getByte(const String & name) const;
```

Get the byte identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the byte.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The byte retrieved from the body of the map message. No data conversion is performed on the byte.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The byte retrieved from the body of the map message. No data conversion is performed on the byte.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Bytes

C interface:

```
xmsRC xmsMapMsgGetBytes(xmsHMsg message,
                        xmsCHAR *name,
                        xmsBYTE *buffer,
                        xmsSIZE bufferLength,
                        xmsSIZE *actualLength,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT getBytes(const String & name,
                xmsBYTE *buffer,
                const xmsSIZE bufferLength,
                xmsSIZE *actualLength) const;
```

Get the array of bytes identified by name from the body of the map message.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33. If you are using the C++ interface, see “C++ methods that return a byte array” on page 38 instead.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the array of bytes.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

buffer (output)
The buffer to contain the array of bytes. No data conversion is performed on the bytes that are returned.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, the array of bytes is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The number of bytes in the array.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The number of bytes in the array.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Bytes by Reference

C interface:

```
xmsRC xmsMapMsgGetBytesByRef(xmsHMsg message,
                             xmsCHAR *name,
                             xmsBYTE **array,
                             xmsSIZE *length,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

Not applicable

Get a pointer to an array of bytes in the body of the map message and get the length of the array. The array of bytes is identified by name.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the array of bytes. The name is in the format of a null terminated string.

array (output)
A pointer to the array of bytes.

length (output)
The number of bytes in the array.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

XMS_X_GENERAL_EXCEPTION

Get Character

C interface:

```
xmsRC xmsMapMsgGetChar(xmsHMsg message,
                       xmsCHAR *name,
                       xmsCHAR16 *value,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsCHAR16 getChar(const String & name) const;
```

Get the character identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the character.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The character retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The character retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Double Precision Floating Point Number

C interface:

```
xmsRC xmsMapMsgGetDouble(xmsHMsg message,
                         xmsCHAR *name,
                         xmsDOUBLE *value,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsDOUBLE getDouble(const String & name) const;
```

Get the double precision floating point number identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the double precision floating point number.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The double precision floating point number retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The double precision floating point number retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Floating Point Number

C interface:

```
xmsRC xmsMapMsgGetFloat(xmsHMsg message,
                        xmsCHAR *name,
                        xmsFLOAT *value,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsFLOAT getFloat(const String & name) const;
```

Get the floating point number identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the floating point number.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The floating point number retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The floating point number retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer

C interface:

```
xmsRC xmsMapMsgGetInt(xmsHMsg message,
                      xmsCHAR *name,
                      xmsINT *value,
                      xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT getInt(const String & name) const;
```

Get the integer identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the integer.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The integer retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The integer retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Long Integer

C interface:
```
xmsRC xmsMapMsgGetLong(xmsHMsg message,
                       xmsCHAR *name,
                       xmsLONG *value,
                       xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsLONG getLong(const String & name) const;
```

Get the long integer identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the long integer.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The long integer retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The long integer retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Name-Value Pairs

C interface:
```
xmsRC xmsMapMsgGetMapNames(xmsHMsg message,
                           xmsHIterator *iterator,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
```
Iterator getMapNames() const;
```

Get a list of the name-value pairs in the body of the map message.

The call returns an iterator, which the application can then use to access each of the name-value pairs in turn. The iterator encapsulates a list of Property objects, where each Property object encapsulates a name-value pair.

Note: The equivalent JMS method performs a slightly different function. The JMS method returns an enumeration of only the names, not the values, in the body of the map message.

Parameters:

message (input)
The handle for the message.

iterator (output)
The handle for the iterator.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Iterator object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Object

C interface:

```
xmsRC xmsMapMsgGetObject(xmsHMsg message,
                         xmsCHAR *name,
                         xmsBYTE *buffer,
                         xmsSIZE bufferLength,
                         xmsSIZE *actualLength,
                         xmsOBJECT_TYPE *objectType,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsOBJECT_TYPE getObject(const String & name,
                         xmsBYTE *buffer,
                         const xmsSIZE bufferLength,
                         xmsSIZE *actualLength) const;
```

Get the value of a name-value pair, and its data type, from the body of the map message. The name-value pair is identified by name.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33. If you are using the C++ interface, see "C++ methods that return a byte array" on page 38 instead.

Parameters:

message (input)
The handle for the message.

name (input)
The name of the name-value pair.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

buffer (output)
The buffer to contain the value, which is returned as an array of bytes. If data conversion is required, this is the value after conversion.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, the object is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value in bytes. If data conversion is required, this is the length after conversion.

objectType (output)
The data type of the value, which is one of the following object types:

```
XMS_OBJECT_TYPE_BOOL
XMS_OBJECT_TYPE_BYTE
XMS_OBJECT_TYPE_CHAR
XMS_OBJECT_TYPE_DOUBLE
XMS_OBJECT_TYPE_FLOAT
XMS_OBJECT_TYPE_INT
XMS_OBJECT_TYPE_LONG
XMS_OBJECT_TYPE_SHORT
XMS_OBJECT_TYPE_STRING
```

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the objectType parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Short Integer

C interface:

```
xmsRC xmsMapMsgGetShort(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSHORT *value,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsSHORT getShort(const String & name) const;
```

Get the short integer identified by name from the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the short integer.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (output)
The short integer retrieved from the body of the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The short integer retrieved from the body of the map message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String

C interface:
```
xmsRC xmsMapMsgGetString(xmsHMsg message,
                         xmsCHAR *name,
                         xmsCHAR *buffer,
                         xmsSIZE bufferLength,
                         xmsSIZE *actualLength,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
String getString(const String & name) const;
```

Get the string identified by name from the body of the map message.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the string.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

buffer (output)
The buffer to contain the string. If data conversion is required, this is the string after conversion.

bufferLength (input)
The length of the buffer in bytes. If you specify a length of zero, the string is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the string in bytes. If data conversion is required, this is the length of the string after conversion.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the string retrieved from the body of the map message. If data conversion is required, this is the string after conversion.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String by Reference

C interface:

```
xmsRC xmsMapMsgGetStringByRef(xmsHMsg message,
                              xmsCHAR *name,
                              xmsCHAR **string,
                              xmsSIZE *length,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

Not applicable

Get a pointer to the string identified by name and get the length of the string.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

message (input)
The handle for the message.

name (input)
The name that identifies the string. The name is in the format of a null terminated string.

string (output)
A pointer to the string. If data conversion is required, this is the string after conversion.

length (output)
The length of the string in bytes. If data conversion is required, this is the length after conversion.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

XMS_X_GENERAL_EXCEPTION

Set Boolean Value

C interface:

```
xmsRC xmsMapMsgSetBoolean(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL value,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setBoolean(const String & name,
                   const xmsBOOL value);
```

Set a boolean value in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the boolean value in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The boolean value to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Byte

C interface:

```
xmsRC xmsMapMsgSetByte(xmsHMsg message,
                       xmsCHAR *name,
                       xmsBYTE value,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setByte(const String & name,
                const xmsBYTE value);
```

Set a byte in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the byte in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The byte to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Bytes

C interface:

```
xmsRC xmsMapMsgSetBytes(xmsHMsg message,
                        xmsCHAR *name,
                        xmsBYTE *value,
                        xmsSIZE length,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setBytes(const String & name,
                 const xmsBYTE *value,
                 const xmsSIZE length);
```

Set an array of bytes in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the array of bytes in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The array of bytes to be set.

length (input)
The number of bytes in the array.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Character

C interface:

```
xmsRC xmsMapMsgSetChar(xmsHMsg message,
                       xmsCHAR *name,
                       xmsCHAR16 value,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setChar(const String & name,
                const xmsCHAR16 value);
```

Set a 2-byte character in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the character in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The character to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Double Precision Floating Point Number

C interface:

```
xmsRC xmsMapMsgSetDouble(xmsHMsg message,
                         xmsCHAR *name,
                         xmsDOUBLE value,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setDouble(const String & name,
                  const xmsDOUBLE value);
```

Set a double precision floating point number in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the double precision floating point number in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The double precision floating point number to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Floating Point Number

C interface:

```
xmsRC xmsMapMsgSetFloat (xmsHMsg message,
                         xmsCHAR *name,
                         xmsFLOAT value,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setFloat(const String & name,
                 const xmsFLOAT value);
```

Set a floating point number in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the floating point number in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The floating point number to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Integer

C interface:

```
xmsRC xmsMapMsgSetInt(xmsHMsg message,
                      xmsCHAR *name,
                      xmsINT value,
                      xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setInt(const String & name,
               const xmsINT value);
```

Set an integer in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the integer in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The integer to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Long Integer

C interface:

```
xmsRC xmsMapMsgSetLong (xmsHMsg message,
                        xmsCHAR *name,
                        xmsLONG value,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setLong(const String & name,
                const xmsLONG value);
```

Set a long integer in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the long integer in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The long integer to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Object

C interface:

```
xmsRC xmsMapMsgSetObject(xmsHMsg message,
                         xmsCHAR *name,
                         xmsBYTE *value,
                         xmsSIZE length,
                         xmsOBJECT_TYPE objectType,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setObject(const String & name,
                  const xmsOBJECT_TYPE objectType,
                  const xmsBYTE *value,
                  const xmsSIZE length);
```

Set a value, with a specified data type, in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the value in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The array of bytes representing the value to be set.

length (input)
The number of bytes in the array.

objectType (input)
The data type of the value, which must be one of the following objecttypes:

```
XMS_OBJECT_TYPE_BOOL
XMS_OBJECT_TYPE_BYTE
XMS_OBJECT_TYPE_CHAR
XMS_OBJECT_TYPE_DOUBLE
XMS_OBJECT_TYPE_FLOAT
XMS_OBJECT_TYPE_INT
XMS_OBJECT_TYPE_LONG
XMS_OBJECT_TYPE_SHORT
XMS_OBJECT_TYPE_STRING
```

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Short Integer

C interface:

```
xmsRC xmsMapMsgSetShort(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSHORT value,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setShort(const String & name,
                 const xmsSHORT value);
```

Set a short integer in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the short integer in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The short integer to be set.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set String

C interface:
```
xmsRC xmsMapMsgSetString(xmsHMsg message,
                         xmsCHAR *name,
                         xmsCHAR *value,
                         xmsSIZE length,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setString(const String & name,
                  const String value);
```

Set a string in the body of the map message.

Parameters:

message (input)
The handle for the message.

name (input)
The name to identify the string in the body of the map message.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

value (input)
The string to be set.

In the C interface, this parameter is a character array.

In the C++ interface, this parameter is a String object.

length (input)
The length of the string in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the Message class:

clearBody, clearProperties, getHandle, getJMSCorrelationID, getJMSDeliveryMode, getJMSDestination, getJMSExpiration, getJMSMessageID, getJMSPriority, getJMSRedelivered, getJMSReplyTo, getJMSTimestamp,

getJMSType, getObjectProperty, getProperties, getProperty, isNull, propertyExists, setJMSCorrelationID, setJMSDeliveryMode, setJMSDestination, setJMSExpiration, setJMSMessageID, setJMSPriority, setJMSRedelivered, setJMSReplyTo, setJMSTimestamp, setJMSType, setObjectProperty, setProperty

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

Message

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::Message
```

A Message object represents a message that an application sends or receives.

For a list of the JMS message header fields in a Message object, see "Header fields in an XMS message" on page 47. For a list of the JMS defined properties of a Message object, see "JMS defined properties of a message" on page 49. For a list of the IBM defined properties of a Message object, see "IBM defined properties of a message" on page 49.

Methods

Check Property Exists

C interface:

```
xmsRC xmsMsgPropertyExists(xmsHMsg message,
                           xmsCHAR *propertyName,
                           xmsBOOL *propertyExists,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBOOL propertyExists(const String & propertyName) const;
```

Check whether the message has a property with the specified name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

propertyExists (output)
If the value is `xmsTRUE`, the message has a property with the specified name. If the value is `xmsFALSE`, the message does not have a property with the specified name.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the propertyExists parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the Message object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the Message object is a null object.
- `xmsFALSE`, if the Message object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Clear Body

C interface:
```
xmsRC xmsMsgClearBody(xmsHMsg message,
                      xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID clearBody();
```

Clear the body of the message. The header fields and message properties are not cleared.

If you clear a message body that is read-only, the body is left in the same state as an empty body in a newly created message.

Parameters:

message (input)
The handle for the message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Clear Properties

C interface:
```
xmsRC xmsMsgClearProperties(xmsHMsg message,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID clearProperties();
```

Clear the properties of the message. The header fields and the message body are not cleared.

If you clear message properties that are read-only, the properties become readable and writable.

Parameters:

message (input)
The handle for the message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Message

C interface:

```
xmsRC xmsMsgDispose(xmsHMsg *message,
                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual ~Message();
```

Delete the message.

If an application tries to delete a message that is already deleted, the call is ignored.

Parameters:

message (input)
On input, the handle for the message. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Boolean Property

C interface:

```
xmsRC xmsMsgGetBooleanProperty(xmsHMsg message,
                               xmsCHAR *propertyName,
                               xmsBOOL *propertyValue,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the boolean property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Property

C interface:
```
xmsRC xmsMsgGetByteProperty(xmsHMsg message,
                            xmsCHAR *propertyName,
                            xmsBYTE *propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the byte property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Double Precision Floating Point Property

C interface:
```
xmsRC xmsMsgGetDoubleProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsDOUBLE *propertyValue,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the double precision floating point property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Floating Point Property

C interface:

```
xmsRC xmsMsgGetFloatProperty(xmsHMsg message,
                             xmsCHAR *propertyName,
                             xmsFLOAT *propertyValue,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the floating point property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:

```
xmsHMsg getHandle() const;
```

Get the handle that a C application would use to access the message.

Parameters:
None

C++ method returns:
The handle for the message.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property

C interface:

```
xmsRC xmsMsgGetFloatProperty(xmsHMsg message,
                             xmsCHAR *propertyName,
                             xmsFLOAT *propertyValue,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

Inherited from the PropertyContext class

Get the value of the integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

XMS_X_GENERAL_EXCEPTION

Get JMSCorrelationID

C interface:

```
xmsRC xmsMsgGetJMSCorrelationID(xmsHMsg message,
                                xmsCHAR *correlID,
                                xmsSIZE length,
                                xmsSIZE *actualLength,
                                xmsHErrorBlock errorBlock);
```

C++ interface:

```
String getJMSCorrelationID() const;
```

Get the correlation identifier of the message in the format of a string.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

message (input)
The handle for the message.

correlID (output)
The buffer to contain the correlation identifier.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the correlation identifier is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the correlation identifier in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the correlation identifier.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSDeliveryMode

C interface:
```
xmsRC xmsMsgGetJMSDeliveryMode(xmsHMsg message,
                               xmsDELIVERY_MODE *deliveryMode,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsDELIVERY_MODE getJMSDeliveryMode() const;
```

Get the delivery mode of the message. The delivery mode is set by the Send call when the message is sent.

Parameters:

message (input)
The handle for the message.

deliveryMode (output)
The delivery mode of the message, which is one of the following values:

XMSC_NON_PERSISTENT

XMSC_PERSISTENT

For a newly created message that has not been sent, the delivery mode is XMSC_NON_PERSISTENT. For a message that has been received, the call returns the delivery mode that was set by the Send call when the message was sent unless the receiving application changes the delivery mode by calling Set JMSDeliveryMode.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the deliveryMode parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSDestination

C interface:
```
xmsRC xmsMsgGetJMSDestination(xmsHMsg message,
                              xmsHDest *destination,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
```
Destination getJMSDestination() const;
```

Get the destination of the message. The destination is set by the Send call when the message is sent.

Parameters:

message (input)
The handle for the message.

destination (output)
The handle for the destination of the message.

For a message that has been received, the call returns a handle for the destination that was set by the Send call when the message was sent unless the receiving application changes the destination by calling Set JMSDestination.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Destination object.

For a message that has been received, the call returns a Destination object for the destination that was set by the Send call when the message was sent unless the receiving application changes the destination by calling Set JMSDestination.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSExpiration

C interface:

```
xmsRC xmsMsgGetJMSExpiration(xmsHMsg message,
                             xmsLONG *expiration,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsLONG getJMSExpiration() const;
```

Get the expiration time of the message.

The expiration time is set by the Send call when the message is sent. Its value is calculated by adding the time to live, as specified by the sending application, to the time when the message is sent. The expiration time is expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

If the time to live is zero, the Send call sets the expiration time to zero to indicate that the message does not expire.

Parameters:

message (input)
The handle for the message.

expiration (output)
The expiration time of the message.

For a newly created message that has not been sent, the expiration time is zero unless the sending application sets a different expiration time by calling Set JMSExpiration. For a message that has been received, the call returns the expiration time that was set

by the Send call when the message was sent unless the receiving application changes the expiration time by calling Set JMSExpiration.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the expiration parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSMessageID

C interface:

```
xmsRC xmsMsgGetJMSMessageID(xmsHMsg message,
                            xmsCHAR *msgID,
                            xmsSIZE length,
                            xmsSIZE *actualLength,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
String getJMSMessageID() const;
```

Get the message identifier of the message. The message identifier is set by the Send call when the message is sent.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

message (input)
The handle for the message.

msgID (output)
The buffer to contain the message identifier.

For a message that has been received, the call returns the message identifier that was set by the Send call when the message was sent unless the receiving application changes the message identifier by calling Set JMSMessageID.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the message identifier is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the message identifier in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the message identifier.

See also the description of the msgID parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Notes:

1. If a message has no message identifier, a call from a C application leaves the contents of the buffer unchanged, sets the actualLength parameter to zero, and returns error code XMS_E_NOT_SET. A call from a C++ application in the same circumstances throws an exception with error code XMS_E_NOT_SET.

Get JMSPriority

C interface:

```
xmsRC xmsMsgGetJMSPriority(xmsHMsg message,
                           xmsINT *priority,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT getJMSPriority() const;
```

Get the priority of the message. The priority is set by the Send call when the message is sent.

Parameters:

message (input)
The handle for the message.

priority (output)
The priority of the message. The value is an integer in the range 0, the lowest priority, to 9, the highest priority.

For a newly created message that has not been sent, the priority is 4 unless the sending application sets a different priority by calling Set JMSPriority. For a message that has been received, the call returns the priority that was set by the Send call when the message was sent unless the receiving application changes the priority by calling Set JMSPriority.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the priority parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSRedelivered

C interface:

```
xmsRC xmsMsgGetJMSRedelivered(xmsHMsg message,
                              xmsBOOL *redelivered,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBOOL getJMSRedelivered() const;
```

Get an indication of whether the message is being re-delivered. The indication is set by the Receive call when the message is received.

Parameters:

message (input)
The handle for the message.

redelivered (output)
If the connection uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport, the value is always `xmsFALSE`, which means that the message is not being re-delivered.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the redelivered parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSReplyTo

C interface:

```
xmsRC xmsMsgGetJMSReplyTo(xmsHMsg message,
                          xmsHDest *destination,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
Destination getJMSReplyTo() const;
```

Get the destination where a reply to the message is to be sent.

Parameters:

message (input)
The handle for the message.

destination (output)
The handle for the destination where a reply to the message is to be sent.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A Destination object for the destination where a reply to the message is to be sent.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get JMSTimestamp

C interface:

```
xmsRC xmsMsgGetJMSTimestamp(xmsHMsg message,
                            xmsLONG *timestamp,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsLONG getJMSTimestamp() const;
```

Get the time when the message was sent. The timestamp is set by the Send call when the message is sent and is expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

Parameters:

message (input)
: The handle for the message.

timestamp (output)
: The time when the message was sent.

  For a newly created message that has not been sent, the timestamp is zero unless the sending application sets a different timestamp by calling Set JMSTimestamp. For a message that has been received, the call returns the timestamp that was set by the Send call when the message was sent unless the receiving application changes the timestamp by calling Set JMSTimestamp.

errorBlock (input)
: The handle for an error block or a null handle.

C++ method returns:
See the description of the timestamp parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Notes:

1. If the timestamp is undefined, a call from a C application sets the timestamp parameter to zero, but returns no error. A call from a C++ application in the same circumstances returns zero, but throws no exception.

Get JMSType

C interface:

```
xmsRC xmsMsgGetJMSType(xmsHMsg message,
                       xmsCHAR *type,
                       xmsSIZE length,
                       xmsSIZE *actualLength,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
String getJMSType() const;
```

Get the type of the message.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

message (input)
: The handle for the message.

type (output)
: The buffer to contain the type of the message. If data conversion is required, this is the type after conversion.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the type of the message is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the type of the message in bytes. If data conversion is required, this is the length after conversion.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String encapsulating the type of the message. If data conversion is required, this is the type after conversion.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Long Integer Property

C interface:

```
xmsRC xmsMsgGetLongProperty(xmsHMsg message,
                            xmsCHAR *propertyName,
                            xmsLONG *propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the long integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Object Property

C interface:

```
xmsRC xmsMsgGetObjectProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsBYTE *propertyValue,
                              xmsSIZE length,
                              xmsSIZE *actualLength,
                              xmsOBJECT_TYPE *objectType,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsOBJECT_TYPE getObjectProperty(const String & propertyName,
                                 xmsBYTE *propertyValue,
                                 const xmsSIZE length,
                                 xmsSIZE *actualLength) const;
```

Get the value and data type of a property identified by name.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33. If you are using the C++ interface, see "C++ methods that return a byte array" on page 38 instead.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

propertyValue (output)
The buffer to contain the value of the property, which is returned as an array of bytes. If data conversion is required, this is the value after conversion.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the value of the property is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value of the property in bytes. If data conversion is required, this is the length after conversion.

objectType (output)
The data type of the value of the property, which is one of the following object types:

XMS_OBJECT_TYPE_BOOL

XMS_OBJECT_TYPE_BYTE

XMS_OBJECT_TYPE_CHAR

XMS_OBJECT_TYPE_DOUBLE

XMS_OBJECT_TYPE_FLOAT

XMS_OBJECT_TYPE_INT

XMS_OBJECT_TYPE_LONG

XMS_OBJECT_TYPE_SHORT

XMS_OBJECT_TYPE_STRING

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the objectType parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Properties

C interface:

```
xmsRC xmsMsgGetProperties(xmsHMsg message,
                          xmsHIterator *iterator,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
Iterator getProperties() const;
```

Get a list of the properties of the message.

The call returns an iterator, which the application can then use to access each of the properties in turn.

Note: The equivalent JMS method performs a slightly different function. The JMS method returns an enumeration of only the names of the properties of the message, not their values.

Parameters:

message (input)
The handle for the message.

iterator (input)
The handle for the iterator.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Iterator object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:

```
xmsRC xmsMsgGetProperty(xmsHMsg message,
                        xmsCHAR *propertyName,
                        xmsHProperty *property,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual Property getProperty(String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Short Integer Property

C interface:

```
xmsRC xmsMsgGetShortProperty(xmsHMsg message,
                             xmsCHAR *propertyName,
                             xmsSHORT *propertyValue,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the short integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property

C interface:

```
xmsRC xmsMsgGetStringProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsCHAR *propertyValue,
                              xmsSIZE length,
                              xmsSIZE *actualLength,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Get the value of the string property identified by name.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
The buffer to contain the value of the property. If data conversion is required, this is the value after conversion.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the value of the property is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value of the property in bytes. If data conversion is required, this is the length after conversion.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property by Reference

C interface:

```
xmsRC xmsMsgGetStringPropertyByRef(xmsHMsg message,
                                   xmsCHAR *propertyName,
                                   xmsCHAR **propertyValue,
                                   xmsSIZE *length,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the value of the string property identified by name.

For more information about how to use this function, see “C functions that return a string or byte array by reference” on page 34.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (output)
A pointer to the value of the property. If data conversion is required, this is the value after conversion.

length (output)
The length of the value of the property in bytes. If data conversion is required, this is the length of the value after conversion.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Type

C interface:

```
xmsRC xmsMsgGetTypeId(xmsHMsg message,
                      xmsMESSAGE_TYPE *type,
                      xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get the body type of the message.

For information about message body types, see "The body of an XMS message" on page 50.

Parameters:

message (input)
The handle for the message.

type (output)
The body type of the message, which is one of the following values:

XMSC_T_BYTES_MSG

XMSC_T_MAP_MSG

XMSC_T_MSG

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Boolean Property

C interface:

```
xmsRC xmsMsgSetBooleanProperty(xmsHMsg message,
                               xmsCHAR *propertyName,
                               xmsBOOL propertyValue,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the boolean property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Byte Property

C interface:

```
xmsRC xmsMsgSetByteProperty(xmsHMsg message,
                            xmsCHAR *propertyName,
                            xmsBYTE propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the byte property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Double Precision Floating Point Property

C interface:

```
xmsRC xmsMsgSetDoubleProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsDOUBLE propertyValue,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the double precision floating point property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Floating Point Property

C interface:

```
xmsRC xmsMsgSetFloatProperty(xmsHMsg message,
                             xmsCHAR *propertyName,
                             xmsFLOAT propertyValue,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the floating point property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Integer Property

C interface:

```
xmsRC xmsMsgSetIntProperty(xmsHMsg message,
                           xmsCHAR *propertyName,
                           xmsINT propertyValue,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set JMSCorrelationID

C interface:
```
xmsRC xmsMsgSetJMSCorrelationID(xmsHMsg message,
                                xmsCHAR *correlID,
                                xmsSIZE length,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setJMSCorrelationID(const String correlID);
```

Set the correlation identifier of the message.

Parameters:

message (input)
The handle for the message.

correlID (input)
The correlation identifier.

In the C interface, this parameter is a character array.

In the C++ interface, this parameter is a String object.

length (input)
The length of the correlation identifier in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSDeliveryMode

C interface:
```
xmsRC xmsMsgSetJMSDeliveryMode(xmsHMsg message,
                               xmsDELIVERY_MODE deliveryMode,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setJMSDeliveryMode(const xmsDELIVERY_MODE deliveryMode);
```

Set the delivery mode of the message.

A delivery mode set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the delivery mode of a message that has been received.

Parameters:

message (input)
The handle for the message.

deliveryMode (input)
The delivery mode of the message. The value must be `XMSC_NON_PERSISTENT`.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSDestination

C interface:

```
xmsRC xmsMsgSetJMSDestination(xmsHMsg message,
                              xmsHDest destination,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSDestination(const Destination & destination);
```

Set the destination of the message.

A destination set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the destination of a message that has been received.

Parameters:

message (input)
The handle for the message.

destination (input)
The destination of the message.

In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSExpiration

C interface:

```
xmsRC xmsMsgSetJMSExpiration(xmsHMsg message,
                             xmsLONG expiration,
                             xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSExpiration(const xmsLONG expiration);
```

Set the expiration time of the message.

An expiration time set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the expiration time of a message that has been received.

Parameters:

message (input)
The handle for the message.

expiration (input)
The expiration time of the message expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSMessageID

C interface:

```
xmsRC xmsMsgSetJMSMessageID(xmsHMsg message,
                            xmsCHAR *msgID,
                            xmsSIZE length,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSMessageID(const String & msgID);
```

Set the message identifier of the message.

A message identifier set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the message identifier of a message that has been received.

Parameters:

message (input)
The handle for the message.

msgID (input)
The message identifier.

In the C interface, this parameter is a character array.

In the C++ interface, this parameter is a String object.

length (input)
The length of the message identifier in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSPriority

C interface:
```
xmsRC xmsMsgSetJMSPriority(xmsHMsg message,
                           xmsINT priority,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setJMSPriority(const xmsINT priority);
```

Set the priority of the message.

A priority set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the priority of a message that has been received.

Parameters:

message (input)
The handle for the message.

priority (input)
The priority of the message. The value can be an integer in the range `0`, the lowest priority, to `9`, the highest priority.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSRedelivered

C interface:
```
xmsRC xmsMsgSetJMSRedelivered(xmsHMsg message,
                              xmsBOOL redelivered,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setJMSRedelivered(const xmsBOOL redelivered);
```

Indicate whether the message is being re-delivered.

An indication of re-delivery set by this method before the message is sent is ignored by the Send call when the message is sent, and is ignored and replaced by

the Receive call when the message is received. However, you can use this method to change the indication for a message that has been received.

Parameters:

message (input)
The handle for the message.

redelivered (input)
The value xmsTRUE means that the message is being re-delivered. The value xmsFALSE means that the message is not being re-delivered.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSReplyTo

C interface:

```
xmsRC xmsMsgSetJMSReplyTo(xmsHMsg message,
                          xmsHDest destination,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSReplyTo(const Destination & destination);
```

Set the destination where a reply to the message is to be sent.

Parameters:

message (input)
The handle for the message.

destination (input)
The destination where a reply to the message is to be sent.

In the C interface, this parameter is the handle for the destination. A null handle means that no reply is expected.

In the C++ interface, this parameter is the Destination object for the destination. A null Destination object means that no reply is expected.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSTimestamp

C interface:

```
xmsRC xmsMsgSetJMSTimestamp(xmsHMsg message,
                            xmsLONG timestamp,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSTimestamp(const xmsLONG timestamp);
```

Set the time when the message is sent.

A timestamp set by this method before the message is sent is ignored and replaced by the Send call when the message is sent. However, you can use this method to change the timestamp of a message that has been received.

Parameters:

message (input)
The handle for the message.

timestamp (input)
The time when the message is sent expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set JMSType

C interface:

```
xmsRC xmsMsgSetJMSType(xmsHMsg message,
                       xmsCHAR *type,
                       xmsSIZE length,
                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setJMSType(const String & type);
```

Set the type of the message.

Parameters:

message (input)
The handle for the message.

type (input)
The type of the message.

In the C interface, this parameter is a character array.

In the C++ interface, this parameter is a String object.

length (input)
The length of the type of the message in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Long Integer Property

C interface:
```
xmsRC xmsMsgSetLongProperty(xmsHMsg message,
                            xmsCHAR *propertyName,
                            xmsLONG propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the long integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Object Property

C interface:
```
xmsRC xmsMsgSetObjectProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsBYTE *propertyValue,
                              xmsSIZE length,
                              xmsOBJECT_TYPE objectType,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setObjectProperty(const String & propertyName,
                          const xmsOBJECT_TYPE objectType,
                          const xmsBYTE *propertyValue,
                          const xmsSIZE length);
```

Set the value and data type of a property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

propertyValue (input)
The value of the property as an array of bytes.

length (input)
The number of bytes in the array.

objectType (input)
The data type of the value of the property, which must be one of the following object types:

XMS_OBJECT_TYPE_BOOL

XMS_OBJECT_TYPE_BYTE

XMS_OBJECT_TYPE_CHAR

XMS_OBJECT_TYPE_DOUBLE

XMS_OBJECT_TYPE_FLOAT

XMS_OBJECT_TYPE_INT

XMS_OBJECT_TYPE_LONG

XMS_OBJECT_TYPE_SHORT

XMS_OBJECT_TYPE_STRING

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:

```
xmsRC xmsMsgSetProperty(xmsHMsg message,
                        xmsHProperty property,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

message (input)
The handle for the message.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Short Integer Property

C interface:
```
xmsRC xmsMsgSetShortProperty(xmsHMsg message,
                              xmsCHAR *propertyName,
                              xmsSHORT propertyValue,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the short integer property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set String Property

C interface:
```
xmsRC xmsMsgSetStringProperty(xmsHMsg message,
                               xmsCHAR *propertyName,
                               xmsCHAR *propertyValue,
                               xmsSIZE length,
                               xmsHErrorBlock errorBlock);
```

C++ interface:
Inherited from the PropertyContext class

Set the value of the string property identified by name.

Parameters:

message (input)
The handle for the message.

propertyName (input)
The name of the property in the format of a null terminated string.

propertyValue (input)
The value of the property as a character array.

length (input)
The length of the value of the property in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

MessageConsumer

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::MessageConsumer
```

An application uses a message consumer to receive messages sent to a destination.

For a list of the XMS defined properties of a MessageConsumer object, see "Properties of MessageConsumer" on page 240.

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the MessageConsumer object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the MessageConsumer object is a null object.
- `xmsFALSE`, if the MessageConsumer object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Close Message Consumer

C interface:
```
xmsRC xmsMsgConsumerClose(xmsHMsgConsumer *consumer,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID close();
```

Close the message consumer.

If an application tries to close a message consumer that is already closed, the call is ignored.

Parameters:

consumer (input/output)
On input, the handle for the message consumer. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHMsgConsumer getHandle() const;
```

Get the handle that a C application would use to access the message consumer.

Parameters:
None

C++ method returns:
The handle for the message consumer.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Message Listener

C interface:
```
xmsRC xmsMsgConsumerGetMessageListener(xmsHMsgConsumer consumer,
                                       fpXMS_MESSAGE_CALLBACK *lsr,
                                       xmsCONTEXT *context,
                                       xmsHErrorBlock errorBlock);
```

C++ interface:
```
MessageListener * getMessageListener() const;
```

In a C application, get pointers to the message listener function and context data that are registered with the message consumer.

In a C++ application, get a pointer to the message listener that is registered with the message consumer.

For more information about using message listener functions in a C application, see "Using message listener functions in C" on page 31. If you are using C++, see "Using message listeners in C++" on page 41 instead.

Parameters:

consumer (input)
The handle for the message consumer.

lsr (output)
A pointer to the message listener function. If no message listener function is registered with the message consumer, the call returns a null pointer.

context (output)
A pointer to the context data. If no message listener function is registered with the connection, the call returns a null pointer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to the message listener. If no message listener is registered with the message consumer, the call returns a null pointer.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Message Selector

C interface:
```
xmsRC xmsMsgConsumerGetMessageSelector(xmsHMsgConsumer consumer,
                                       xmsCHAR *messageSelector,
                                       xmsSIZE length,
                                       xmsSIZE *actualLength,
                                       xmsHErrorBlock errorBlock);
```

C++ interface:
```
String getMessageSelector() const;
```

Get the message selector for the message consumer.

For more information about how to use this method in a C application, see “C functions that return a string or byte array by value” on page 33.

Parameters:

consumer (input)
The handle for the message consumer.

messageSelector (output)
The buffer to contain the message selector expression.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the message selector expression is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the message selector expression in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the message selector expression.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsMsgConsumerGetProperty(xmsHMsgConsumer consumer,
                                xmsCHAR *propertyName,
                                xmsHProperty *property,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

consumer (input)
The handle for the message consumer.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Receive

C interface:

```
xmsRC xmsMsgConsumerReceive(xmsHMsgConsumer consumer,
                            xmsHMsg *message,
                            xmsHErrorBlock errorBlock);
```

C++ interface:

```
Message * receive() const;
```

Receive the next message for the message consumer. The call waits indefinitely for a message, or until the message consumer is closed.

Parameters:

consumer (input)
The handle for the message consumer.

message (output)
The handle for the message. If the message consumer is closed while the call is waiting for a message, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to the Message object. If the message consumer is closed while the call is waiting for a message, the call returns a pointer to a null Message object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Receive (with a wait interval)

C interface:

```
xmsRC xmsMsgConsumerReceiveWithWait(xmsHMsgConsumer consumer,
                                    xmsLONG waitInterval,
                                    xmsHMsg *message,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
Message * receive(const xmsLONG waitInterval) const;
```

Receive the next message for the message consumer. The call waits only a specified period of time for a message, or until the message consumer is closed.

Parameters:

consumer (input)
The handle for the message consumer.

waitInterval (input)
The time, in milliseconds, that the call waits for a message. If you specify a wait interval of zero, the call waits indefinitely for a message.

message (output)
The handle for the message. If no message arrives during the wait interval, or if the message consumer is closed while the call is waiting for a message, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to the Message object. If no message arrives during the wait interval, or if the message consumer is closed while the call is waiting for a message, the call returns a pointer to a null Message object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Receive with No Wait

C interface:

```
xmsRC xmsMsgConsumerReceiveNoWait(xmsHMsgConsumer consumer,
                                  xmsHMsg *message,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
Message * receiveNoWait() const;
```

Receive the next message for the message consumer if one is available immediately.

Parameters:

consumer (input)
The handle for the message consumer.

message (output)
The handle for the message. If no message is available immediately, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to a Message object. If no message is available immediately, the call returns a pointer to a null Message object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Message Listener

C interface:

```
xmsRC xmsMsgConsumerSetMessageListener(xmsHMsgConsumer consumer,
                                       fpXMS_MESSAGE_CALLBACK lsr,
                                       xmsCONTEXT context,
                                       xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setMessageListener(const MessageListener *lsr);
```

In a C application, register a message listener function and context data with the message consumer.

In a C++ application, register a message listener with the message consumer.

For more information about using message listener functions in a C application, see "Using message listener functions in C" on page 31. If you are using C++, see "Using message listeners in C++" on page 41 instead.

Parameters:

consumer (input)
The handle for the message consumer.

lsr (input)
In the C interface, this parameter is a pointer to the message listener function.

In the C++ interface, this parameter is a pointer to the message listener.

If a message listener function or message listener is already registered with the message consumer, you can cancel the registration by specifying a null pointer instead.

context (input)
A pointer to the context data.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:

```
xmsRC xmsMsgConsumerSetProperty(xmsHMsgConsumer consumer,
                                xmsHProperty property,
                                xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

consumer (input)
The handle for the message consumer.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

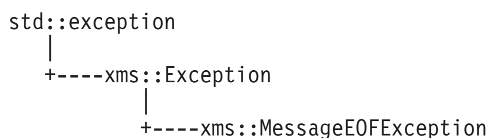
Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

MessageEOFException

C++ inheritance hierarchy:


```
std::exception
   |
   +----xms::Exception
           |
           +----xms::MessageEOFException
```

XMS throws this exception if XMS encounters the end of a bytes message stream when an application is reading the body of a bytes message.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

MessageFormatException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::MessageFormatException
```

XMS throws this exception if XMS encounters a message with a format that is not valid.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

MessageListener

C++ inheritance hierarchy:
None

An application uses a message listener to receive messages asynchronously.

Methods

On Message

C interface:

```
xmsVOID onMessage(xmsCONTEXT context,
                  xmsHMsg message);
```

C++ interface:

```
virtual xmsVOID onMessage(const Message *message);
```

Deliver a message asynchronously to the message consumer.

In a C application, onMessage() is the message listener function that is registered with the message consumer. The name of the function does not have to be onMessage.

In a C++ application, onMessage() is a method of the message listener that is registered with the message consumer. The name of the method must be onMessage.

For more information about using message listener functions in a C application, see "Using message listener functions in C" on page 31. If you are using C++, see "Using message listeners in C++" on page 41 instead.

Parameters:

context (input)
A pointer to the context data that is registered with the message consumer.

message (input)
In the C interface, this parameter is the handle for the message.

In the C++ interface, this parameter is a pointer to the Message object.

C++ method returns:
Void

MessageNotReadableException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::MessageNotReadableException
```

XMS throws this exception if an application attempts to read the body of a message that is write-only.

Only the C++ interface uses this class.

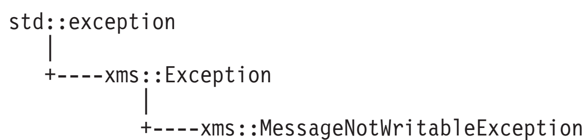
Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

MessageNotWritableException

C++ inheritance hierarchy:



XMS throws this exception if an application attempts to write to the body of a message that is read-only.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

MessageProducer

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::MessageProducer
```

An application uses a message producer to send messages to a destination.

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the MessageProducer object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the MessageProducer object is a null object.
- `xmsFALSE`, if the MessageProducer object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Close Message Producer

C interface:
```
xmsRC xmsMsgProducerClose(xmsHMsgProducer *producer,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID close();
```

Close the message producer.

If an application tries to close a message producer that is already closed, the call is ignored.

Parameters:

producer (input/output)
On input, the handle for the message producer. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Default Delivery Mode

C interface:

```
xmsRC xmsMsgProducerGetDeliveryMode(xmsHMsgProducer producer,
                                    xmsDELIVERY_MODE *deliveryMode,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsDELIVERY_MODE getDeliveryMode() const;
```

Get the default delivery mode for messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

deliveryMode (output)
The default delivery mode. If the connection uses WebSphere MQ Real-Time Transport, the value is always XMSC_NON_PERSISTENT.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the deliveryMode parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Default Priority

C interface:

```
xmsRC xmsMsgProducerGetPriority(xmsHMsgProducer producer,
                                xmsINT *priority,
                                xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsINT getPriority() const;
```

Get the default priority for messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

priority (output)
The default message priority. The value is an integer in the range 0, the lowest priority, to 9, the highest priority.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the priority parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Default Time to Live

C interface:

```
xmsRC xmsMsgProducerGetTimeToLive(xmsHMsgProducer producer,
                                  xmsLONG *timeToLive,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsLONG getTimeToLive() const;
```

Get the default length of time that a message exists before it expires. The time is measured from when the message producer sends the message.

Parameters:

producer (input)
The handle for the message producer.

timeToLive (output)
The default time to live in milliseconds. A value of zero means that a message never expires. If the connection uses WebSphere MQ Real-Time Transport, the value is always zero.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the timeToLive parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Destination

C interface:

```
xmsRC xmsMsgProducerGetDestination(xmsHMsgProducer producer,
                                   xmsHDest *destination,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:

```
Destination getDestination() const;
```

Get the destination for the message producer.

Parameters:

producer (input)
The handle for the message producer.

destination (output)
The handle for the destination. If the message producer does not have a destination, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Destination object. If the message producer does not have a destination, the call returns a null Destination object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Disable Message ID Flag

C interface:

```
xmsRC xmsMsgProducerGetDisableMsgID(xmsHMsgProducer producer,
                                    xmsBOOL *msgIDDisabled,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBOOL getDisabledMessageID() const;
```

Get an indication of whether a receiving application uses message identifiers that are included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

msgIDDisabled (output)
If the value is `xmsTRUE`, a receiving application does not use message identifiers included in messages sent by the message producer. If the value is `xmsFALSE`, a receiving application does use message identifiers.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the msgIDDisabled parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Disable Timestamp Flag

C interface:

```
xmsRC xmsMsgProducerGetDisableMsgTS(xmsHMsgProducer producer,
                                    xmsBOOL *timestampDisabled,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBOOL getDisableMessageTimestamp() const;
```

Get an indication of whether a receiving application uses timestamps that are included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

timestampDisabled (output)
If the value is `xmsTRUE`, a receiving application does not use timestamps included in messages sent by the message producer. If the value is `xmsFALSE`, a receiving application does use timestamps.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the timestampDisabled parameter.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHMsgProducer getHandle() const;
```

Get the handle that a C application would use to access the message producer.

Parameters:
None

C++ method returns:
The handle for the message producer.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsMsgProducerGetProperty(xmsHMsgProducer producer,
                                xmsCHAR *propertyName,
                                xmsHProperty *propertyValue,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

producer (input)
The handle for the message producer.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Send

C interface:

```
xmsRC xmsMsgProducerSend(xmsHMsgProducer producer,
                         xmsHMsg message,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID send(const Message & message) const;
```

Send a message to the destination that was specified when the message producer was created. Send the message using the message producer's default delivery mode, priority, and time to live.

Parameters:

producer (input)
The handle for the message producer.

message (input)
In the C interface, this parameter is the handle for the message.

In the C++ interface, this parameter is the Message object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

Send (specifying a delivery mode, priority, and time to live)

C interface:

```
xmsRC xmsMsgProducerSendWithAttr(xmsHMsgProducer producer,
                                 xmsHMsg message,
                                 xmsDELIVERY_MODE deliveryMode,
                                 xmsINT priority,
                                 xmsLONG timeToLive,
                                 xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID send(const Message & message,
             const xmsDELIVERY_MODE deliveryMode,
             const xmsINT priority,
             const xmsLONG timeToLive) const;
```

Send a message to the destination that was specified when the message producer was created. Send the message using the specified delivery mode, priority, and time to live.

Parameters:

producer (input)
The handle for the message producer.

message (input)
In the C interface, this parameter is the handle for the message.

In the C++ interface, this parameter is the Message object.

deliveryMode (input)
The delivery mode for the message. If the connection uses WebSphere MQ Real-Time Transport, the value must be `XMSC_NON_PERSISTENT`.

priority (input)
The priority of the message. The value can be an integer in the range `0`, for the lowest priority, to `9`, for the highest priority. If the connection uses WebSphere MQ Real-Time Transport, the value is ignored.

timeToLive (input)
The time to live for the message in milliseconds. A value of zero means that the message never expires. If the connection uses WebSphere MQ Real-Time Transport, the value must be zero.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

Send (to a specified destination)

C interface:
```
xmsRC xmsMsgProducerSendDest(xmsHMsgProducer producer,
                             xmsHDest destination,
                             xmsHMsg message,
                             xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID send(const Destination & destination,
             const Message & message) const;
```

Send a message to a specified destination if you are using a message producer for which no destination was specified when the message producer was created. Send the message using the message producer's default delivery mode, priority, and time to live.

Typically, you specify a destination when you create a message producer but, if you do not, you must specify a destination every time you send a message.

Parameters:

producer (input)
The handle for the message producer.

destination (input)
In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

message (input)
In the C interface, this parameter is the handle for the message.

In the C++ interface, this parameter is the Message object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

Send (to a specified destination, specifying a delivery mode, priority, and time to live)

C interface:

```
xmsRC xmsMsgProducerSendWithAttr(xmsHMsgProducer producer,
                                 xmsHDest destination,
                                 xmsHMsg message,
                                 xmsDELIVERY_MODE deliveryMode,
                                 xmsINT priority,
                                 xmsLONG timeToLive,
                                 xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID send(const Destination & destination,
             const Message & message,
             const xmsDELIVERY_MODE deliveryMode,
             const xmsINT priority,
             const xmsLONG timeToLive) const;
```

Send a message to a specified destination if you are using a message producer for which no destination was specified when the message producer was created. Send the message using the specified delivery mode, priority, and time to live.

Typically, you specify a destination when you create a message producer but, if you do not, you must specify a destination every time you send a message.

Parameters:

producer (input)
The handle for the message producer.

destination (input)
In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

message (input)
In the C interface, this parameter is the handle for the message.

In the C++ interface, this parameter is the Message object.

deliveryMode (input)
The delivery mode for the message. If the connection uses WebSphere MQ Real-Time Transport, the value must be `XMSC_NON_PERSISTENT`.

priority (input)
: The priority of the message. The value can be an integer in the range `0`, for the lowest priority, to `9`, for the highest priority. If the connection uses WebSphere MQ Real-Time Transport, the value is ignored.

timeToLive (input)
: The time to live for the message in milliseconds. A value of zero means that the message never expires. If the connection uses WebSphere MQ Real-Time Transport, the value must be zero.

errorBlock (input)
: The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

Set Default Delivery Mode

C interface:
```
xmsRC xmsMsgProducerSetDeliveryMode(xmsHMsgProducer producer,
                                    xmsDELIVERY_MODE deliveryMode,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setDeliveryMode(const xmsDELIVERY_MODE deliveryMode);
```

Set the default delivery mode for messages sent by the message producer.

Parameters:

producer (input)
: The handle for the message producer.

deliveryMode (input)
: The default delivery mode. If the connection uses WebSphere MQ Real-Time Transport, the value must be `XMSC_NON_PERSISTENT`, which is the default value in this case.

errorBlock (input)
: The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Default Priority

C interface:

```
xmsRC xmsMsgProducerSetPriority(xmsHMsgProducer producer,
                                xmsINT priority,
                                xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setPriority(const xmsINT priority);
```

Set the default priority for messages sent by the message producer.

If the connection uses WebSphere MQ Real-Time Transport, the priority of a message is ignored.

Parameters:

producer (input)
The handle for the message producer.

priority (input)
The default message priority. The value can be an integer in the range `0`, for the lowest priority, to `9`, for the highest priority. The default value is `4`.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Default Time to Live

C interface:

```
xmsRC xmsMsgProducerSetTimeToLive(xmsHMsgProducer producer,
                                  xmsLONG timeToLive,
                                  xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setTimeToLive(const xmsLONG timeToLive);
```

Set the default length of time that a message exists before it expires. The time is measured from when the message producer sends the message.

Parameters:

producer (input)
The handle for the message producer.

timeToLive (input)
The default time to live in milliseconds. A value of zero means that a message never expires. If the connection uses WebSphere MQ Real-Time Transport, the value must be zero, which is the default value in this case.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Disable Message ID Flag

C interface:
```
xmsRC xmsMsgProducerSetDisableMsgID(xmsHMsgProducer producer,
                                    xmsBOOL msgIDDisabled,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setDisableMessageID(const xmsBOOL msgIDDisabled);
```

Indicate whether a receiving application uses message identifiers that are included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

msgIDDisabled (input)
The value xmsTRUE means that a receiving application does not use message identifiers included in messages sent by the message producer. The value xmsFALSE means that a receiving application does use message identifiers. The default value is xmsFALSE.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Disable Timestamp Flag

C interface:
```
xmsRC xmsMsgProducerSetDisableMsgTS(xmsHMsgProducer producer,
                                    xmsBOOL timestampDisabled,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setDisableMessageTimestamp(const xmsBOOL timestampDisabled);
```

Indicate whether a receiving application uses timestamps that are included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

timestampDisabled (input)
The value xmsTRUE means that a receiving application does not use timestamps included in messages sent by the message producer. The value xmsFALSE means that a receiving application does use timestamps. The default value is xmsFALSE.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:
```
xmsRC xmsMsgProducerSetProperty(xmsHMsgProducer producer,
                                xmsHProperty property,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

producer (input)
The handle for the message producer.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

Property

C++ inheritance hierarchy:
None

A Property object represents a property of an object. A Property object has three attributes:

Property name
The name of the property

Property value
The value of the property

Property type
The data type of the value of the property

This class is a helper class.

Constructors

Copy Property

C interface:

```
xmsRC xmsPropertyDuplicate(xmsHProperty property,
                           xmsHProperty *copiedProperty,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
Property(const Property & property);
```

Copy the Property object.

Parameters:

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

copiedProperty (output)
The handle for the copy of the Property object.

errorBlock (input)
The handle for an error block or a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Property

C interface:
Not applicable

C++ interface:

```
Property(const String & propertyName,
         const xmsBOOL propertyValue);
```

```
Property(const String & propertyName,
         const xmsBYTE *propertyValue,
         xmsSIZE length);

Property(const String & propertyName,
         const xmsBYTE propertyValue);

Property(const String & propertyName,
         const xmsCHAR16 propertyValue);

Property(const String & propertyName,
         const xmsDOUBLE propertyValue);

Property(const String & propertyName,
         const xmsFLOAT propertyValue);

Property(const String & propertyName,
         const xmsINT propertyValue);

Property(const String & propertyName,
         const xmsLONG propertyValue);

Property(const String & propertyName,
         const xmsSHORT propertyValue);

Property(const String & propertyName,
         const String & propertyValue);
```

Create a Property object with a property name, a property value, and a property type.

Parameters:

propertyName (input)
A String object encapsulating the property name.

propertyValue (input)
The property value. The property type is determined by the data type of the property value.

length (input)
The length of the property value in bytes. This parameter is applicable only if the property value is an array of bytes.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Property (with no property value or property type)

C interface:

```
xmsRC xmsPropertyCreate(xmsCHAR *propertyName,
                        xmsHProperty *property,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
Property(const String & propertyName);
```

Create a Property object with no property value or property type.

Parameters:

propertyName (input)
The property name.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the Property object is a null object.

Parameters:
None

C++ method returns:
- xmsTRUE, if the Property object is a null object.
- xmsFALSE, if the Property object is not a null object.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Check Property Type

C interface:
```
xmsRC xmsPropertyIsType(xmsHProperty property,
                        xmsPROPERTY_TYPE propertyType,
                        xmsBOOL *isType,
                        xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBOOL isType(const xmsPROPERTY_TYPE propertyType) const;
```

Check whether the Property object has the specified property type.

Parameters:

property (input)
The handle for the Property object.

propertyType (input)
: The property type, which must be one of the following values:

XMS_PROPERTY_TYPE_UNKNOWN
XMS_PROPERTY_TYPE_BOOL
XMS_PROPERTY_TYPE_BYTE
XMS_PROPERTY_TYPE_BYTEARRAY
XMS_PROPERTY_TYPE_CHAR
XMS_PROPERTY_TYPE_STRING
XMS_PROPERTY_TYPE_SHORT
XMS_PROPERTY_TYPE_INT
XMS_PROPERTY_TYPE_LONG
XMS_PROPERTY_TYPE_FLOAT
XMS_PROPERTY_TYPE_DOUBLE

isType (output)
: If the value is `xmsTRUE`, the Property object has the specified property type. If the value is `xmsFALSE`, the Property object does not have the specified property type.

errorBlock (input)
: The handle for an error block or a null handle.

C++ method returns:
See the description of the isType parameter.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete Property

C interface:

```
xmsRC xmsPropertyDispose(xmsHProperty *property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
virtual ~Property();
```

Delete the Property object.

If an application tries to delete a Property object that is already deleted, the call is ignored.

Parameters:

property (input/output)
: On input, the handle for the Property object. On output the call returns a null handle.

errorBlock (input)
: The handle for an error block or a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Boolean Property Value

C interface:
```
xmsRC xmsPropertyGetBoolean(xmsHProperty property,
                            xmsBOOL *propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBOOL getBoolean() const;
```

Get the boolean property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The boolean property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The boolean property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Array Property Value

C interface:
```
xmsRC xmsPropertyGetByteArray(xmsHProperty property,
                              xmsBYTE *propertyValue,
                              xmsSIZE length,
                              xmsSIZE *actualLength,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsBYTE * getByteArray(xmsBYTE *propertyValue,
                       const xmsSIZE length,
                       xmsSIZE *actualLength) const;
```

Get the byte array property value from the Property object.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33. If you are using the C++ interface, see "C++ methods that return a byte array" on page 38 instead.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The buffer to contain the property value, which is an array of bytes.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the property value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A pointer to the property value, which is an array of bytes.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Array Property Value by Reference

C interface:

```
xmsRC xmsPropertyGetByteArrayByRef(xmsHProperty property,
                                   xmsBYTE **propertyValue,
                                   xmsSIZE *length,
                                   xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the byte array property value in the Property object.

For more information about how to use this function, see "C functions that return a string or byte array by reference" on page 34.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
A pointer to the property value, which is an array of bytes.

length (output)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Property Value

C interface:

```
xmsRC xmsPropertyGetByte(xmsHProperty property,
                         xmsBYTE *propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsBYTE getByte() const;
```

Get the byte property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The byte property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The byte property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Character Property Value

C interface:

```
xmsRC xmsPropertyGetChar(xmsHProperty property,
                         xmsCHAR16 *propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsCHAR16 getChar() const;
```

Get the 2-byte character property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The 2-byte character property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The 2-byte character property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Double Precision Floating Point Property Value

C interface:

```
xmsRC xmsPropertyGetDouble(xmsHProperty property,
                           xmsDOUBLE *propertyValue,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsDOUBLE getDouble() const;
```

Get the double precision floating point property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The double precision floating point property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The double precision floating point property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Floating Point Property Value

C interface:

```
xmsRC xmsPropertyGetFloat(xmsHProperty property,
                          xmsFLOAT *propertyValue,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsFLOAT getFloat() const;
```

Get the floating point property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The floating point property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The floating point property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:
```
xmsHProperty getHandle() const;
```

Get the handle that a C application would use to access the Property object.

Parameters:
None

C++ method returns:
The handle for the Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property Value

C interface:
```
xmsRC xmsPropertyGetInt(xmsHProperty property,
                        xmsINT *propertyValue,
                        xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsINT getInt() const;
```

Get the integer property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The integer property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Long Integer Property Value

C interface:

```
xmsRC xmsPropertyGetLong(xmsHProperty property,
                         xmsLONG *propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsLONG getLong() const;
```

Get the long integer property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The long integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The long integer property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property Name

C interface:

```
xmsRC xmsPropertyGetName(xmsHProperty property,
                         xmsCHAR *propertyName,
                         xmsSIZE length,
                         xmsSIZE *actualLength,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
String name() const;
```

Get the property name from the Property object.

For more information about how to use this method in a C application , see "C functions that return a string or byte array by value" on page 33.

Parameters:

property (input)
The handle for the Property object.

propertyName (output)
The buffer to contain the property name.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the property name is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the property name in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the property name.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property Type

C interface:

```
xmsRC xmsPropertyGetType(xmsHProperty property,
                         xmsPROPERTY_TYPE *propertyType,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsPROPERTY_TYPE getType() const;
```

Get the property type from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyType (output)
The property type, which is one of the following values:

```
XMS_PROPERTY_TYPE_UNKNOWN
XMS_PROPERTY_TYPE_BOOL
XMS_PROPERTY_TYPE_BYTE
XMS_PROPERTY_TYPE_BYTEARRAY
XMS_PROPERTY_TYPE_CHAR
XMS_PROPERTY_TYPE_STRING
XMS_PROPERTY_TYPE_SHORT
XMS_PROPERTY_TYPE_INT
XMS_PROPERTY_TYPE_LONG
XMS_PROPERTY_TYPE_FLOAT
XMS_PROPERTY_TYPE_DOUBLE
```

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
See the description of the propertyType parameter.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Short Integer Property Value

C interface:

```
xmsRC xmsPropertyGetShort(xmsHProperty property,
                          xmsSHORT *propertyValue,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsSHORT getShort() const;
```

Get the short integer property value from the Property object.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The short integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The short integer property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property Value

C interface:

```
xmsRC xmsPropertyGetString(xmsHProperty property,
                           xmsCHAR *propertyValue,
                           xmsSIZE length,
                           xmsSIZE *actualLength,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
String getString() const;
```

Get the string property value from the Property object.

For more information about how to use this method in a C application, see "C functions that return a string or byte array by value" on page 33.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The buffer to contain the string property value.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the property value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
A String object encapsulating the string property value.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property Value by Reference

C interface:
```
xmsRC xmsPropertyGetStringByRef(xmsHProperty property,
                                xmsCHAR **propertyValue,
                                xmsSIZE *length,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
Not applicable

Get a pointer to the string property value in the Property object.

For more information about how to use this function, see "C functions that return a string or byte array by reference" on page 34.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
A pointer to the string property value.

length (output)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Boolean Property Value

C interface:
```
xmsRC xmsPropertySetBoolean(xmsHProperty property,
                            xmsBOOL propertyValue,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setBoolean(const xmsBOOL propertyValue);
```

Set a boolean property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The boolean property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Byte Array Property Value

C interface:

```
xmsRC xmsPropertySetByteArray(xmsHProperty property,
                              xmsBYTE *propertyValue,
                              xmsSIZE length,
                              xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setByteArray(const xmsBYTE *propertyValue,
                     const xmsSIZE length);
```

Set a byte array property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The property value, which is an array of bytes.

length (input)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Byte Property Value

C interface:

```
xmsRC xmsPropertySetByte(xmsHProperty property,
                         xmsBYTE propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setByte(const xmsBYTE propertyValue);
```

Set a byte property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The byte property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Character Property Value

C interface:

```
xmsRC xmsPropertySetChar(xmsHProperty Property,
                         xmsCHAR16 propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setChar(const xmsCHAR16 propertyValue);
```

Set a 2-byte character property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The 2-byte character property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Double Precision Floating Point Property Value

C interface:

```
xmsRC xmsPropertySetDouble(xmsHProperty property,
                           xmsDOUBLE propertyValue,
                           xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setDouble(const xmsDOUBLE propertyValue);
```

Set a double precision floating point property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The double precision floating point property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Floating Point Property Value

C interface:

```
xmsRC xmsPropertySetFloat(xmsHProperty property,
                          xmsFLOAT propertyValue,
                          xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setFloat(const xmsFLOAT propertyValue);
```

Set a floating point property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The floating point property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Integer Property Value

C interface:

```
xmsRC xmsPropertySetInt(xmsHProperty property,
                        xmsINT propertyValue,
                        xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setInt(const xmsINT propertyValue);
```

Set an integer property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Long Integer Property Value

C interface:

```
xmsRC xmsPropertySetLong(xmsHProperty property,
                         xmsLONG propertyValue,
                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsVOID setLong(const xmsLONG propertyValue);
```

Set a long integer property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The long integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Short Integer Property Value

C interface:
```
xmsRC xmsPropertySetShort(xmsHProperty property,
                          xmsSHORT propertyValue,
                          xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setShort(const xmsSHORT propertyValue);
```

Set a short integer property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The short integer property value.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set String Property Value

C interface:
```
xmsRC xmsPropertySetString(xmsHProperty property,
                           xmsCHAR *propertyValue,
                           xmsSIZE length,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID setString(const String & propertyValue);
```

Set a string property value in the Property object and set the property type.

Parameters:

property (input)
The handle for the Property object.

propertyValue (input)
The string property value.

In the C interface, this parameter is a character array.

In the C++ interface, this parameter is a String object.

length (input)
The length of the property value in bytes.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

PropertyContext

C++ inheritance hierarchy:
None

PropertyContext is an abstract superclass that encapsulates methods that get and set properties. These methods are inherited by other classes.

Only the C++ interface uses this class.

Methods

Get Boolean Property

C++ interface:
```
xmsBOOL getBooleanProperty(const String & propertyName) const;
```

Get the value of the boolean property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Property

C++ interface:
```
xmsBYTE getByteProperty(const String & propertyName) const;
```

Get the value of the byte property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Byte Array Property

C++ interface:

```
xmsBYTE * getBytesProperty(const String & propertyName,
                           xmsBYTE *propertyValue,
                           const xmsSIZE length,
                           xmsSIZE *actualLength) const;
```

Get the value of the byte array property identified by name.

For more information about how to use this method, see “C++ methods that return a byte array” on page 38.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (output)
The buffer to contain the value of the property, which is an array of bytes.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the array of bytes is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The number of bytes in the array.

C++ method returns:
A pointer to the value of the property, which is an array of bytes.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Character Property

C++ interface:

```
xmsCHAR16 getCharProperty(const String & propertyName) const;
```

Get the value of the 2-byte character property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Double Precision Floating Point Property

C++ interface:

```
xmsDOUBLE getDoubleProperty(const String & propertyName) const;
```

Get the value of the double precision floating point property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Floating Point Property

C++ interface:

```
xmsFLOAT getFloatProperty(const String & propertyName) const;
```

Get the value of the floating point property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Integer Property

C++ interface:

```
xmsINT getIntProperty(const String & propertyName) const;
```

Get the value of the integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Long Integer Property

C++ interface:
```
xmsLONG getLongProperty(const String & propertyName) const;
```

Get the value of the long integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Short Integer Property

C++ interface:
```
xmsSHORT getShortProperty(const String & propertyName) const;
```

Get the value of the short integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String Property

C++ interface:
```
String getStringProperty(const String & propertyName) const;
```

Get the value of the string property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

C++ method returns:
A String object encapsulating the string that is the value of the property. If data conversion is required, this is the string after conversion.

Thread context:
Determined by the subclass

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Boolean Property

C++ interface:

```
xmsVOID setBooleanProperty(const String & propertyName,
                           const xmsBOOL propertyValue);
```

Set the value of the boolean property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Byte Property

C++ interface:

```
xmsVOID setByteProperty(const String & propertyName,
                        const xmsBYTE propertyValue);
```

Set the value of the byte property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Byte Array Property

C++ interface:

```
xmsVOID setBytesProperty(const String & propertyName,
                         const xmsBYTE *propertyValue,
                         const xmsSIZE length);
```

Set the value of the byte array property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property, which is an array of bytes.

length (input)
The number of bytes in the array.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Character Property

C++ interface:

```
xmsVOID setCharProperty(const String & propertyName,
                        const xmsCHAR16 propertyValue);
```

Set the value of the 2-byte character property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Double Precision Floating Point Property

C++ interface:

```
xmsVOID setDoubleProperty(const String & propertyName,
                          const xmsDOUBLE propertyValue);
```

Set the value of the double precision floating point property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Floating Point Property

C++ interface:

```
xmsVOID setFloatProperty(const String & propertyName,
                         const xmsFLOAT propertyValue);
```

Set the value of the floating point property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Integer Property

C++ interface:

```
xmsVOID setIntProperty(const String & propertyName,
                       const xmsINT propertyValue);
```

Set the value of the integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Long Integer Property

C++ interface:

```
xmsVOID setLongProperty(const String & propertyName,
                        const xmsLONG propertyValue);
```

Set the value of the long integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set Short Integer Property

C++ interface:

```
xmsVOID setShortProperty(const String & propertyName,
                         const xmsSHORT propertyValue);
```

Set the value of the short integer property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
The value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Set String Property

C++ interface:

```
xmsVOID setStringProperty(const String & propertyName,
                          const String & propertyValue);
```

Set the value of the string property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

propertyValue (input)
A String object encapsulating the string that is the value of the property.

C++ method returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

ResourceAllocationException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::ResourceAllocationException
```

XMS throws this exception if XMS cannot allocate the resources required by a method.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

SecurityException

C++ inheritance hierarchy:

```
std::exception
   |
   +----xms::Exception
           |
           +----xms::SecurityException
```

XMS throws this exception if the user identifer and password provided to authenticate an application are rejected. XMS also throws this exception if an authority check fails and prevents a method from completing.

Only the C++ interface uses this class.

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getJMSException, getHandle, getLinkedException, isNull

Session

C++ inheritance hierarchy:

```
xms::PropertyContext
   |
   +----xms::Session
```

A session is a single threaded context for sending and receiving messages.

Methods

Check Whether Null

C interface:
Not applicable

C++ interface:
```
xmsBOOL isNull() const;
```

Determine whether the Session object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the Session object is a null object.
- `xmsFALSE`, if the Session object is not a null object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Close Session

C interface:
```
xmsRC xmsSessClose(xmsHSess *session,
                   xmsHErrorBlock errorBlock);
```

C++ interface:
```
xmsVOID close();
```

Close the session.

All objects dependent on the session are deleted. For information about which objects are deleted, see “Deleting objects” on page 26.

If an application tries to close a session that is already closed, the call is ignored.

Parameters:

session (input/output)
On input, the handle for the session. On output, the call returns a null handle.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Bytes Message

C interface:
```
xmsRC xmsSessCreateBytesMessage(xmsHSess session,
                                xmsHMsg *message,
                                xmsHErrorBlock errorBlock);
```

C++ interface:
```
BytesMessage createBytesMessage() const;
```

Create a bytes message.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the bytes message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The BytesMessage object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Consumer

C interface:
```
xmsRC xmsSessCreateConsumer(xmsHSess session,
                            xmsHDest destination,
                            xmsHMsgConsumer *consumer,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
```
MessageConsumer createConsumer(const Destination & destination) const;
```

Create a message consumer for the specified destination.

Parameters:

session (input)
The handle for the session.

destination (input)
In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

consumer (output)
The handle for the message consumer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The MessageConsumer object.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

Create Consumer (with message selector)

C interface:
```
xmsRC xmsSessCreateConsumerSelector(xmsHSess session,
                                    xmsHDest destination,
                                    xmsCHAR *messageSelector,
                                    xmsSIZE length,
                                    xmsHMsgConsumer *consumer,
                                    xmsHErrorBlock errorBlock);
```

C++ interface:
```
MessageConsumer createConsumer(const Destination & destination,
                               const String & messageSelector) const;
```

Create a message consumer for the specified destination using a message selector.

Parameters:

session (input)
The handle for the session.

destination (input)
In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

messageSelector (input)
A message selector expression. Only those messages with properties that match the message selector expression are delivered to the message consumer.

In the C interface, this parameter is a character array. A value of null or an empty string indicates that there is no message selector for the message consumer.

In the C++ interface, this parameter is a String object. A null String object indicates that there is no message selector for the message consumer.

length (input)
The length of the message selector expression in bytes.

consumer (output)
The handle for the message consumer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The MessageConsumer object.

Exceptions:
- XMS_X_GENERAL_EXCEPTION

- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

Create Consumer (with message selector and local message flag)

C interface:

```
xmsRC xmsSessCreateConsumerSelectorLocal(xmsHSess session,
                                         xmsHDest destination,
                                         xmsCHAR *messageSelector,
                                         xmsSIZE length,
                                         xmsBOOL noLocal,
                                         xmsHMsgConsumer *consumer,
                                         xmsHErrorBlock errorBlock);
```

C++ interface:

```
MessageConsumer createConsumer(const Destination & destination,
                               const String & messageSelector,
                               const xmsBOOL noLocal) const;
```

Create a message consumer for the specified destination using a message selector, and specifying whether the message consumer can receive messages published by its own connection.

Parameters:

session (input)
The handle for the session.

destination (input)
In the C interface, this parameter is the handle for the destination.

In the C++ interface, this parameter is the Destination object.

messageSelector (input)
A message selector expression. Only those messages with properties that match the message selector expression are delivered to the message consumer.

In the C interface, this parameter is a character array. A value of null or an empty string indicates that there is no message selector for the message consumer.

In the C++ interface, this parameter is a String object. A null String object indicates that there is no message selector for the message consumer.

length (input)
The length of the message selector expression in bytes.

noLocal (input)
If the value is `xmsTRUE`, the message consumer does not receive the messages published by its own connection. By default, a message consumer receives messages published by its own connection.

consumer (output)
The handle for the message consumer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The MessageConsumer object.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

Create Map Message

C interface:
```
xmsRC xmsSessCreateMapMessage(xmsHSess session,
                              xmsHMsg *message,
                              xmsHErrorBlock errorBlock);
```

C++ interface:
```
MapMessage createMapMessage() const;
```

Create a map message.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the map message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The MapMessage object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Message

C interface:
```
xmsRC xmsSessCreateMessage(xmsHSess session,
                           xmsHMsg *message,
                           xmsHErrorBlock errorBlock);
```

C++ interface:
```
Message createMessage() const;
```

Create a message that has no body.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the message.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Message object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create Producer

C interface:
```
xmsRC xmsSessCreateProducer(xmsHSess session,
                            xmsHDest destination,
                            xmsHMsgProducer *producer,
                            xmsHErrorBlock errorBlock);
```

C++ interface:
```
MessageProducer createProducer(const Destination & destination) const;
```

Create a message producer to send messages to the specified destination.

Parameters:

session (input)
The handle for the session.

destination (input)
In the C interface, this parameter is the handle for the destination. If you specify a null handle, the message producer is created without a destination.

In the C++ interface, this parameter is the Destination object. If you specify a null Destination object, the message producer is created without a destination.

producer (output)
The handle for the message producer.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The MessageProducer object.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

Create Topic

C interface:
Not implemented. Use one of the following methods instead:
- “Create Destination (using a URI)” on page 96
- “Create Destination (specifying a type and name)” on page 96

C++ interface:
```
Destination createTopic(const String & topicName) const;
```

Create a topic.

Parameter:

topicName (input)
A String object encapsulating the name of the topic or a topic URI.

C++ method returns:
A Destination object for the topic.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Acknowledgement Mode

C interface:

```
xmsRC xmsSessGetAcknowledgeMode(xmsHSess session,
                                xmsACKNOWLEDGE_MODE *acknowledgeMode,
                                xmsHErrorBlock errorBlock);
```

C++ interface:

```
xmsACKNOWLEDGE_MODE getAcknowledgeMode() const;
```

Get the acknowledgement mode for the session. The acknowledgement mode is specified when the session is created.

Parameters:

session (input)
The handle for the session.

acknowledgeMode (output)
The acknowledgement mode, which is `XMSC_AUTO_ACKNOWLEDGE`.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The acknowledgement mode, which is `XMSC_AUTO_ACKNOWLEDGE`.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Handle

C interface:
Not applicable

C++ interface:

```
xmsHSess getHandle() const;
```

Get the handle that a C application would use to access the session.

Parameters:
None

C++ method returns:
The handle for the session.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Property

C interface:
```
xmsRC xmsSessGetProperty(xmsHSess session,
                         xmsCHAR *propertyName,
                         xmsHProperty *property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual Property getProperty(const String & propertyName) const;
```

Get a Property object for the property identified by name.

Parameters:

session (input)
The handle for the session.

propertyName (input)
The name of the property.

In the C interface, this parameter is a null terminated string.

In the C++ interface, this parameter is a String object.

property (output)
The handle for the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
The Property object.

Exceptions:
XMS_X_GENERAL_EXCEPTION

Set Property

C interface:
```
xmsRC xmsSessSetProperty(xmsHSess session,
                         xmsHProperty property,
                         xmsHErrorBlock errorBlock);
```

C++ interface:
```
virtual xmsVOID setProperty(const Property & property);
```

Set the value of a property using a Property object.

Parameters:

session (input)
The handle for the session.

property (input)
In the C interface, this parameter is the handle for the Property object.

In the C++ interface, this parameter is the Property object.

errorBlock (input)
The handle for an error block or a null handle.

C++ method returns:
Void

Exceptions:
XMS_X_GENERAL_EXCEPTION

Inherited methods in the C++ interface

The following methods are inherited from the PropertyContext class:

getBooleanProperty, getByteProperty, getBytesProperty, getCharProperty, getDoubleProperty, getFloatProperty, getIntProperty, getLongProperty, getShortProperty, getStringProperty, setBooleanProperty, setByteProperty, setBytesProperty, setCharProperty, setDoubleProperty, setFloatProperty, setIntProperty, setLongProperty, setShortProperty, setStringProperty

String

C++ inheritance hierarchy:
None

A String object encapsulates a string.

This class is a helper class, and only the C++ interface uses the class.

Constructors

Create String

C++ interface:

```
String();
```

Create a String object that encapsulates a null string.

Parameters:
None

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create String (from a byte array)

C++ interface:

```
String(const xmsBYTE *value,
       const xmsSIZE length);
```

Create a String object from an array of bytes.

Parameters:

value (input)
The array of bytes that is copied to form the string encapsulated by the String object.

length (input)
The number of bytes in the array.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Create String (from a character array)

C++ interface:

```
String(const xmsCHAR *value);
```

Create a String object from an array of characters.

Parameters:

value (input)
The null terminated array of characters that is copied to form the string encapsulated by the String object.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Methods

Check Whether Null

C++ interface:

```
xmsBOOL isNull() const;
```

Determine whether the String object is a null object.

Parameters:
None

C++ method returns:
- `xmsTRUE`, if the String object is a null object.
- `xmsFALSE`, if the String object is not a null object.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Compare Strings

C++ interface:

```
xmsBOOL equalTo(const String & string) const;
```

Determine whether the string encapsulated by the String object is equal to the string encapsulated by a second String object.

Parameters:

string (input)
The second String object.

C++ method returns:
- `xmsTRUE`, if the two strings are equal.
- `xmsFALSE`, if the two strings are not equal.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Concatenate Strings

C++ interface:

```
String & concatenate(const String & string) const;
```

Concatenate the string encapsulated by the String object with the string encapsulated by a second String object.

Parameters:

string (input)
The second String object.

C++ method returns:
The original String object encapsulating the concatenated strings.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Delete String

C++ interface:

```
virtual ~String();
```

Delete the String object.

Parameters:
None

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get Pointer to String

C++ interface:

```
xmsCHAR * c_str() const;
```

Get a pointer to the string encapsulated by the String object.

Parameters:
None

C++ method returns:
A pointer to the string encapsulated by the String object.

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Get String

C++ interface:

```
xmsVOID get(xmsBYTE *value,
            const xmsSIZE length,
            xmsSIZE *actualLength) const;
```

Get the string encapsulated by the String object.

For more information about how to use this method, see “C++ methods that return a byte array” on page 38.

Parameters:

value (output)
The buffer to contain the string.

length (input)
The length of the buffer in bytes. If you specify a length of zero, the string is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the string in bytes.

C++ method returns:
Void

Thread context:
Any

Exceptions:
XMS_X_GENERAL_EXCEPTION

Chapter 9. Properties of XMS objects

This chapter describes the properties of objects that are defined by XMS. For the properties of a Message object however, see Chapter 7, "XMS messages," on page 47.

The name of each property is defined as a named constant in the header file, xmsc.h, and so an application can use the name to identify the property.

If an application defines its own properties of the objects discussed in this chapter, it does not cause an error but it might cause unpredictable results.

The chapter contains the following sections:

- "Properties of Connection"
- "Properties of ConnectionFactory"
- "Properties of ConnectionMetaData" on page 238
- "Properties of Destination" on page 239
- "Properties of MessageConsumer" on page 240
- "Properties of Session" on page 241

Properties of Connection

Table 16 lists the XMS defined properties of a Connection object. For each property, the table provides its name, data type, description, and valid values.

Table 15. Properties of Connection

Name of property	Data type	Description	Valid values
XMSC_CLIENT_CCSID[1]	xmsINT	The identifier of the coded character set, or code page, that the application uses for the connection.	A coded character set identifier (CCSID)

Notes:
1. For information about how this property is set, see "Coded character set identifiers (CCSIDs)" on page 28.

Properties of ConnectionFactory

Table 16 lists the XMS defined properties of a ConnectionFactory object. For each property, the table provides its name, data type, description, and valid values, including its default value.

Table 16. Properties of ConnectionFactory

Name of property	Data type	Description	Valid values (default value in bold)
XMSC_CLIENT_CCSID[1]	xmsINT	The identifier of the coded character set, or code page, that the application uses for a connection.	A coded character set identifier (CCSID)

Table 16. Properties of ConnectionFactory (continued)

Name of property	Data type	Description	Valid values (default value in bold)
XMSC_HOST_NAME	String[2]	The host name or IP address of the system on which the broker resides.	• **Not set** • localhost • A host name • An IP address
XMSC_MULTICAST[3]	xmsINT	The multicast setting for the connection factory. This setting is used only when the multicast setting for a destination is XMSC_MULTICAST_ASCF.	• **XMSC_MULTICAST_DISABLED** • XMSC_MULTICAST_NOT_RELIABLE • XMSC_MULTICAST_RELIABLE • XMSC_MULTICAST_ENABLED
XMSC_PASSWORD	Byte array[4]	A password that can be used to authenticate the application when it attempts to create a connection. The password is used in conjunction with the value of XMSC_USERID.	• **Not set** • An array of bytes
XMSC_PORT	xmsINT	The port number on which the broker listens for published messages and subscription requests.	• **Not set** • A port number
XMSC_TRANSPORT_TYPE	xmsINT	The means by which an application connects to a broker.	**XMSC_TP_DIRECT_TCPIP** - The connection uses WebSphere MQ Real-Time Transport or WebSphere MQ Multicast Transport.
XMSC_USERID	String[2]	A user identifier that can be used to authenticate the application when it attempts to create a connection. The user identifier is used in conjunction with the value of XMSC_PASSWORD.	• **Not set** • A null terminated string

Notes:

1. For information about how this property is set, see "Coded character set identifiers (CCSIDs)" on page 28.
2. This is the data type if you are using C++. If you are programming in C, it is a character array.
3. For a full description of this property, and a description of each of its valid values, see Table 18 on page 240.
4. A value with this data type can be set using the Set Byte Array Property Value method of a Property object.

Properties of ConnectionMetaData

Table 17 on page 239 lists the XMS defined properties of a ConnectionMetaData object. For each property, the table provides its name, data type, description, and value.

Table 17. Properties of ConnectionMetaData

Name of property	Data type	Description	Value
XMSC_JMS_MAJOR_VERSION	xmsINT	The major version number of the JMS specification upon which XMS is based.	1
XMSC_JMS_MINOR_VERSION	xmsINT	The minor version number of the JMS specification upon which XMS is based.	1
XMSC_JMS_VERSION	String[1]	The version identifier of the JMS specification upon which XMS is based.	"1.1"
XMSC_MAJOR_VERSION	xmsINT	The version number of the XMS client.	1
XMSC_MINOR_VERSION	xmsINT	The release number of the XMS client.	0
XMSC_PROVIDER_NAME	String[1]	The provider of the XMS client.	"IBM"
XMSC_VERSION	String[1]	The version identifier of the XMS client.	"1.0"

Notes:

1. This is the data type if you are using C++. If you are programming in C, it is a character array.

Properties of Destination

Table 18 on page 240 lists the XMS defined properties of a Destination object. For each property, the table provides its name, data type, description, and valid values, including its default value.

Table 18. Properties of Destination

Name of property	Data type	Description	Valid values (default value in bold)
XMSC_MULTICAST	xmsINT	The multicast setting for the destination. The setting determines how messages are delivered to a message consumer that receives messages from the destination. Only a destination that is a topic can have this property. The property has no effect on how a message producer sends messages to the destination.	• **XMSC_MULTICAST_ASCF** - Messages are delivered to the message consumer according to the multicast setting for the connection factory associated with the message consumer. The multicast setting for the connection factory is noted at the time the message consumer is created. • XMSC_MULTICAST_DISABLED - Messages are not delivered to the message consumer using WebSphere MQ Multicast Transport. • XMSC_MULTICAST_NOT_RELIABLE - If the topic is configured for multicast in the broker, messages are delivered to the message consumer using WebSphere MQ Multicast Transport. A reliable quality of service is not used even if the topic is configured for reliable multicast. • XMSC_MULTICAST_RELIABLE - If the topic is configured for reliable multicast in the broker, messages are delivered to the message consumer using WebSphere MQ Multicast Transport with a reliable quality of service. If the topic is not configured for reliable multicast, you cannot create a message consumer for the topic. • XMSC_MULTICAST_ENABLED - If the topic is configured for multicast in the broker, messages are delivered to a message consumer using WebSphere MQ Multicast Transport. A reliable quality of service is used if the topic is configured for reliable multicast.
XMSC_PRIORITY[1]	xmsINT	The priority of messages sent to the destination.	• **XMSC_PROPERTY_AS_APP** - A message has the priority specified by the Send call. If the Send call specifies no priority, the default priority of the message producer is used instead. • An integer in the range 0, for the lowest priority, to 9, for the highest priority - A message has the priority specified for the destination. The default priority of the message producer and any priority specified on the Send call are ignored.

Notes:

1. WebSphere MQ Real-Time Transport and WebSphere MQ Multicast Transport take no action based upon the priority of a message. The priority of a message is honoured only if the message is eventually put on a WebSphere MQ queue and the `MsgDeliverySequence` attribute of the queue is MQMDS_PRIORITY.

Properties of MessageConsumer

Table 19 on page 241 lists the XMS defined properties of a MessageConsumer object. For each property, the table provides its name, data type, description, and valid values.

Table 19. Properties of MessageConsumer

Name of property	Data type	Description	Valid values
XMSC_IS_SUBSCRIPTION_ MULTICAST	xmsBOOL	Indicates whether messages are being delivered to the message consumer using WebSphere MQ Multicast Transport.	• xmsTRUE - Messages are being delivered using WebSphere MQ Multicast Transport. • xmsFALSE - Messages are not being delivered using WebSphere MQ Multicast Transport.
XMSC_IS_SUBSCRIPTION_ RELIABLE_MULTICAST	xmsBOOL	Indicates whether messages are being delivered to the message consumer using WebSphere MQ Multicast Transport with a reliable quality of service.	• xmsTRUE - Messages are being delivered using WebSphere MQ Multicast Transport with a reliable quality of service. • xmsFALSE - Messages are not being delivered using WebSphere MQ Multicast Transport with a reliable quality of service.

Properties of Session

Table 20 lists the XMS defined properties of a Session object. For each property, the table provides its name, data type, description, and valid values.

Table 20. Properties of Session

Name of property	Data type	Description	Valid values
XMSC_CLIENT_CCSID[1]	xmsINT	The identifier of the coded character set, or code page, that the application uses for the session.	A coded character set identifier (CCSID)

Notes:

1. For information about how this property is set, see "Coded character set identifiers (CCSIDs)" on page 28.

Appendix. Notices

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

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

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

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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

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

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

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

© Copyright IBM Corp. 2004

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

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

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

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

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

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

Trademarks

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

IBM	WebSphere

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

Other company, product, or service names may be trademarks or service marks of others.

Index

A

applications, building
  C 18
  C++ 19
applications, sample
  C
    building 18
    description 15
    running 17
  C++
    building 18
    description 16
  WebSphere MQ JMS 16
applications, writing
  general 23
  in C 31
  in C++ 37
assigning XMS objects in C++ 43
asynchronous message delivery 25
attribute
  introduction 6
  property value
    description 193
    setting 27

B

body types of messages 50
building
  C applications
    sample 18
    your own 18
  C++ applications
    sample 18
    your own 19
byte array
  C functions returning by reference 34
  C functions returning by value 33
  C++ methods returning 38
bytes message 52
BytesMessage class
  interface definition 61

C

C
  building applications 18
  functions returning a string or byte array by reference 34
  functions returning a string or byte array by value 33
  handling errors 34
  object handles
    data types 31
    introduction 6
  sample applications
    building 18
    description 15
    running 17
  supported compilers 7

C *(continued)*
  using exception listener functions 32
  using message listener functions 31
  writing applications 31
C++
  application, using the C API 45
  assigning XMS objects 43
  building applications 19
  handling errors 39
  methods returning a byte array 38
  sample applications
    building 18
    description 16
  supported compilers 7
  using exception listeners 43
  using message listeners 41
  using namespaces 37
  using the PropertyContext class 39
  using the String class 38
  writing applications 37
CCSID
  See coded character set identifier (CCSID)
closing a connection 24
coded character set identifier (CCSID) 28
compilers, supported 7
Connection class
  interface definition 76
  introduction 5
Connection object, properties 237
ConnectionFactory class
  interface definition 83
  introduction 5
ConnectionFactory object, properties 237
ConnectionMetaData class
  interface definition 92
ConnectionMetaData object, properties 238
connections
  closing 24
  handling exceptions 24
  introduction 23
  starting 24
  stopping 24
context data 32

D

data types compatible with Java 51
deleting objects 26
delivering messages to an application
  asynchronously 25
  synchronously 25
Destination class
  interface definition 96
  introduction 5
Destination object, properties 239

E

error block 35
ErrorBlock class
  interface definition 105
errors
  handling in C 34
  handling in C++ 39
Exception class
  interface definition 109
exception listener functions, using in C 32
exception listeners, using in C++ 43
ExceptionListener class
  interface definition 113

H

handles, object
  data types 31
  introduction 6
handling errors
  in C 34
  in C++ 39
handling exceptions on a connection 24

I

IllegalStateException class
  interface definition 114
installed directories
  Linux 10
  Windows 11
installing XMS
  Linux 9
  Windows 9
InvalidDestinationException class
  interface definition 115
InvalidSelectorException class
  interface definition 116
Iterator class
  interface definition 117
iterators 27

J

Java compatible data types 51

L

Linux
  installed directories 10
  installing XMS 9
  supported compilers 7
  uninstalling XMS 12

M

map message 52

© Copyright IBM Corp. 2004

MapMessage class
  interface definition 120
message
  body 50
  body type
    bytes 52
    map 52
  bytes 52
  delivery
    asynchronous 25
    synchronous 25
  header fields 47
  map 52
  properties
    application defined 50
    IBM defined 49
    introduction 48
    JMS defined 49
  selectors 53
  structure 47
Message class
  interface definition 139
  introduction 5
message listener functions, using in C 31
message listeners, using in C++ 41
Message object
  header fields 47
  properties
    application defined 50
    IBM defined 49
    introduction 48
    JMS defined 49
MessageConsumer class
  interface definition 169
  introduction 5
MessageConsumer object, properties 240
MessageEOFException class
  interface definition 176
MessageFormatException class
  interface definition 177
MessageListener class
  interface definition 178
MessageNotReadableException class
  interface definition 179
MessageNotWritableException class
  interface definition 180
MessageProducer class
  interface definition 181
  introduction 5
messaging
  point-to-point 3
  publish/subscribe 3
  styles 3
  transports 4

N

namespaces, using in C++ 37

O

object handles
  data types 31
  introduction 6
object model, XMS 5
objects, deleting 26
operating environments, supported 7

P

point-to-point messaging 3
problem determination 19
properties
  Connection object 237
  ConnectionFactory object 237
  ConnectionMetaData object 238
  Destination object 239
  Message object
    application defined 50
    IBM defined 49
    introduction 48
    JMS defined 49
  MessageConsumer object 240
  Session object 241
property
  introduction 6
  setting the value 27
  XMSC_CLIENT_CCSID
    Connection object 237
    ConnectionFactory object 237
    Session object 241
    use in code page conversion 28
  XMSC_HOST_NAME 237
  XMSC_IS_SUBSCRIPTION_MULTICAST 240
  XMSC_IS_SUBSCRIPTION_RELIABLE_MULTICAST 240
  XMSC_JMS_MAJOR_VERSION 238
  XMSC_JMS_MINOR_VERSION 238
  XMSC_JMS_VERSION 238
  XMSC_MAJOR_VERSION 238
  XMSC_MINOR_VERSION 238
  XMSC_MULTICAST
    ConnectionFactory object 237
    Destination object 239
  XMSC_PASSWORD 237
  XMSC_PORT 237
  XMSC_PRIORITY 239
  XMSC_PROVIDER_NAME 238
  XMSC_TRANSPORT_TYPE 237
  XMSC_USERID 237
  XMSC_VERSION 238
Property class
  interface definition 193
Property object, property value attribute
  description 193
  setting 27
property value attribute
  description 193
  setting 27
PropertyContext class
  interface definition 212
  using in C++ 39
publish/subscribe messaging 3

R

receiving messages
  asynchronously 25
  synchronously 25
removing XMS
  Linux 12
  Windows
    by running the uninstaller program 12
    using Add/Remove Programs 13
ResourceAllocationException class
  interface definition 221
return codes 34
running the C sample applications 17

S

sample applications
  C
    building 18
    description 15
    running 17
  C++
    building 18
    description 16
  WebSphere MQ JMS 16
SecurityException class
  interface definition 222
selectors, message 53
Session class
  interface definition 223
  introduction 5
Session object, properties 241
sessions
  asynchronous message delivery 25
  introduction 25
  synchronous message delivery 25
setting the property value attribute of a Property object 27
setting the value of a property 27
starting a connection 24
stopping a connection 24
string
  C functions returning by reference 34
  C functions returning by value 33
String class
  interface definition 232
  using in C++ 38
structure of a message 47
styles of messaging 3
supported compilers 7
supported operating environments 7
synchronous message delivery 25

T

threading model 23
tracing 19
trademarks 244
transports, messaging 4

U

uniform resource identifier (URI) 26
uninstalling XMS
  Linux 12
  Windows
    by running the uninstaller program 12
    using Add/Remove Programs 13

URI
  See uniform resource identifier (URI)

W

WebSphere MQ Enterprise Transport 4
WebSphere MQ JMS sample applications 16
WebSphere MQ Multicast Transport 4
WebSphere MQ Real-Time Transport 4
Windows
  installed directories 11
  installing XMS 9
  supported compilers 7
  uninstalling XMS
    by running the uninstaller program 12
    using Add/Remove Programs 13
writing applications
  general 23
  in C 31
  in C++ 37

X

XMS
  classes 59
  first release 7
  installing 9
  messages 47
  object model 5
  supported compilers 7
  supported operating environments 7
  threading model 23
XMSC_CLIENT_CCSID property
  Connection object 237
  ConnectionFactory object 237
  Session object 241
  use in code page conversion 28
XMSC_HOST_NAME property 237
XMSC_IS_SUBSCRIPTION_MULTICAST property 240
XMSC_IS_SUBSCRIPTION_RELIABLE_MULTICAST property 240
XMSC_JMS_MAJOR_VERSION property 238
XMSC_JMS_MINOR_VERSION property 238
XMSC_JMS_VERSION property 238
XMSC_MAJOR_VERSION property 238
XMSC_MINOR_VERSION property 238
XMSC_MULTICAST property
  ConnectionFactory object 237
  Destination object 239
XMSC_PASSWORD property 237
XMSC_PORT property 237
XMSC_PRIORITY property 239
XMSC_PROVIDER_NAME property 238
XMSC_TRANSPORT_TYPE property 237
XMSC_USERID property 237
XMSC_VERSION property 238

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–816151
  - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

© Copyright IBM Corp. 2004

Printed in USA

SC34-6363-01