3.0

IBM Message Service Client (XMS) for C/C++

Note

Before using this information and the product it supports, read the information in "Notices" on page 503.

This edition applies to version 3 release 0 of IBM® MQ Message Service Client for C/C++ and to all subsequent releases and modifications until otherwise indicated in new editions.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures ix

Chapter 1. Welcome to the documentation for Message Service Client for C/C++..... 1

What's new in this release..... 1

Chapter 2. Introduction to Message Service Client for C/C++..... 3

What is Message Service Client for C/C++? 3
Styles of messaging 4
The XMS object model..... 5
Attributes and properties of objects..... 6
Administered objects 7
The XMS message model 8
Operating environments..... 8
Prerequisites for XMS applications connecting to IBM MQ 9

Chapter 3. Installing Message Service Client for C/C++.....11

Installing Message Service Client for C/C++ 11
Installing Message Service Client for C/C++ using the installation wizard..... 11
Installing from the command line 14
What is installed on AIX, Linux, and Solaris 16
What is installed on Windows (C/C++) 17
Uninstalling Message Service Client for C/C++ 18

Chapter 4. Setting up the messaging server environment..... 21

Configuring the queue manager for an application that connects to a IBM MQ queue manager..... 21
Configuring the service integration bus for an application that connects to a WebSphere service integration bus 24

Chapter 5. Developing XMS applications 25

Writing XMS applications 25
The threading model..... 26
ConnectionFactories and Connection objects..... 26
Connection started and stopped mode..... 27
Connection closure..... 27
Exception Handling 27
Connection to a WebSphere service integration bus..... 28
Sessions 28
Transacted sessions..... 29
Message acknowledgement 29
Asynchronous message delivery 31
Synchronous message delivery..... 32
Message delivery mode 32
Destinations..... 32
Topic uniform resource identifiers 33
Queue uniform resource identifiers 35
Temporary destinations..... 36
Message producers..... 36
Message producers with no associated destination..... 36
Message producers with associated destination 37
Message consumers..... 37
Durable subscribers 37
Non-durable subscribers..... 38
Synchronous message consumers..... 39
Asynchronous message consumers..... 39
Poison messages..... 39
Queue browsers..... 41
Requestors 41
Object Deletion 42
XMS primitive types 42
Implicit conversion of a property value from one data type to another..... 43
Iterators..... 45
Coded character set identifiers..... 46
XMS error and exception codes 48
Building your own applications 48
Network stack selection mechanism 49
Automatic WMQ client reconnection through XMS 53
Connecting applications in a multiple installation environment 54

Chapter 6. Writing XMS applications in C 57

Object handles in C..... 57
Object Properties in C..... 58
C functions that return a string by value..... 58
C functions that return a byte array by value..... 59
C functions that return a string or byte array by reference..... 60
C functions that accept a string as input..... 61
Error handling in C..... 61
Return codes 61
The error block 61
Message and exception listener functions in C..... 62
Message listener functions in C..... 62
Exception listener functions in C..... 63

Chapter 7. Writing XMS applications in C++ 65

Namespaces in C++ 65
String objects in C++ 66
C++ methods that return a byte array..... 67
Properties in C++ 67
Assignment of XMS objects to variables in C++..... 67

Error handling in C++ 70
Message and exception listeners in C++ 72
Message listeners in C++ 72
Exception listeners in C++ 74
Use of C APIs in a C++ application 74

Chapter 8. Working with administered objects 77
Supported types of administered object repository 77
Property mapping for administered objects 78
Required properties for administered ConnectionFactory objects 78
Required properties for administered Destination objects 79
Creating administered objects 80
InitialContext objects 81
InitialContext properties 81
URI format for XMS initial contexts 81
JNDI Lookup Web service 83
Retrieval of administered objects 83

Chapter 9. Securing communications for XMS applications 85
Secure connections to a IBM MQ queue manager 85
CipherSuite and CipherSpec name mappings for connections to a WebSphere MQ queue manager 86
Secure connections to a WebSphere service integration bus messaging engine 88
CipherSuite and CipherSpec name mappings for connections to a WebSphere service integration bus 90

Chapter 10. XMS messages 91
Parts of an XMS message 91
Header fields in an XMS message 91
Properties of an XMS message 92
JMS-defined properties of a message 93
IBM-defined properties of a message 94
Application-defined properties of a message 95
The body of an XMS message 95
Data types for elements of application data 96
Bytes messages 97
Map messages 98
Object messages 98
Stream messages 99
Text messages 100
Message selectors 100
Mapping XMS messages onto IBM MQ messages 101
Using the XMS sample applications 102
The sample applications 102
Running the sample applications 104
Building the C or C++ sample applications 105

Chapter 11. Troubleshooting 107
Problem determination for C/C++ applications 107
Error conditions that can be handled at run time 107
Error conditions that cannot be handled at run time 108
Repeatable failures 108
FFDC and trace configuration for C/C++ applications 109
Tips for troubleshooting 111

Chapter 12. C classes 113
BytesMessage 114
Functions 114
Connection 127
Functions 127
ConnectionFactory for the C class 132
Functions 133
ConnectionMetaData 135
Functions 135
Destination for the C class 136
Functions 136
ErrorBlock 140
Functions 141
ExceptionListener 144
Functions 145
InitialContext 145
Functions 145
Iterator 147
Functions 147
MapMessage 149
Functions 149
Message 164
Functions 165
MessageConsumer 180
Functions 180
MessageListener 184
Functions 184
MessageProducer 185
Functions 185
ObjectMessage 194
Functions 194
Property 196
Functions 196
PropertyContext 211
Functions 211
QueueBrowser 228
Functions 228
Requestor 230
Functions 230
Session 232
Functions 232
StreamMessage 245
Functions 245
TextMessage 259
Functions 259

Chapter 13. Additional C functions 261
Process CCSID functions 261
Functions 261

Chapter 14. C++ classes 263
BytesMessage 265
Methods 265
Inherited methods 275
Connection 275

Methods ... 275
Inherited methods ... 280
ConnectionFactory for the C++ class ... 280
Constructors ... 280
Methods ... 281
Inherited methods ... 283
ConnectionMetaData ... 283
Methods ... 283
Inherited methods ... 285
Destination for the C++ class ... 285
Constructors ... 285
Methods ... 286
Inherited methods ... 288
Exception ... 288
Methods ... 289
ExceptionListener ... 292
Methods ... 292
IllegalStateException ... 293
Inherited methods ... 293
InitialContext ... 293
Constructors ... 293
Methods ... 294
Inherited methods ... 296
InvalidClientIDException ... 296
Inherited methods ... 296
InvalidDestinationException ... 296
Inherited methods ... 296
InvalidSelectorException ... 296
Inherited methods ... 296
Iterator ... 297
Methods ... 297
MapMessage ... 299
Methods ... 300
Inherited methods ... 310
Message ... 311
Methods ... 311
Inherited methods ... 323
MessageConsumer ... 323
Methods ... 323
Inherited methods ... 327
MessageEOFException ... 327
Inherited methods ... 328
MessageFormatException ... 328
Inherited methods ... 328
MessageListener ... 328
Methods ... 328
MessageNotReadableException ... 329
Inherited methods ... 329
MessageNotWritableException ... 329
Inherited methods ... 329
MessageProducer ... 329
Methods ... 329
Inherited methods ... 338
ObjectMessage ... 338
Methods ... 339
Inherited methods ... 340
Property ... 340
Constructors ... 341
Methods ... 342
PropertyContext ... 353
Methods ... 353
QueueBrowser ... 365
Methods ... 365
Inherited methods ... 367
Requestor ... 368
Constructors ... 368
Methods ... 368
Inherited methods ... 370
ResourceAllocationException ... 370
Inherited methods ... 370
SecurityException ... 371
Inherited methods ... 371
Session ... 371
Methods ... 371
Inherited methods ... 384
StreamMessage ... 384
Methods ... 384
Inherited methods ... 394
String ... 394
Constructors ... 394
Methods ... 396
TextMessage ... 398
Methods ... 398
Inherited methods ... 399
TransactionInProgressException ... 400
Inherited methods ... 400
TransactionRolledBackException ... 400
Inherited methods ... 400

Chapter 15. Properties of XMS objects 401

Properties of Connection ... 401
Properties of ConnectionFactory ... 402
Properties of ConnectionMetaData ... 406
Properties of Destination ... 407
Properties of InitialContext ... 408
Properties of Message ... 409
Properties of MessageConsumer ... 414
Properties of MessageProducer ... 414
Properties of Session ... 414
Property definitions ... 414
JMS_IBM_ArmCorrelator ... 417
JMS_IBM_CHARACTER_SET ... 417
JMS_IBM_ENCODING ... 418
JMS_IBM_EXCEPTIONMESSAGE ... 419
JMS_IBM_EXCEPTIONPROBLEMDESTINATION ... 419
JMS_IBM_EXCEPTIONREASON ... 419
JMS_IBM_EXCEPTIONTIMESTAMP ... 419
JMS_IBM_FEEDBACK ... 420
JMS_IBM_FORMAT ... 420
JMS_IBM_LAST_MSG_IN_GROUP ... 420
JMS_IBM_MSGTYPE ... 421
JMS_IBM_PUTAPPLTYPE ... 421
JMS_IBM_PUTDATE ... 421
JMS_IBM_PUTTIME ... 422
JMS_IBM_REPORT_COA ... 422
JMS_IBM_REPORT_COD ... 423
JMS_IBM_REPORT_DISCARD_MSG ... 423
JMS_IBM_REPORT_EXCEPTION ... 423
JMS_IBM_REPORT_EXPIRATION ... 424
JMS_IBM_REPORT_NAN ... 425
JMS_IBM_REPORT_PAN ... 425
JMS_IBM_REPORT_PASS_CORREL_ID ... 425

JMS_IBM_REPORT_PASS_MSG_ID 426
JMS_IBM_RETAIN 426
JMS_IBM_SYSTEM_MESSAGEID 427
JMS_TOG_ARM_Correlator 427
JMSX_APPID 427
JMSX_DELIVERY_COUNT 428
JMSX_GROUPID 428
JMSX_GROUPSEQ 428
JMSX_USERID 429
XMSC_ASYNC_EXCEPTIONS 429
XMSC_CLIENT_CCSID 429
XMSC_CLIENT_ID 430
XMSC_CONNECTION_TYPE 430
XMSC_DELIVERY_MODE 431
XMSC_IC_PROVIDER_URL 432
XMSC_IC_SECURITY_AUTHENTICATION 432
XMSC_IC_SECURITY_CREDENTIALS 432
XMSC_IC_SECURITY_PRINCIPAL 432
XMSC_IC_SECURITY_PROTOCOL 433
XMSC_IC_URL 433
XMSC_JMS_MAJOR_VERSION 433
XMSC_JMS_MINOR_VERSION 433
XMSC_JMS_VERSION 433
XMSC_MAJOR_VERSION 434
XMSC_MINOR_VERSION 434
XMSC_PASSWORD 434
XMSC_PRIORITY 434
XMSC_PROVIDER_NAME 435
XMSC_TIME_TO_LIVE 437
XMSC_USERID 437
XMSC_VERSION 437
XMSC_WMQ_CCSID 439
XMSC_WMQ_CHANNEL 440
XMSC_WMQ_CLIENT_RECONNECT_OPTIONS 440
XMSC_WMQ_CONNECTION_MODE 441
XMSC_WMQ_CONNECTION_NAME_LIST 442
XMSC_WMQ_DUR_SUBQ 442
XMSC_WMQ_ENCODING 443
XMSC_WMQ_FAIL_IF_QUIESCE 444
XMSC_WMQ_MESSAGE_BODY 444
XMSC_WMQ_MQMD_MESSAGE_CONTEXT 445
XMSC_WMQ_MQMD_READ_ENABLED 446
XMSC_WMQ_MQMD_WRITE_ENABLED 447
XMSC_WMQ_PUT_ASYNC_ALLOWED 447
XMSC_WMQ_READ_AHEAD_ALLOWED 448
XMSC_WMQ_READ_AHEAD_CLOSE_POLICY 448
XMSC_WMQ_RESOLVED_QUEUE_MANAGER 449
XMSC_WMQ_HOST_NAME 450
XMSC_WMQ_LOCAL_ADDRESS 450
XMSC_WMQ_MESSAGE_SELECTION 451
XMSC_WMQ_MSG_BATCH_SIZE 451
XMSC_WMQ_POLLING_INTERVAL 452
XMSC_WMQ_PORT 452
XMSC_WMQ_PROVIDER_VERSION 452
XMSC_WMQ_PUB_ACK_INTERVAL 454
XMSC_WMQ_QMGR_CCSID 454
XMSC_WMQ_QUEUE_MANAGER 454
XMSC_WMQ_RECEIVE_EXIT 455
XMSC_WMQ_RECEIVE_EXIT_INIT 455
XMSC_WMQ_SECURITY_EXIT 456
XMSC_WMQ_SECURITY_EXIT_INIT 456
XMSC_WMQ_SEND_EXIT 456
XMSC_WMQ_SEND_EXIT_INIT 457
XMSC_WMQ_SEND_CHECK_COUNT 457
XMSC_WMQ_SHARE_CONV_ALLOWED 457
XMSC_WMQ_SSL_CERT_STORES 458
XMSC_WMQ_SSL_CIPHER_SPEC 458
XMSC_WMQ_SSL_CIPHER_SUITE 460
XMSC_WMQ_SSL_ENCRYPTION_POLICY_SUITEB 461
XMSC_WMQ_SSL_CRYPTO_HW 461
XMSC_WMQ_SSL_FIPS_REQUIRED 462
XMSC_WMQ_SSL_KEY_REPOSITORY 462
XMSC_WMQ_SSL_KEY_RESETCOUNT 463
XMSC_WMQ_SSL_PEER_NAME 463
XMSC_WMQ_SYNCPOINT_ALL_GETS 463
XMSC_WMQ_TARGET_CLIENT 464
XMSC_WMQ_TEMP_Q_PREFIX 464
XMSC_WMQ_TEMP_TOPIC_PREFIX 465
XMSC_WMQ_TEMPORARY_MODEL 465
XMSC_WMQ_WILDCARD_FORMAT 465
XMSC_WPM_BUS_NAME 466
XMSC_WPM_CONNECTION_PROTOCOL 466
XMSC_WPM_CONNECTION_PROXIMITY 467
XMSC_WPM_DUR_SUB_HOME 467
XMSC_WPM_HOST_NAME 467
XMSC_WPM_LOCAL_ADDRESS 468
XMSC_WPM_ME_NAME 469
XMSC_WPM_NON_PERSISTENT_MAP 469
XMSC_WPM_PERSISTENT_MAP 469
XMSC_WPM_PORT 470
XMSC_WPM_PROVIDER_ENDPOINTS 470
XMSC_WPM_SSL_CIPHER_SUITE 471
XMSC_WPM_SSL_ENCRYPTION_POLICY_SUITEB 472
XMSC_WPM_SSL_KEY_REPOSITORY 473
XMSC_WPM_SSL_KEYRING_LABEL 473
XMSC_WPM_SSL_KEYRING_PW 473
XMSC_WPM_SSL_KEYRING_STASH_FILE 473
XMSC_WPM_SSL_FIPS_REQUIRED 474
XMSC_WPM_TARGET_GROUP 474
XMSC_WPM_TARGET_SIGNIFICANCE 474
XMSC_WPM_TARGET_TRANSPORT_CHAIN 475
XMSC_WPM_TARGET_TYPE 475
XMSC_WPM_TEMP_Q_PREFIX 476
XMSC_WPM_TEMP_TOPIC_PREFIX 476
XMSC_WPM_TOPIC_SPACE 476

Chapter 16. Best Practices 478
Recap on XMS Objects and its Relationship 480

Figures

1. XMS objects and their relationships 6
2. Typical use of administered objects by an XMS application 8
3. Connecting applications in a multiple installation environment 55

Chapter 1. Welcome to the documentation for Message Service Client for C/C++

This documentation is about IBM® Message Service Client for C/C++ Version 3.0.0. The documentation describes and documents the APIs provided by Message Service Client for C/C++. This API is referred to as XMS.

What's new in this release

This topic summarizes what is new in this release of Message Service Client for C/C++.

1) Message Service Client is available as a redistributable package in zip/tar format.
2) IBM MQ client 9.2.2 and above is required for running the XMS application.
3) Various APAR and defect fixes found in the field.
4) Compiler update on all the supported platforms.
5) Decommission support for Real-Time Transport.

Chapter 2. Introduction to Message Service Client for C/C++

This chapter describes Message Service Client for C/C++.

This chapter contains the following sections:

- "What is Message Service Client for C/C++?"
- "Styles of messaging" on page 4
- "The XMS object model" on page 5
- "The XMS message model" on page 8
- "Operating environments" on page 8
- "Prerequisites for XMS applications connecting to IBM MQ" on page 9

What is Message Service Client for C/C++?

Message Service Client for C/C++ provides an API called XMS that has the same set of interfaces as the Java Message Service (JMS) API. Message Service Client for C/C++ contains two implementations of XMS, one for use by C applications and another for use by C++ applications.

XMS supports both the point-to-point and the publish/subscribe styles of messaging, and supports both synchronous and asynchronous message delivery.

An XMS application can connect to, and use the resources of, any of the following messaging servers:

- A IBM MQ queue manager.

  The application can connect in either bindings or client mode.
- A WebSphere service integration bus.

  The application can use a direct TCP/IP connection, or it can use HTTP over TCP/IP.

An XMS application can exchange messages with any of the following types of application:

- An XMS application
- A IBM MQ JMS application
- A native IBM MQ application
- A JMS application that is using the WebSphere default messaging provider

Styles of messaging

XMS supports the *point-to-point* and *publish/subscribe* 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.

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.
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 it routes published messages to subscribers. A subscriber receives messages on 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 it 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.
The embedded publish/subscribe function also provides some additional

features such as retained publications and a choice of two wildcard schemes for specifying a range of topics to which an application wants to subscribe.

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 ConnectionFactory to create a connection. An application can create a ConnectionFactory object at run time, or it can create a ConnectionFactory object from an object definition that is retrieved from a repository of administered objects.

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

Destination
The source from where an application sends messages or receives messages. In the publish/subscribe domain, a Destination object encapsulates a topic and, in the point-to-point domain, a Destination object encapsulates a queue. An application can create a Destination object at run time, or it can create a Destination object from an object definition that is retrieved from a repository of administered objects.

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 (MessageProducer) or receives (MessageConsumer).

MessageProducer
An object used by an application to send messages to a destination.

MessageConsumer
An object used by an application to receive messages sent to a destination.

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

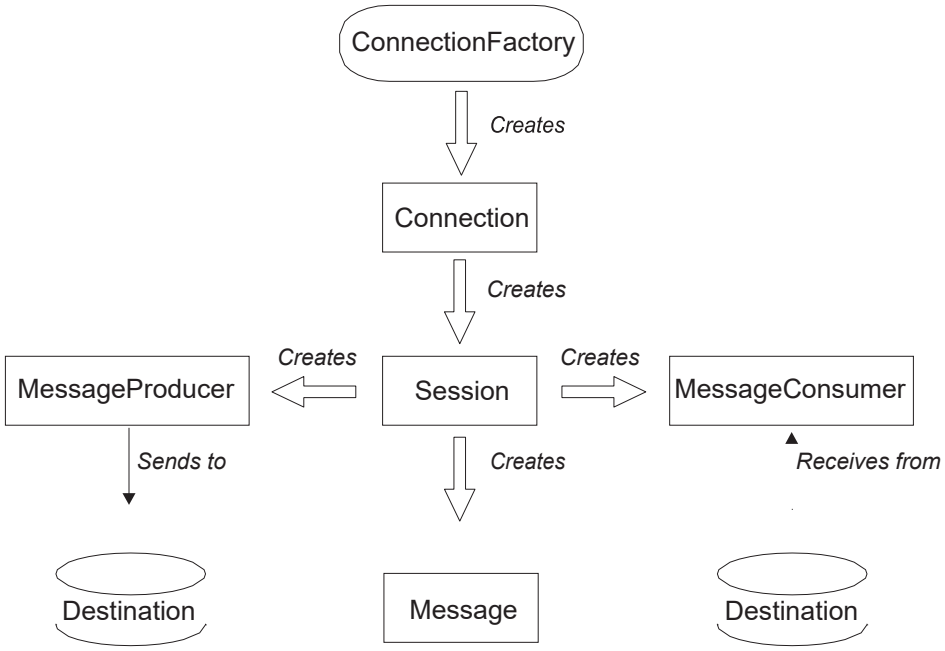


Figure 1. XMS objects and their relationships

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 calls a function to create 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 access the object. For example, if a C application creates a ConnectionFactory, XMS returns a handle for the ConnectionFactory 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 on the domain independent interfaces that are described in *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, that are implemented in different ways.

Attributes

An object characteristic that is always present and occupies storage, even if the attribute does not have a value. In this respect, an attribute is similar to a field in a fixed length data structure. A distinguishing feature of attributes 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 (nor can the storage be recovered) after its value has been set, although you can change its value. XMS provides a set of generic methods for setting and getting property values.

Related concepts:

"XMS primitive types" on page 42: A property of an object is present and

occupies storage only after its value is set. However, a property cannot be deleted (nor can the storage be recovered) after its value has been set, although you can change its value. XMS provides a set of generic methods for setting and getting property values.

"Implicit conversion of a property value from one data type to another" on page 43: When an application gets the value of a property, the value can be converted by XMS into another data type. Many rules govern which conversions are supported and how XMS performs the conversions.

Related reference:

"Data types for elements of application data" on page 96: To ensure that an XMS application can exchange messages with a IBM MQ JMS application, both the applications must be able to interpret the application data in the body of a message in the same way

Administered objects

Using administered objects, you can administer the connection settings used by client applications to be administered from a central repository. An application retrieves object definitions from the central repository and uses them to create ConnectionFactory and Destination objects. This allows applications to be de-coupled from the resources that they use at runtime.

For example, XMS applications can be written and tested with administered objects that reference a set of connections and destinations in a test environment. When the applications are deployed, the administered objects can be changed to point the applications to a production environment.

XMS supports two types of administered object:

- A ConnectionFactory object, which is used by applications to make the initial connection to the server
- A Destination object, which is used by applications to specify the destination for messages that are being sent, and the source of messages that are being received. A destination is either a topic or a queue on the server to which an application connects.

The IBM MQ JMS administration tool (JMSAdmin) available with IBM MQ can be used to create and manage administered objects for IBM MQ in a central repository of administered objects.

The administered objects in the repository can be used by IBM MQ JMS applications, and also by XMS applications for ConnectionFactories and Destinations for IBM MQ queue manager. An administrator can change the object definitions held in the repository without affecting application code.

The following diagram shows how an XMS application typically uses administered objects.

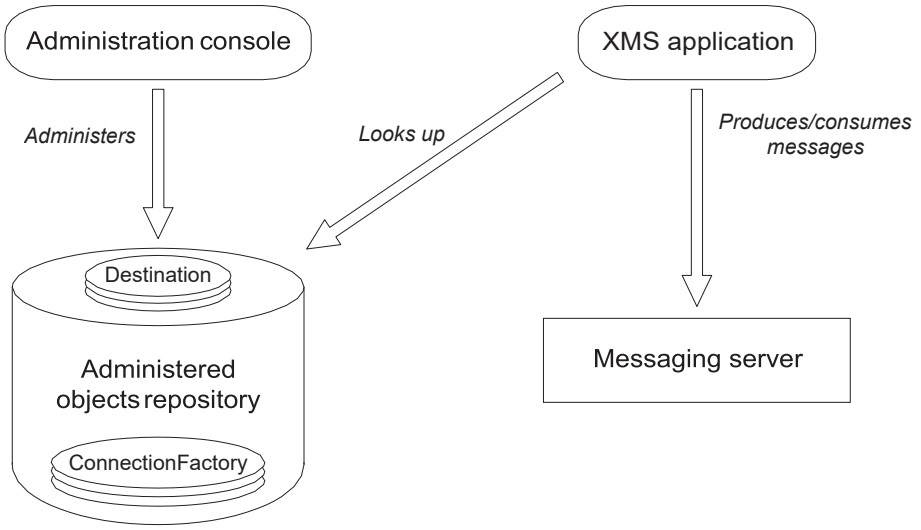


Figure 2. Typical use of administered objects by an XMS application

The XMS message model

The XMS message model is the same as the WebSphere JMS message model.

In particular, XMS implements the same message header fields and message properties that WebSphere 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 applications can exchange messages with WebSphere JMS applications. For each message sent by an XMS or WebSphere JMS application, some of the header fields and properties are set by the application, others are set by XMS or WebSphere JMS when the message is sent, and the remainder are set by XMS or WebSphere JMS when the message is received. Where appropriate, these header fields and properties are propagated with a message through a messaging server and are made available to any application that receives the message.

Operating environments

An XMS client is supplied for each of the tested operating systems.

Table 1 lists the compiler for each client platform.

Table 1. Message Service Client for C/C++ platforms and compilers

Operating system	Compiler
x64 bit Microsoft Windows 10 Pro Microsoft Windows Server 2019 Standard Edition, Microsoft Windows Server 2016 Standard Edition	Microsoft Visual Studio 2019

Table 1. Message Service Client for C/C++ platforms and compilers (continued)

Operating system	Compiler
32 bit: Microsoft Windows Server 2019 Standard Edition , Microsoft Windows Server 2016 Standard Edition	Microsoft Visual Studio 2019
Red Hat Enterprise Linux 7.6 on x86-64	gcc 4.8
AIX® 7.1 AIX 7.2	IBM® XL C/C++ Enterprise Edition for AIX 16.0 (The minimum level of IBM XL C/C++ Enterprise Edition for AIX 16.0 is 16.1.0)

Prerequisites for XMS applications

Some prerequisites apply if your XMS application connects to IBM MQ or WPM.

For XMS applications that connect to a IBM MQ queue manager or WPM connectivity, you must install the IBM MQ client libraries on the machine you use to run the XMS application. You can have the installed version of the IBM MQ client or Redistributable IBM MQ clients on the system. Refer IBM documentation for IBM MQ client installation.

Related tasks:

The chapter Chapter 4, "Setting up the messaging server environment," on page 21 describes how to set up the messaging server environment to allow XMS applications to connect to a server.

Prerequisites for XMS applications using LDAP Server for JNDI objects

Windows Only: IBM Directory Server Client v6.4 must be installed if applicationsuse a LDAP server to store JNDI objects. The PATH environment variable must include the path of IBM Directory Server client libraries. XMS will load relevant libraries from the specified path.

Chapter 3. Installing Message Service Client for C/C++

This chapter describes how to install Message Service Client for C/C++ (XMS).

About this task

Follow these instructions to install or uninstall Message Service Client for C/C++.

This chapter contains the following section:

v "Installing Message Service Client for C/C++"

Installing Message Service Client for C/C++

This chapter describes how to install Message Service Client for C/C++ (XMS). Follow these instructions to install or uninstall Message Service Client for C/C++.

About this task

Redistributable Message Service Client for C/C++ is a collection of runtime files that are provided in a .zip or .tar file that can be redistributed to third parties under redistributable license terms, which provides a simple way of distributing applications and the runtime files that they require in a single package.

The following sections describe how to install and uninstall Message Service Client for C/C++ in more detail:

v "Installing Message Service Client for C/C++"

v "What is installed on AIX and Linux x86-64" on page 16

v "What is installed on Windows (C/C++)" on page 17

v "Uninstalling Message Service Client for C/C++" on page 18

Installing Message Service Client for C/C++

Before you begin

The IBM IPLA license agreement is extended for IBM MQ to enable you to download a number of additional runtime files from Fix Central.

IBM MQ client is the pre-req for Message Service Client for C/C++. You can have the installed version of the IBM MQ client or Redistributable IBM MQ clients on the system. Refer the MQ documentation for installation of the IBM MQ client.

The directory where XMS is extracted from archive or unzipped should have enough privilege for application user and able to communicate MQ Client.

About this task

To install Message Service Client for C/C++ on AIX, Linux, or Windows, follow this procedure.

Procedure

1. For Message Service Client for C/C++ v3.0.0.0, images are available under the following file names:
   a. Linux x86-64: 3.0.0.0-IBM-IA94-Redist-LinuxX64.tar.gz.
   b. AIX: 3.0.0.0-IBM-IA94-Redist-AIX.tar.gz.
   c. Windows: 3.0.0.0-IBM-IA94-Redist-Win64.zip.

2. Extract the package to the required directory
   a. On AIX, Linux, log in as user who has enough privilege to write in the file system and communicate with IBM MQ Client .
   b. On AIX, Linux create a directory and extract the contents of the tar into the directory. Example for Linux: -

      $ gunzip 3.0.0.0-IBM-IA94-Redist-LinuxX64.tar.gz

      $ tar -xvf 3.0.0.0-IBM-IA94-Redist-LinuxX64.tar
   c. Windows, Unzip the XMS package using Windows utilities

Results

You have now successfully installed Message Service Client for C/C++, which is ready to use.

What to do next

Before running any XMS applications, including the sample applications provided with XMS, you must set up the messaging server environment as described in Chapter 4, "Setting up the messaging server environment,". Set up the IBM MQ client and Message Service Client as described in "Using the XMS sample applications" .

Related concepts:

Refer to "JNDI Lookup Web service" on page 83 to access a COS naming directory from XMS, a JNDI Lookup Web service must be deployed on a WebSphere service integration bus server. This Web service translates the Java information from the COS naming service into a form that XMS applications can read.

The chapter Chapter 4, "Setting up the messaging server environment," describes how to set up the messaging server environment to allow XMS applications to connect to a server.

The chapter "Using the XMS sample applications" provides information about how to use the sample applications provided with XMS.

What is installed on AIX and Linux

Table 2 lists the installed directories, relative to the installation directory, and describes their contents.

Table 2. Installed directories on AIX and Linux and their contents

Installed feature	Installed directory	Contents
Runtime	bin	Programs, for example, gxitrcfmt and gxisc
	lib	The shared object libraries required to compile and run XMS applications
	lib64	The shared object libraries required to compile and run XMS applications on 64 bit platforms **Note:** This directory will only appear on currently supported 64 bit platforms.
	license	license agreement files.
	ReadMe.Redist	ReadMe for the product
Development Tools	tools/c/include	The XMS header files for C
	tools/cpp/include	The XMS header files for C++

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

Installed feature	Installed directory	Contents
	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/ SampleConsumerC	The source and makefile for the C message consumer sample application
	tools/samples/ SampleProducerC	The source and makefile for the C message producer sample application
	tools/samples/ SampleConsumerCPP	The source and makefile for the C++ message consumer sample application
	tools/samples/ SampleProducerCPP	The source and makefile for the C++ message producer sample application
	tools/samples/ SampleConfigC	The sampleconfig tool
	tools/samples/ simple	Simple sample for connecting to IBM MQ and WPM.

What is installed on Windows (C/C++)

On Windows x86, XMS is installed in the C:\Program Files\IBM\XMS directory and on Windows x64, XMS is installed in the C:\Program Files (x86) \IBM\XMS 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 feature	Installed directory	Contents
Runtime	bin	The *.dll and *.pdb files required to run XMS applications. Programs, for example, gxitrcfmt and gxisc
	bin64	The *.dlls required to run 64bit XMS aplications
	licenses	The licenses for Message Service Client for C/C++.
	ReadMe.Redist	ReadMe file of the product.
Development Tools	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\lib64	On x64 only, 64bit XMS link libraries for C and C++

Table 3. (continued)

Installed feature	Installed directory	Contents
Samples	tools\samples\bin	The compiled sample applications and the command file to run them
	tools\samples\ SampleConsumerC	The source and makefile for the C message consumer sample application
	tools\samples\ SampleProducerC	The source and makefile for the C message producer sample application
	tools\samples\ SampleConsumerCPP	The source and makefile for the C++ message consumer sample application
	tools\samples\ SampleProducerCPP	The source and makefile for the C++ message producer sample application
	tools\samples\c\ sampleconfig	The sampleconfig tool
	tools\samples\ readme.txt	The readme.txt file for the sample applications
	tools\samples\simple	Simple sample for connecting to IBM MQ and WPM.

Note: If you do not have Visual C++ 2019 installed on the system, you need to install the Microsoft Visual C++ 2019 Redistributable Package. This package has the runtime components of Visual C++ Libraries that are required to run applications developed with Visual C++ 2019 on a computer that does not have Visual C++ 2019 installed. To download this package, refer to the Microsoft Website for more information.

Uninstalling Message Service Client for C/C++

When the level of Message Service Client for C/C++ is not required any more you can uninstall by deleting the directory and its content..

About this task

When the level of Message Service Client for C/C++ is not required any more please follow the below procedure before deleting the directory.

Procedure

1. End all the Message Service Client application.
2. Close all the prompts where the application is run.
3. Delete the folder and its contents where Message Service Client is extracted.

Results

You have now successfully removed the Message Service Client for C/C++ from your system.

Chapter 4. Setting up the messaging server environment

This chapter describes how to set up the messaging server environment to allow XMS applications to connect to a server.

Before you begin

The following prerequisite applies to setting up the messaging server environment:

v For applications that connect to a IBM MQ queue manager, the IBM MQ client (or queue manager for bindings mode) is required.

For additional information about prerequisites, refer to the readme.txt file for Message Service Client for C/C++.

About this task

You must set up the messaging server environment before running any XMS applications, including the sample applications provided with XMS.

This chapter contains the following sections:

v "Configuring the queue manager for an application that connects to a IBM MQ queue manager"

v "Configuring the service integration bus for an application that connects to a WebSphere service integration bus" on page 24

Configuring the queue manager for an application that connects to a IBM MQ queue manager

This section assumes that you are using IBM MQ version 7.0. Before you can run an application that connects to a IBM MQ queue manager, you must configure the queue manager.

Before you begin

Before starting this task, you must do the following:

v Make sure that your application has access to a queue manager that is running.

v Make sure that your application uses a connection factory whose properties are set appropriately to connect to the queue manager.. For more information about the properties of a connection factory, "Properties of ConnectionFactory" on page 402.

About this task

Any queue or topic object that application require must be defined. By default SYSTEM.DEFAULT.TOPIC will be there when queue manager is created, if application require further topic or queue, you must create these yourself. The following steps summarize what you need to do:

Procedure

1. On the queue manager, create the queues or topics that your application needs. For information about how to do this, see the *IBM MQ Documentation*.
2. Grant the user ID associated with your application the authority to connect to

the queue manager and the appropriate authorities to access the queues.

For information about how to do this, see the *IBM MQ Documentation*. If your application connects to the queue manager in client mode, see also *IBM MQ documentation* or *IBM MQ documentation*.

Results

You can now start your application.

Configuring the service integration bus for an application that connects to a WebSphere service integration bus

Before you can run an application that connects to a WebSphere service integration bus, you must configure the service integration bus in the same way that you configure the service integration bus to run JMS applications that use the default messaging provider.

Before you begin

Before starting this task, you must do the following:

- Make sure that a messaging bus has been created and that your server has been added to the bus as a bus member.
- Make sure that your application has access to a service integration bus that contains at least one messaging engine that is running.
- If HTTP operation, is required then an HTTP messaging engine inbound transport channel must be defined. By default, channels for SSL and TCP will already have been predefined during the server installation.
- Make sure that your application uses a connection factory whose properties are set appropriately to connect to the service integration bus using a bootstrap server. The minimum information that you need to specify is:
  - The provider endpoint, which describes the location and protocol to use when negotiating a connection to the messaging server (that is, via the bootstrap server). In its simplest form, for a server installed with default settings, this can be set to the hostname of the server.
  - The name of the bus through which messages should be sent.

  For more information about the properties of a connection factory, see "Properties of ConnectionFactory" on page 402.

About this task

Any queue or topic spaces that you require must be defined. By default a topic space called Default.Topic.Space will already have been predefined during the server installation but, if you require further topic spaces, you must create these yourself. You do not need to predefine individual topics within a topic space, since the server instantiates these dynamically as required.

The following steps summarize what you need to do but, for more details, see the WebSphere Application Server Information Center.

Procedure

1. Create the queues that your application needs for point-to-point messaging.
2. Create any additional topic spaces that your application needs for publish/subscribe messaging.

Results

You can now start your application.

Chapter 5. Developing XMS applications

This chapter provides information that you might find useful when writing XMS applications.

About this task

The information in this chapter applies to C, and C++ applications.

For information about writing XMS applications, refer to the following topics:

Writing XMS applications

About this task

The information in this chapter applies to C and C++ applications.

If you are writing applications in C, see also Chapter 6, "Writing XMS applications in C," on page 57. If you are writing applications in C++, see also Chapter 7, "Writing XMS applications in C++," on page 65.

This chapter contains the following sections:

v "The threading model" on page 26
v "ConnectionFactories and Connection objects" on page 26
v "Sessions" on page 28
v "Destinations" on page 32
v "Message producers" on page 36
v "Message consumers" on page 37 v
"Queue browsers" on page 41
v "Requestors" on page 41
v "Object Deletion" on page 42
v "XMS primitive types" on page 42
v "Implicit conversion of a property value from one data type to another" on page 43
v "Iterators" on page 45
v "Coded character set identifiers" on page 46
v "XMS error and exception codes" on page 48
v "Building your own applications" on page 48
v "Network stack selection mechanism" on page 49

Related tasks:

The chapter Chapter 6, "Writing XMS applications in C," on page 57 provides information help you write XMS applications in C.

The chapter Chapter 7, "Writing XMS applications in C++," on page 65 provides information to help you when writing XMS applications in C++.

The threading model

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 on only a single thread at any one time.

Exceptions to these rules are indicated by entries labelled "Thread context" in the interface definitions of the methods in the API reference chapters Reference.

ConnectionFactories and Connection objects

A ConnectionFactory object provides a template that an application uses to create a Connection object. The application uses the Connection object to create a Session object.

For C and C++ applications a single type of ConnectionFactory has a property that enables you to select which type of protocol you want to use for a connection.

An XMS application can create multiple connections, and a multithreaded application can use a single Connection object concurrently on multiple threads. A Connection object encapsulates a communications connection between an application and a messaging server.

A connection serves several purposes:

- When an application creates a connection, the application can be authenticated.
- An application can associate a unique client identifier with a connection. The client identifier is used to support durable subscriptions in the publish/subscribe domain. The client identifier can be set in two ways:

  The preferred way of assigning a connection's client identifier is to configure in a client-specific ConnectionFactory object using properties and transparently assign it to the connection it creates.

  An alternative way of assigning a client identifier is to use a provider-specific value that is set on the Connection object. This value does not override the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If an administratively specified identifier does exist, an attempt to override it with a provider-specific value causes an exception to be thrown. If an application explicitly sets an identifier, it must do this immediately after creating the connection and before any other action on the connection is taken; otherwise, an exception is thrown.
- 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.

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.

Connection started and stopped mode

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 initialize sessions, and it can send messages but cannot receive them, either synchronously or asynchronously.

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.

Connection closure

An application closes a connection by calling the Close Connection method.

When an application closes a connection, XMS performs the following actions:

- It 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 "Object Deletion" on page 42. At the same time, XMS rolls back any transactions currently in progress within the sessions.
- It ends the communications connection with the messaging server.
- It releases the memory and other internal resources used by the connection.

XMS does not acknowledge the receipt of any messages that it has failed to acknowledge during a session, prior to closing the connection. For more information about acknowledging the receipt of messages, see "Message acknowledgement" on page 29.

Exception Handling

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 when 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 it learns about a problem with the connection only by using an exception listener.

For more information about using exception listener functions in a C application, see "Exception listener functions in C" on page 63. If you are using C++, see "Exception listeners in C++" on page 74.

Connection to a WebSphere service integration bus

An XMS application can connect to a WebSphere service integration bus either by using a direct TCP/IP connection or by using HTTP over TCP/IP.

The HTTP protocol can be used in situations where a direct TCP/IP connection is not possible. One common situation is when communicating through a firewall, such as when two enterprises exchange messages. Using HTTP to communicate through a firewall is often referred to as *HTTP tunnelling*. HTTP tunnelling, however, is inherently slower than using a direct TCP/IP connection because HTTP headers add significantly to the amount of data that is transferred, and because the HTTP protocol requires more communication flows than TCP/IP.

To create a TCP/IP connection, an application can use a connection factory whose XMSC_WPM_TARGET_TRANSPORT_CHAIN property is set to XMSC_WPM_TARGET_TRANSPORT_CHAIN_BASIC. This is the default value of the property. If the connection is created successfully, the XMSC_WPM_CONNECTION_PROTOCOL property of the connection is set to XMSC_WPM_CP_TCP.

To create a connection that uses HTTP, an application must use a connection factory whose XMSC_WPM_TARGET_TRANSPORT_CHAIN property is set to the name of an inbound transport chain that is configured to use an HTTP transport channel. If the connection is created successfully, the XMSC_WPM_CONNECTION_PROTOCOL property of the connection is set to XMSC_WPM_CP_HTTP. For information about how to configure transport chains, see the WebSphere Application Server Version 6.0x Information Center.

An application has a similar choice of communication protocols when connecting to a bootstrap server. The XMSC_WPM_PROVIDER_ENDPOINTS property of a connection factory is a sequence of one or more endpoint addresses of bootstrap servers. The bootstrap transport chain component of each endpoint address can be either XMSC_WPM_BOOTSTRAP_TCP, for a TCP/IP connection to a bootstrap server or XMSC_WPM_BOOTSTRAP_HTTP, for a connection that uses HTTP.

Sessions

A session is a single threaded context for sending and receiving messages.

An application can use a session to create messages, message producers, message consumers, queue browsers, and temporary destinations. An application can also use a session to run local transactions.

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, each receives a copy of any message published on that topic.

Unlike a Connection object, a Session object cannot be used concurrently on different threads. Only the Close Session method of a Session object can be called from a thread other than the one that the Session object is using at the time. The Close Session method ends a session and releases any system resources allocated to the session.

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

Transacted sessions

XMS applications can run local transactions. A *local transaction* is a transaction that involves changes only to the resources of the queue manager or service integration bus to which the application is connected.

The information in this section is relevant only if an application connects to a IBM MQ queue manager or a WebSphere service integration bus..

To run local transactions, an application must first create a transacted session by calling the Create Session method of a Connection object, specifying as a parameter that the session is transacted. Subsequently, all messages sent and received within the session are grouped into a sequence of transactions. A transaction ends when the application commits or rolls back the messages it has sent and received since the transaction began.

To commit a transaction, an application calls the Commit method of the Session object. When a transaction is committed, all messages sent within the transaction become available for delivery to other applications, and all messages received within the transaction are acknowledged so that the messaging server does not attempt to deliver them to the application again. In the point-to-point domain, the messaging server also removes the received messages from their queues.

To roll back a transaction, an application calls the Rollback method of the Session object. When a transaction is rolled back, all messages sent within the transaction are discarded by the messaging server, and all messages received within the transaction become available for delivery again. In the point-to-point domain, the messages that were received are put back on their queues and become visible to other applications again.

A new transaction starts automatically when an application creates a transacted session or calls the Commit or Rollback method. Therefore, a transacted session always has an active transaction.

When an application closes a transacted session, an implicit rollback occurs. When an application closes a connection, an implicit rollback occurs for all the connection's transacted sessions.

A transaction is wholly contained within a transacted session. A transaction cannot span sessions. This means that it is not possible for an application to send and receive messages in two or more transacted sessions and then commit or roll back all these actions as a single transaction.

Message acknowledgement

Every session that is not transacted has an acknowledgement mode that determines how messages received by the application are acknowledged. Three acknowledgement modes are available, and the choice of acknowledgement mode affects the design of the application.

The information in this section is relevant only if an application connects to a IBM MQ queue manager or a WebSphere service integration bus..

XMS uses the same mechanism for acknowledging the receipt of messages that JMS uses.

If a session is not transacted, the way that messages received by the application are acknowledged is determined by the acknowledgement mode of the session. The three acknowledgement modes are described in the following paragraphs:

XMSC_AUTO_ACKNOWLEDGE

The session automatically acknowledges each message received by the application.

If messages are delivered synchronously to the application, the session acknowledges receipt of a message every time a Receive call completes successfully. If messages are delivered asynchronously to a C application, the session acknowledges receipt of a message every time a call to a message listener function completes successfully. For a C++ application, the session acknowledges receipt of a message every time a call to the onMessage() method of a message listener completes successfully.

If the application receives a message successfully, but a failure prevents acknowledgement from occurring, the message becomes available for delivery again. The application must therefore be able to handle a message that is re-delivered.

XMSC_DUPS_OK_ACKNOWLEDGE

The session acknowledges the messages received by the application at times it selects.

Using this acknowledgement mode reduces the amount of work the session must do, but a failure that prevents message acknowledgement might result in more than one message becoming available for delivery again. The application must therefore be able to handle messages that are re-delivered.

Restriction: In AUTO_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE modes, XMS C/C++ does not support an application throwing an unhandled exception in a message listener. This means that messages are always acknowledged when the message listener returns, regardless of whether it was processed successfully (provided any failures are non-fatal and do not prevent the application from continuing). If you require finer control of message acknowledgement, use the CLIENT_ACKNOWLEDGE or transacted modes, which give the application full control of the acknowledgement functions.

XMSC_CLIENT_ACKNOWLEDGE

The application acknowledges the messages it receives by calling the Acknowledge method of the Message class.

The application can acknowledge the receipt of each message individually, or it can receive a batch of messages and call the Acknowledge method only for the last message it receives. When the Acknowledge method is called all messages received since the last time the method was called are acknowledged.

In conjunction with any of these acknowledgement modes, an application can stop and restart the delivery of messages in a session by calling the Recover method of the Session class. Messages whose receipt was previously unacknowledged are re-delivered. However, they might not be delivered in the same sequence in which they were previously delivered. In the meantime, higher priority messages might have arrived, and some of the original messages might have expired. In the point-to-point domain, some of the original messages might have been consumed by another application.

An application can determine whether a message is being re-delivered by examining the contents of the JMSRedelivered header field of the message. The application does this by calling the Get JMSRedelivered method of the Message class.

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 requires 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.

IBM MQ also supports asynchronous message consumption. An application can register a callback function for a destination. When a suitable message is sent to the destination, IBM MQ calls the function and passes the message as a parameter. The function then processes the message asynchronously. In previous releases of IBM MQ, this feature was available only when using IBM MQ classes for JMS and Message Service Client for C/C++.

Message Service Client for C/C++ has been changed to use this new feature in IBM MQ V7.1. The implementation of XMS message listeners is now a more natural fit with WebSphere MQ. Message Service Client for C/C++ no longer has to find a destination to check whether a suitable message has been sent to the destination. The performance of XMS message listeners is improved as a result, particularly when an application uses multiple message listeners in a session to

monitor multiple destinations. Message throughput is increased, and the time taken to deliver a message to a message listener after it has arrived at a destination is reduced.

For more information about using message listener functions in a C application, see "Message listener functions in C" on page 62. If you are using C++, see "Message listeners in C++" on page 72.

Synchronous message delivery

Messages are delivered synchronously to an application if the application uses the Receive methods of MessageConsumer objects.

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.

Message delivery mode

XMS supports two modes of message delivery.

- *Persistent* messages are delivered once and once only. A messaging server takes special precautions, such as logging the messages, to ensure that persistent messages are not lost in transit, even in the event of a failure.
- *Nonpersistent* messages are delivered no more than once. Nonpersistent messages are less reliable than persistent messages because they can be lost in transit in the event of a failure.

The choice of delivery mode is a trade-off between reliability and performance. Nonpersistent messages are typically transported more quickly than persistent messages.

Destinations

An XMS application uses a Destination object to specify the destination of messages that are being sent, and the source of messages that are being received.

An XMS application can either create a Destination object at run time, or obtain a predefined destination from the repository of administered objects.

As with a ConnectionFactory, the most flexible way for an XMS application to specify a destination is to define it as an administered object. Using this approach, applications written in C, C++ languages, as well as Java, can share the same definition of the destination. Using this approach, applications written in C and C++ languages, as well as Java, can share the same definition of the destination.. The properties of administered Destination objects can be changed without changing any code.

You can create a destination for a C or C++ application in either of the following ways:

- By specifying a *uniform resource identifier (URI)*, which is a string that identifies a destination, you have the option to specify one or more properties of the destination
- By specifying whether you require a queue or topic and providing a destination name

For further information, see “Destination for the C class” on page 136 for C or “Destination for the C++ class” on page 285 for C++.

For further information about creating a URI, see “Topic uniform resource identifiers” and “Queue uniform resource identifiers” on page 35.

Topic uniform resource identifiers

The topic uniform resource identifier (URI) specifies the name of the topic; it can also specify one or more properties for it.

The URI for a topic begins with the sequence topic://, followed by the name of the topic and (optional) a list of name-value pairs that set the remaining topic properties. A topic name cannot be empty.

Here is an example in a fragment of C++ code:

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

For more information about the properties of a topic, including the name and valid values that you can use in a URI, see “Properties of Destination” on page 407.

When specifying a topic URI for use in a subscription, wildcards can be used. The syntax for these wildcards depends on the connection type; the following options are available:

- IBM MQ queue manager with Character level wildcard format
- IBM MQ queue manager with Topic level wild card format
- WebSphere service integration bus

IBM MQ queue manager with Character level wildcard format

IBM MQ queue manager with Character level wildcard format uses the following wild card characters:

* for 0 or more characters

? for 1 character

% for an escape character

Table 4 gives some examples of how to use this wildcard scheme.

Table 4. Example URIs using character level wildcard scheme for IBM MQ queue manager

Uniform Resource Identifier	Matches	Examples
"topic://Sport*Results"	All topics starting with "Sport" and ending in "Results"	"topic://SportsResults" and "topic://Sport/Hockey/National/Div3/Results"
"topic://Sport?Results"	All topics starting with "Sport" followed by a single character, followed by "Results"	"topic://SportsResults" and "topic://SportXResults"
"topic://Sport/*ball*/Div?/Results/*/???"	Topics	"topic://Sport/Football/Div1/Results/2002/Nov" and "topic://Sport/Netball/National/Div3/Results/02/Jan"

IBM MQ queue manager with Topic level wild card format

IBM MQ queue manager with Topic level wild card format uses the following wildcard characters:

to match multiple levels

+ to match a single level

Table 5 gives some examples of how to use this wildcard scheme.

Table 5. Example URIs using topic level wildcard scheme for IBM MQ queue manager

Uniform Resource Identifier	**Matches**	**Examples**
"topic://Sport/+/Results"	All topics with a single hierarchical level name between Sport and Results	"topic://Sport/Football/Results" and "topic://Sport/Ju-Jitsu/Results"
"topic://Sport/#/Results"	All topics starting with "Sport/" and ending in "/Results"	"topic://Sport/Football/Results" and "topic://Sport/Hockey/National/Div3/Results"
"topic://Sport/Football/#"	All topics starting with "Sport/Football/"	"topic://Sport/Football/Results" and "topic://Sport/Football/TeamNews/Signings/Managerial"

WebSphere service integration bus

IBM MQ with uses the following wildcard characters:

* to match any characters at one level in the hierarchy

// to match 0 or more levels

//. to match 0 or more levels (at the end of a Topic expression)

Table 6 on page 35 gives some examples of how to use this wildcard scheme.

Table 6. Example URIs using wildcard scheme for WebSphere service integration bus

Uniform Resource Identifier	Matches	Examples
"topic://Sport/*ball/Results"	All topics with a single hierarchical level name ending in "ball" between Sport and Results	"topic://Sport/Football/Results" and "topic://Sport/Netball/Results"
"topic://Sport//Results"	All topics starting with "Sport/" and ending in "/Results"	"topic://Sport/Football/Results" and "topic://Sport/Hockey/National/Div3/Results"
"topic://Sport/Football//."	All topics starting with "Sport/Football/"	"topic://Sport/Football/Results" and "topic://Sport/Football/TeamNews/Signings/Managerial"
"topic://Sport/*ball//Results//."	Topics	"topic://Sport/Football/Results" and "topic://Sport/Netball/National/Div3/Results/2002/November"

Queue uniform resource identifiers

The URI for a queue specifies the name of the queue; it can also specify one or more properties of the queue.

The URI for a queue begins with the sequence queue://, followed by the name of the queue; it might also include a list of name-value pairs that set the remaining queue properties.

For IBM MQ queues (but not for WebSphere Application Server default messaging provider queues), the queue manager on which the queue resides may be specified before the queue, with a / separating the queue manager name from the queue name.

If a queue manager is specified, then it must be the one to which XMS is directly connected for the connection using this queue, or it must be accessible from this queue. Remote queue managers are only supported for retrieving messages from queues, not for putting messages onto queues. For full details, refer to the IBM MQ queue manager documentation.

If no queue manager is specified, then the extra / separator is optional, and its presence or absence makes no difference to the definition of the queue.

The following queue definitions are all equivalent for a IBM MQ queue called QB on a queue manager called QM_A, to which XMS is directly connected:

```
queue://QB
queue:///QB
queue://QM_A/QB
```

The following is an example of queue definitions for C++:

```
ioQueue = session.createQueue("queue:///SYSTEM.DEFAULT.LOCAL.QUEUE");
```

The name of the queue manager is omitted. This is interpreted as the queue manager to which the owning connection is connected at the time when the Queue object is used.

The following example of C code connects to queue Q1 on queue manager HOST1.QM1, and causes all messages to be sent as nonpersistent and priority 5 messages:

```
rc = xmsDestCreate(
  "queue://HOST1.QM1/Q1?persistence=1&priority=5",
  &ioQueue);
```

Temporary destinations

XMS applications can create and use temporary destinations.

An application typically uses a temporary destination to receive replies to request messages. To specify the destination where a reply to a request message is to be sent, an application calls the Set JMSReplyTo method of the Message object representing the request message. The destination specified on the call can be a temporary destination.

To create a temporary destination, a C application calls the xmsDestCreateTemporaryByType() function. As parameters on the call, the application specifies the handle for the session in which the temporary destination is being created and the type of temporary destination, which is either a queue or a topic.

A C++ application creates a temporary queue by calling the createTemporaryQueue() method of a Session object, and it creates a temporary topic by calling the createTemporaryTopic() method of a Session object.

Although a session is used to create a temporary destination, the scope of a temporary destination is actually the connection that was used to create the session. Any of the connection's sessions can create message producers and message consumers for the temporary destination. The temporary destination remains until it is explicitly deleted, or the connection ends, whichever happens first.

When an application creates a temporary queue, a queue is created in the messaging server to which the application is connected. If the application is connected to a queue manager, a dynamic queue is created from the model queue whose name is specified by the XMSC_WMQ_TEMPORARY_MODEL property, and the prefix that is used to form the name of the dynamic queue is specified by the XMSC_WMQ_TEMP_Q_PREFIX property. If the application is connected to a service integration bus, a temporary queue is created in the bus, and the prefix that is used to form the name of the temporary queue is specified by the XMSC_WPM_TEMP_Q_PREFIX property.

When an application that is connected to a service integration bus creates a temporary topic, the prefix that is used to form the name of the temporary topic is specified by the XMSC_WPM_TEMP_TOPIC_PREFIX property.

Message producers

In XMS, a message producer can be created either with a valid destination or with no associated destination. When creating a message producer with a null destination, a valid destination needs to be specified when sending a message.

Message producers with no associated destination

In the C and C++ API, a message producer can be created with a null destination.

In the C API, NULL can be passed into the xmsSessCreateProducer() function, to create a message producer with no associated destination. In this case, the

destination must be specified when the message is sent. For further details about creating a message producer in a C API, see “Session” on page 232.

To create a message producer with no associated destination when using the C++ API, a default xms::Destination object created using the default constructor must be passed into the Session::createProducer() method. For further details about creating a message producer in a C++ API, see “Session” on page 371.

Message producers with associated destination

In this scenario, the message producer is created using a valid destination. During the send operation, the destination need not be specified.

Message consumers

Message consumers can be classified as durable and non-durable subscribers and synchronous and asynchronous message consumers.

Durable subscribers

A durable subscriber is a message consumer that receives all messages published on a topic, including those published while the subscriber is inactive.

The information in this section is relevant only if an application connects to a IBM MQ queue manager or a WebSphere service integration bus.

To create a durable subscriber for a topic, an application calls the Create Durable Subscriber method of a Session object, specifying as parameters a name that identifies the durable subscription and a Destination object representing the topic. The application can create a durable subscriber with or without a message selector, and it can specify whether the durable subscriber is to receive messages published by its own connection.

The session used to create a durable subscriber must have an associated client identifier. The client identifier is the same as that associated with the connection that is used to create the session; it is specified as described in “ConnectionFactories and Connection objects” on page 26.

The name that identifies the durable subscription must be unique within the client identifier, and therefore the client identifier forms part of the full, unique identifier of the durable subscription. The messaging server maintains a record of the durable subscription and ensures that all messages published on the topic are retained until they are acknowledged by the durable subscriber or they expire.

The messaging server continues to maintain the record of the durable subscription even after the durable subscriber closes. To reuse a durable subscription that was created previously, an application must create a durable subscriber specifying the same subscription name, and using a session with the same client identifier, as those associated with the durable subscription. Only one session at a time can have a durable subscriber for a particular durable subscription.

The scope of a durable subscription is the messaging server that is maintaining a record of the subscription. If two applications connected to different messaging servers each create a durable subscriber using the same subscription name and client identifier, two completely independent durable subscriptions are created.

To delete a durable subscription, an application calls the Unsubscribe method of a Session object, specifying as a parameter the name that identifies the durable subscription. The client identifier associated with the session must be the same as that associated with the durable subscription. The messaging server deletes the record of the durable subscription that it is maintaining and does not send any more messages to the durable subscriber.

To change an existing subscription, an application can create a durable subscriber using the same subscription name and client identifier, but specifying a different topic, or message selector (or both). Changing a durable subscription is equivalent to deleting the subscription and creating a new one.

For an application that connects to IBM MQ queue manager, XMS manages the subscriber queues. Hence the application is not required to specify a subscriber queue. XMS will ignore the subscriber queue if specified.

However for an application that connects to IBM MQ v6.0 queue manager, each durable subscriber must have a designated subscriber queue. To specify the name of the subscriber queue for a topic, set the XMSC_WMQ_DUR_SUBQ property of the Destination object representing the topic. The default subscriber queue is SYSTEM.JMS.D.SUBSCRIBER.QUEUE.

Durable subscribers connecting to WebSphere MQ v6.0 queue managers can share a single subscriber queue, or each durable subscriber can retrieve its messages from its own exclusive subscriber queue. For a discussion about which approach to adopt for your application, see *IBM MQ Using Java*.

Note that you cannot change the subscriber queue for a durable subscription. The only way to change the subscriber queue is to delete the subscription and create a new one.

For an application that connects to a service integration bus, each durable subscriber must have a designated durable subscription home. To specify the durable subscription home for all durable subscribers that use the same connection, set the XMSC_WPM_DUR_SUB_HOME property of the ConnectionFactory object that is used to create the connection. To specify the durable subscription home for an individual topic, set the XMSC_WPM_DUR_SUB_HOME property of the Destination object representing the topic. A durable subscription home must be specified for a connection before an application can create a durable subscriber that uses the connection. Any value specified for a destination overrides the value specified for the connection.

Non-durable subscribers

A non-durable subscriber is a message consumer that only receives messages that are published while the subscriber is active. Messages delivered while the subscriber is inactive are lost.

The information in this section is relevant only when you are using publish/subscribe messaging over IBM MQ v6.0 queue manager.

If consumer objects are not deleted before or during the closing of the connection, messages can be left on the broker queues for subscribers that are no longer active.

In this situation, the queues can be cleared of these messages using the Cleanup utility provided with IBM MQ Classes for JMS. Details of how to use this

utility are provided in *IBM MQ Using Java*. You may also need to increase the queue depth of the subscriber queue if there are large numbers of messages left on this queue.

Synchronous message consumers

The synchronous message consumer receives the messages from a queue synchronously.

A synchronous message consumer receives one message at a time. When the Receive(with a wait interval) method is used; the call waits only a specified period of time in milliseconds for a message, or until the message consumer is closed.

If the Receive with No Wait method is used, the synchronous message consumer receives messages without any delay; if the next message is available, it is received immediately, otherwise a pointer to a null Message object is returned.

Asynchronous message consumers

The asynchronous message consumer receives message from a queue asynchronously. The message listener registered by the application is invoked whenever a new message is available on the queue.

Poison messages

A poison message is one which cannot be processed by a receiving MDB application. If a poison message is encountered, the XMS MessageConsumer object can requeue it according to two queue properties, BOQUEUE, and BOTHRESH.

In some circumstances, a message delivered to an MDB might be rolled back onto a IBM MQ queue. This can happen, for example, if a message is delivered within a unit of work that is subsequently rolled back. A message that is rolled back is generally delivered again, but a badly formatted message might repeatedly cause an MDB to fail and therefore cannot be delivered. Such a message is called a poison message. You can configure IBM MQ so that the poison message is automatically transferred to another queue for further investigation or is discarded. For information about how to configure IBM MQ in this way, see Handling poison messages in ASF.

Sometimes, a badly-formatted message arrives on a queue. In this context, badly-formatted means that the receiving application cannot process the message correctly. Such a message can cause the receiving application to fail and to back out this badly-formatted message. The message can then be repeatedly delivered to the input queue and repeatedly backed out by the application. These messages are known as poison messages. The XMS MessageConsumer object detects poison messages and reroutes them to an alternative destination.

The IBM MQ queue manager keeps a record of the number of times that each message has been backed out. When this number reaches a configurable threshold value, the message consumer requeues the message to a named backout queue. If this re-queuing fails for any reason, the message is removed from the input queue and either requeued to the dead-letter queue, or discarded.

XMS ConnectionConsumer objects handle poison messages in the same way and using the same queue properties. If multiple connection consumers are monitoring the same queue, it is possible that the poison message may be delivered to an

application more times than the threshold value before the requeue occurs. This behavior is due to the way individual connection consumers monitor queues and requeue poison messages.

The threshold value and the name of the back out queue are attributes of a IBM MQ queue. The names of the attributes are BackoutThreshold and BackoutRequeueQName. The queue they apply to is as follows:

- For point-to-point messaging, this is the underlying local queue. This is important when message consumers and connection consumers use queue aliases.
- For publish/subscribe messaging in IBM MQ messaging provider normal mode, it is the model queue from which the Topic's managed queue is created.
- For publish/subscribe messaging in IBM MQ messaging provider migration mode, it is the CCSUB queue defined on the TopicConnectionFactory object, or the CCDSUB queue defined on the Topic object.

To set the BackoutThreshold and BackoutRequeueQName attributes, issue the following MQSC command:

```
ALTER QLOCAL(your.queue.name) BOTHRESH(threshold value)
BOQUEUE(your.backout.queue.name)
```

For publish/subscribe messaging, if your system creates a dynamic queue for each subscription, these attribute values are obtained from the IBM MQ classes for JMS model queue, SYSTEM.JMS.MODEL.QUEUE. To alter these settings, use:

```
ALTER QMODEL(SYSTEM.JMS.MODEL.QUEUE) BOTHRESH(threshold value)
BOQUEUE(your.backout.queue.name)
```

If the backout threshold value is zero, poison message handling is disabled, and poison messages remain on the input queue. Otherwise, when the backout count reaches the threshold value, the message is sent to the named backout queue. If the backout count reaches the threshold value, but the message cannot go to the backout queue, the message is sent to the dead-letter queue or it is discarded. This situation occurs if the backout queue is not defined, or if the MessageConsumer object cannot send the message to the backout queue.

Handling poison messages in ASF

When you use Application Server Facilities (ASF), the ConnectionConsumer, rather than the MessageConsumer, processes poison messages. The ConnectionConsumer requeues messages according to the BackoutThreshold and BackoutRequeueQName properties of the queue.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the session that the application server provides:

- When the session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling Session.recover().

  Typically, the client implementation of MessageListener or the application server calls Message.acknowledge(). Message.acknowledge() acknowledges all messages delivered on the session so far.
- When the session is transacted, unacknowledged messages can be backed out by the application server calling Session.rollback().

Queue browsers

An application uses a queue browser to browse messages on a queue without removing them.

To create a queue browser, an application calls the Create Queue Browser method of a Session object, specifying as a parameter a Destination object that identifies the queue to be browsed. The application can create a queue browser with or without a message selector.

After creating a queue browser, the application can call the Get Messages method of the QueueBrowser object to get a list of the messages on the queue. The list of messages is returned as an iterator that encapsulates a list of Message objects. The order of the Message objects in the list is the same as the order in which the messages would be retrieved from the queue. The application can then use the iterator to browse each message in turn.

The iterator is updated dynamically as messages are put on the queue and removed from the queue. Each time the application uses the iterator to browse the next message on the queue, the message returned reflects the current contents of the queue. When the iterator indicates that there are no more messages on the queue, it stops returning messages, even if further messages arrive on the queue. However, by calling the Reset Iterator method of the Iterator object, the application can continue to use the same iterator to browse messages, starting from the beginning of the queue.

An application can call the Get Messages method more than once for a given queue browser. Each call returns a new iterator. The application can therefore use more than one iterator to browse the messages on a queue and maintain multiple positions within the queue.

An application can use a queue browser to search for a suitable message to remove from a queue, and then use a message consumer with a message selector to remove the message. The message selector can select the message according to the value of the JMSMessageID header field. For information about this and other JMS message header fields, see "Header fields in an XMS message" on page 91.

Requestors

An application uses a requestor to send a request message and then to wait for and to receive the reply.

Many messaging applications are based on algorithms that send a request message and then wait for a reply. XMS provides a class called Requestor to help with the development of this style of application.

To create a requestor, an application calls the Create Requestor constructor of the Requestor class, specifying as parameters a Session object and a Destination object that identifies where request messages are to be sent. The session must not be transacted nor have an acknowledgement mode of XMSC_CLIENT_ACKNOWLEDGE. The constructor automatically creates a temporary queue or topic where reply messages are to be sent.

After creating a requestor, the application can call the Request method of the Requestor object to send a request message and then wait for, and receive, a reply from the application that receives the request message. The call waits until the

reply is received or until the session ends, whichever occurs first. Only one reply is required by the requestor for each request message.

When the application closes the requestor, the temporary queue or topic is deleted. The associated session, however, does not close.

Object Deletion

When an application deletes an XMS object that it has created, XMS releases the internal resources that have been allocated to the object.

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 have no function independent from the object. These objects are shown in Table 7. 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 7. 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, MessageProducer, QueueBrowser, and Requestor objects

XMS primitive types

XMS provides equivalents of the eight Java primitive types (byte, short, int, long, float, double, char and boolean). This allows the interchange of messages between XMS and JMS without data becoming lost or corrupted.

Table 8 lists the Java equivalent data type, size, and minimum and maximum value of each XMS primitive type.

Table 8. XMS data types and their Java equivalents

XMS data type	Compatible Java data type	Size	Minimum value	Maximum value
xmsBOOL	boolean	32 bits	xmsFALSE	xmsTRUE
xmsSBYTE	byte	8 bits	-2^7 (-128)	2^7-1 (127)
xmsCHAR	byte	8 bits	-2^7 (-128)	2^7-1 (127)
xmsCHAR16	char	16 bits	0 (\u0000)	2^{16}-1 (\uFFFF)
xmsSHORT	short	16 bits	-2^{15} (-32768)	2^{15}-1 (32767)
xmsINT	int	32 bits	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
xmsLONG	long	64 bits	-2^{63} (-9223372036854775808)	2^{63}-1 (9223372036854775807)

Table 8. XMS data types and their Java equivalents (continued)

XMS data type	Compatible Java data type	Size	Minimum value	Maximum value
xmsFLOAT	float	32 bits	-3.402823E-38 (to 7 digits precision)	3.402823E+38 (to 7 digits precision)
xmsDOUBLE	double	64 bits	-1.79769313486231E-308 (to 15 digits precision)	1.79769313486231E+308 (to 15 digits precision)

Implicit conversion of a property value from one data type to another

When an application gets the value of a property, the value can be converted by XMS into another data type. Many rules govern which conversions are supported and how XMS performs the conversions.

A property of an object has a name and a value; the value has an associated data type, where the value of a property is also referred to as the *property type*.

An application uses the methods of the PropertyContext class to get and set the properties of objects. In order to get the value of a property, an application calls the method that is appropriate for the property type. For example, to get the value of an integer property, an application typically calls the Get Integer Property method.

However, when an application gets the value of a property, the value can be converted by XMS into another data type. For example, to get the value of an integer property, an application can call the Get String Property method, which returns the value of the property as a string. The conversions supported by XMS are shown in Table 9.

Table 9. Supported conversions from a property type to other data types

Property type	Supported target data types
String	xmsBOOL, xmsDOUBLE, xmsFLOAT, xmsINT, xmsLONG, xmsSBYTE, xmsSHORT
xmsBOOL	String, xmsSBYTE, xmsINT, xmsLONG, xmsSHORT
xmsCHAR	String
xmsDOUBLE	String
xmsFLOAT	String, xmsDOUBLE
xmsINT	String, xmsLONG
xmsLONG	String
xmsSBYTE	String, xmsINT, xmsLONG, xmsSHORT
xmsSBYTE array	String
xmsSHORT	String, xmsINT, xmsLONG

The following general rules govern the supported conversions:

- Numeric property values can be converted from one data type to another provided no data is lost during the conversion. For example, the value of a property with data type xmsINT can be converted into a value with data type xmsLONG, but it cannot be converted into a value with data type xmsSHORT.
- A property value of any data type can be converted into a string.

- A string property value can be converted to any other data type provided the string is formatted correctly for the conversion. If an application attempts to convert a string property value that is not formatted correctly, XMS may return errors.
- If an application attempts a conversion that is not supported, XMS may return an error.

The following rules apply when a property value is converted from one data type to another:

- When converting a boolean property value to a string, the value xmsTRUE is converted to the string "true", and the value false is converted to the string "false".
- When converting a boolean property value to a numeric data type, including xmsSBYTE, the value xmsTRUE is converted to 1, and the value xmsFALSE is converted to 0.
- When converting a string property value to a boolean value, the string "true" (not case sensitive) or "1" is converted to xmsTRUE, and the string "false" (not case sensitive) or "0" is converted to xmsFALSE. All other strings cannot be converted.
- When converting a string property value to a value with data type xmsINT, xmsLONG, xmsSBYTE, or xmsSHORT, the string must have the following format:

  [*blanks*][*sign*]*digits*

  The string components are defined as follows:

  blanks Optional leading blank characters.

  sign An optional plus sign (+) or minus sign (-) character.

  digits A contiguous sequence of digit characters (0-9). At least one digit character must be present.

  After the sequence of digit characters, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

  XMS may return an error if the string is not formatted correctly.
- When converting a string property value to a value with data type xmsDOUBLE or xmsFLOAT, the string must have the following format:

  [*blanks*][*sign*][*digits*][*point*[*d_digits*]][*e_char*[*e_sign*]*e_digits*]

  The string components are defined as follows:

  blanks (Optional) Leading blank characters.

  sign (Optional) Plus sign (+) or minus sign (-) character.

  digits A contiguous sequence of digit characters (0-9). At least one digit character must be present in either *digits* or *d_digits*.

  point (Optional) Decimal point (.).

  d_digits
  A contiguous sequence of digit characters (0-9). At least one digit character must be present in either *digits* or *d_digits*.

  e_char An exponent character, which is either *E* or *e*.

  e_sign (Optional) Plus sign (+) or minus sign (-) character for the exponent.

e_digits

A contiguous sequence of digit characters (0-9) for the exponent. At least one digit character must be present if the string contains an exponent character.

After the sequence of digit characters, or the optional characters representing an exponent, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

XMS may return an error if the string is not formatted correctly.

- When converting a numeric property value to a string, including a property value with data type xmsSBYTE, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".
- When converting a byte array property value to a string, each byte is converted to the 2 hexadecimal characters that represent the byte. For example, the byte array {0xF1, 0x12, 0x00, 0xFF} is converted to the string "F11200FF".

Conversions from a property type to other data types are supported by the methods of both the Property and the PropertyContext classes. However, the C functions xmsPropertyGetStringByRef() and xmsGetStringPropertyByRef() make no attempt to convert a property value that is not a string. If an application calls either of these functions to get a pointer to a property value that is not a string, XMS may return an error.

Iterators

An iterator encapsulates a list of objects and a cursor that maintains the current position in the list. A C or C++ application uses an iterator to retrieve each object in the list in turn.

When an iterator is created, the position of the cursor is before the first object. An application uses an iterator to retrieve each object in turn. To retrieve the objects, the application uses the following three methods of the Iterator class:

- Check for More Objects
- Get Next Object
- Reset Iterator

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

An application can use an iterator to perform the following tasks:

- To get the properties of a message
- To get the name-value pairs in the body of a map message
- To browse the messages on a queue
- To get the names of the JMS defined message properties supported by a connection

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 = xmsIteratorGetNext(it, (xmsHObj)&p, xmsError);
    if (rc == XMS_OK)
    {
      xmsPropertyGetName(p, name, 100, &len, xmsError);
      printf("Property name=\"%s\"\n", name);
      xmsPropertyGetTypeId(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:
        {
          xmsINT 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

For C or C++ strings of character set identifiers (CCSIDs) that an object passes to, or receives from, XMS might require conversion. The XMSC_CLIENT_CCSID property of the object tells XMS which code page the object is using.

When an object in a C or C++ application passes a string of character data to XMS across the API XMS converts (if necessary) the character data in the string from the code page used by the object into the code page required by XMS for the data. Similarly, when an object receives a string of character data from XMS across the API XMS converts (if necessary) the character data in the string from the code page that the data is currently in into the code page used by the object. Therefore, in order to convert the character data in a string, XMS must identify which code page an object is using.

The XMSC_CLIENT_CCSID property of a ConnectionFactory, Connection, Session, MessageProducer, or MessageConsumer object specifies which code page the object is using. The value of the XMSC_CLIENT_CCSID propertyis a CCSID which identifies a code page. XMS sets the property when an application creates one of these objects, but the application can change its value subsequently.

When an application starts, XMS derives an appropriate CCSID for the application from the environment in which the application is running. This CCSID is called the *process CCSID*. At any time, the application can change the process CCSID by calling xmsSetClientCCSID(). This is a C function that does not belong to any class, but C++ applications can use the function as well.

When an application creates a connection factory, XMS sets the XMSC_CLIENT_CCSID property of the object. If the connection factory is created from an object definition retrieved from a repository of administered objects, and the object definition specifies a value for the XMSC_CLIENT_CCSID property, XMS uses this value to set the property. Otherwise, XMS sets the property to the special value XMSC_CCSID_PROCESS, which means that the connection factory is using the code page identified by the process CCSID.

When an application uses a connection factory 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 XMSC_CLIENT_CCSID property of the ConnectionFactory object, XMS does not propagate the change to the XMSC_CLIENT_CCSID property of the Connection object.

In the same way, when an application uses a connection to create a session, XMS copies the XMSC_CLIENT_CCSID property of the Connection object to the newly created Session object. When an application uses a session to create a message producer or message consumer, XMS copies the XMSC_CLIENT_CCSID property of the Session object to the newly created MessageProducer or MessageConsumer object. In each case, XMS copies the property only at the time the application creates the object.

At any time, an application can change the value of the XMSC_CLIENT_CCSID property of an object by calling the Set Integer Property method of the PropertyContext class. The application can set the property to one of the following values:

A coded character set identifier (CCSID)
: The object is using the code page identified by the specified CCSID. If the application specifies either a CCSID that is not valid or a CCSID for which the platform does not support code page conversion, the call fails and XMS returns an error.

XMSC_CCSID_UTF8
: The object is using the UTF-8 representation of Unicode data.

XMSC_CCSID_UTF16
: The object is using the UTF-16 representation of Unicode data.

XMSC_CCSID_UTF32
: The object is using the UTF-32 representation of Unicode data.

XMSC_CCSID_PROCESS
: The object is using the code page identified by the process CCSID. XMS queries the process CCSID whenever it needs to determine which code page the object is using. If the application changes the process CCSID by calling xmsSetClientCCSID(), XMS detects the change the next time it determines which code page the object is using.

  This is a special value of the property and is not an actual CCSID.

XMSC_CCSID_HOST
The object is using the code page identified by the CCSID that is derived from the environment in which the application is running. This CCSID is the same as the process CCSID unless the application has changed the process CCSID by calling xmsSetClientCCSID().

This is a special value of the property and is not an actual CCSID.

XMSC_CCSID_NO_CONVERSION
Strings of character data received by the object are not converted.

This is a special value of the property and is not an actual CCSID.

The strings of character data that an application passes to, and receives from, XMS include (but are not exclusively confined to) the strings in messages. The strings that require conversion might be in any of the following parts of a message:

- Header fields (see "Header fields in an XMS message" on page 91)
- Properties (see "Properties of an XMS message" on page 92)
- The body (see "The body of an XMS message" on page 95)

When XMS converts the strings in an outgoing message, it uses the code page associated with the session that created the message. When XMS converts the strings in an incoming message, it uses the code page associated with the message consumer that receives the message. XMS determines the code page from the value of the XMSC_CLIENT_CCSID property of the relevant Session or MessageConsumer object.

Converting strings in messages might have an impact on performance depending on the amount of data to be converted the frequency with which conversion occurs. If you are designing applications to maximize the throughput of messages, you might want to consider ways of reducing the amount of data conversion that is required. The following examples illustrate how this can be done:

- For example, you might know that the strings in incoming messages are in a certain code page (the UTF-8 representation of Unicode data). You might determine this information from a knowledge of the application that sends the messages or the message server environment through which the messages pass. If you can arrange for the application that receives the messages to use the same code page, no data conversion of the strings is required.
- If you can arrange for both the sending and receiving applications to use the same code page, you might consider using bytes messages and reading and writing strings as byte arrays. No data conversion is performed in these circumstances.

XMS error and exception codes

XMS uses a range of error codes to indicate failures. These error codes are not explicitly listed in this documentation because they may vary from release to release. Only XMS exception codes (in the format XMS_X_...) are documented because they remain the same across releases of XMS.

Building your own applications

You build your own applications like you build the sample applications.

About this task

This section lists the prerequisites you need to build your own C or C++ application.This chapter lists the prerequisites you need to build your own C or C++ applications. For additional guidance on how to build your own applications, use the makefiles provided for each sample application.

Tip: To assist with problem diagnosis in the event of a failure, you might find it helpful to compile applications with symbols included.

On Windows, if you are building a C or C++ application, make sure that your compilation settings are correct. All of the XMS libraries are compiled using the multithreaded runtime libraries. Therefore, when you are a building C or C++ application using the XMS libraries, make sure that your project or makefile compiler flag settings are set to select multi-threaded runtime libraries (/MD or, for debug, /MDd), and not single-threaded runtime libraries (/ML or, for debug, /MLd).

Build your application:

- C or C++, as described in "Building the C or C++ sample applications" on page 105

Build your C or C++ application, as described in "Building the C or C++ sample applications" on page 105

Network stack selection mechanism

This section describes the network stack selection mechanism when both IPv4 and IPv6 network stacks are enabled on a machine.

When both IPv4 and IPv6 network stacks are enabled on a machine, the connection binds to either of the two network stacks based on the host name and local address properties.

The host name is specified by any of these properties, XMSC_WMQ_HOST_NAME, and XMSC_WPM_PROVIDER_ENDPOINTS, while the local address may be determined by XMSC_WMQ_LOCAL_ADDRESS, or XMSC_WPM_LOCAL_ADDRESS.

The following table lists the outcome for the possible combinations of network stacks in use for the host name and local address.

Table 10. Network stack selection mechanism

Stack	Host Name	Local Address	Connection result
IPv4 only stack	IPv4 address	None	Connection binds to IPv4 stack
	IPv6 address	None	Connection fails to resolve host name
	Host name resolves to both IPv4 and IPv6 addresses	None	Connection binds to IPv4 stack
	IPv4 address	IPv4 address	Connection binds to IPv4 stack
	IPv6 address	IPv4 address	Connection fails to resolve host name
	Remote host name resolves to both IPv4 and IPv6 addresses	IPv4 address	Connection binds to IPv4 stack
	Any address	IPv6 address	Connection fails to resolve local address
	IPv4 address	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv4 stack
	IPv6 address	Local address resolves to both IPv4 and IPv6 addresses	Connection fails to resolve host name
	Remote host name resolves to both IPv4 and IPv6 addresses	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv4 stack

Table 10. Network stack selection mechanism (continued)

Stack	Host Name	Local Address	Connection result
Dual (IPv4 and IPv6) stack	IPv4 address	None	Connection binds to IPv4 stack
	IPv6 address	None	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	None	For WPM, connection binds to IPv6 stack. For IBM MQ, channel binds to stack determined by the value of the MQIPADDRV environment variable.
	IPv4 address	IPv4 address	Connection binds to IPv4 stack
	IPv6 address	IPv4 address	Connection fails to resolve host name
	Remote host name resolves to both IPv4 and IPv6 addresses	IPv4 address	Connection binds to IPv4 stack
	IPv4 address	IPv6 address	Maps an IPv4 host name to an IPv4 mapped IPv6 address. IPv6 implementations that do not support IPv4 mapped IPv6 addressing fail to resolve host name.
	IPv6 address	IPv6 address	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	IPv6 address	Connection binds to IPv6 stack
	IPv4 address	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv4 stack
	IPv6 address	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	Local address resolves to both IPv4 and IPv6 addresses	For WPM, connection binds to IPv6 stack. For IBM MQ, channel binds to stack determined by the value of the MQIPADDRV environment variable.

Table 10. Network stack selection mechanism (continued)

Stack	Host Name	Local Address	Connection result
IPv6 only stack	IPv4 address	None	Maps an IPv4 host name to an IPv4 mapped IPv6 address. IPv6 implementations that do not support IPv4 mapped IPv6 addressing fail to resolve host name
	IPv6 address	None	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	None	Connection binds to IPv6 stack
	Any address	IPv4 address	Connection fails to resolve local address
	IPv4 address	IPv6 address	Maps an IPv4 host name to an IPv4 mapped IPv6 address. IPv6 implementations that do not support IPv4 mapped IPv6 addressing fail to resolve host name
	IPv6 address	IPv6 address	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	IPv6 address	Connection binds to IPv6 stack
	IPv4 address	Local address resolves to both IPv4 and IPv6 addresses	Maps an IPv4 host name to an IPv4 mapped IPv6 address. IPv6 implementations that do not support IPv4 mapped IPv6 addressing fail to resolve host name
	IPv6 address	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv6 stack
	Remote host name resolves to both IPv4 and IPv6 addresses	Local address resolves to both IPv4 and IPv6 addresses	Connection binds to IPv6 stack

Automatic WMQ client reconnection through XMS

You can make your XMS applications to reconnect automatically following a network, queue manager or server failure. This feature is available only with IBM MQ V7.0 client or above. This is controlled by connection factory properties XMSC_WMQ_CLIENT_RECONNECT_OPTIONS and XMSC_WMQ_CONNECTION_NAME_LIST.

XMSC_WMQ_CLIENT_RECONNECT_OPTIONS

By default, XMS client applications are not automatically reconnected. Automatic reconnection is enabled by setting the property XMSC_WMQ_CLIENT_RECONNECT_OPTIONS to XMSC_WMQ_CLIENT_RECONNECT or XMSC_WMQ_CLIENT_RECONNECT_Q_MGR.

Reconnecting to a queue manager of the same name does not guarantee that you have reconnected to the same instance of a queue manager. Option XMSC_WMQ_CLIENT_RECONNECT_Q_MGR only allows reconnection to an instance of the same queue manager.

Seer XMSC_WMQ_CLIENT_RECONNECT_Q_MGR for details of this property.

XMSC_WMQ_CONNECTION_NAME_LIST

XMS application can connect to queue manager using XMSC_WMQ_HOST_NAME and XMSC_WMQ_PORT. A new property XMSC_WMQ_CONNECTION_NAME_LIST is provided where a list of connection names can be given.

If XMS application looses connection with the server, the connections are tried in the order they are specified in the connection list until a connection is successfully established or for client reconnect timeout duration.

There is a timeout value you can configure to limit the time a client waits for reconnection. The value (in seconds) is set in CHANNELS stanza of mqclient.ini file.

```
CHANNELS:
MQReconnectTimeout = 1800
```

Note: No reconnection attempts are made after the timeout has expired. The default value is 1800 seconds (30 minutes). See XMSC_WMQ_CONNECTION_NAME_LIST property for details.

XMS application can register an exception listener function so that it can be informed about changes in the state of reconnection. If XMS detects a problem with the connection, during reconnection, 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.

Errors are indicated by the error codes:

MQRC_RECONNECTING indicates connection has failed, and the system is attempting to reconnect.

MQRC_RECONNECTED indicates reconnection has been made and all handles are successfully reestablished.

For more information on exception listener function, see Exception listener functions in C.

A reconnectable client is able to reconnect automatically only after connecting.

Note: Some restrictions apply during reconnection. When a client reconnection occurs, the XMS operation will pause. Depending on the nature of the XMS session, it may be that subsequent commits or message acknowledges would fail and need to be retried. An application should be written to handle this.

Connecting applications in a multiple installation environment

With multiple installations of IBM MQ on a system, you need to consider how XMS C/C++ works with a particular installation.

By default, applications use the primary installation. If there is no primary installation, or you do not want to use the primary installation, you must use the **setmqenv** command to specify which installation to use.

For more information about installation of multiple copies of IBM MQ, and supported versions, see the IBM MQ information center.

On UNIX and Linux systems, using the **setmqenv** command set the **LD_LIBRARY_PATH**(**LIBPATH** on AIX), with the **-n** and **-k** option using the following command.

.<INSTALLATION PATH>/bin/setmqenv -n InstallationName -k, where **-n InstallationName**.

The **-k** parameter updates the **LD_LIBRARY_PATH**, or **LIBPATH** environment variable, with the path to the IBM MQ libraries at the start.

On UNIX platforms the leading "**.** "is critical. The dot, followed by a space, instructs the command shell to run **setmqenv** in the same command shell, and therefore inherit the environment set by **setmqenv**.

On Windows platforms, set the PATH environment variable using the **setmqenv -n** option.

INSTALLATION_PATH\bin\setmqenv -n InstallationName, where **-n InstallationName** sets up the environment for the installation named **InstallationName**.

For more information about the other options for **setmqenv**, see the IBM MQ information center.

If the XMS C/C++ application is set to use the IBM MQ say v9.x.x.x environment in bindings connection mode, then the IBM MQ v9.x.x.x libraries internally load the required libraries, depending on the queue manager the application is connecting to.

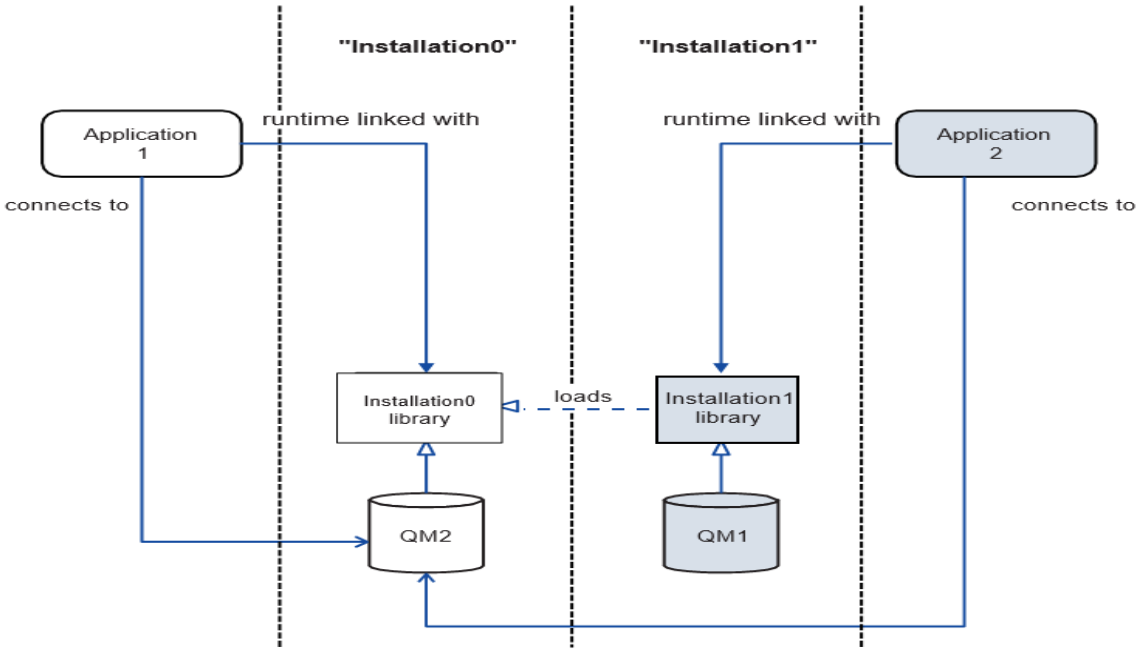


Figure 3. Connecting applications in a multiple installation environment

For example, Figure 3 shows a multiple installation environment with *Installation0* and *Installation1*. Two applications are connected to these installations, but they load different library versions.

Assume that Installtion0 has lower version on MQ installed than Installion1

Application 1 directly loads *Installtion0* library. When application 1 connects to QM2, the Installtion0 libraries are used. If application 1 attempts to connect to QM1, or if QM2 is associated with Installation1, application 1 fails with a 2059 (080B) (RC2059): MQRC_Q_MGR_NOT_AVAILABLE error. The application fails because the *Installtion0* library is not capable of loading other library versions. That is, if Installtion0 libraries are directly loaded, you cannot use a queue manager associated with an installation at a later version of IBM MQ.

If Application 2 connects to QM2, the *Installtion1* libraries load, and use the *Installtion0* library. If application 2 connects to QM1, or if QM2 is associated with *Installation1*, then the *Installtion1* library is loaded, and the application works as expected. directly loads a Installtion1 library.

Chapter 6. Writing XMS applications in C

This chapter provides information help you write XMS applications in C.

About this task

This chapter provides information that is specific to writing XMS applications in C. For general information about writing XMS applications, see Chapter 5, "Developing XMS applications," on page 25.

The chapter contains the following sections:

- "Object handles in C"
- "Object Properties in C" on page 58
- "C functions that return a string by value" on page 58
- "C functions that return a byte array by value" on page 59
- "C functions that return a string or byte array by reference" on page 60
- "C functions that accept a string as input" on page 61
- "Error handling in C" on page 61
- "Message and exception listener functions in C" on page 62

Object handles in C

A C application uses an object handle to access an object. There are two kinds of object handles; one has a data type that is related to the type of the object, and the other is a generic object handle whose data type is not related to the type of the object.

When a C application calls a function to create an object, XMS stores the object internally and returns a handle for the object to the application. The application can then use the handle to access the object.

Every object handle has a data type, which is related to the object type. Table 11 shows the object handle data type for each type of object. Note that BytesMessage, MapMessage, ObjectMessage, StreamMessage, TextMessage, and Message objects all have handles with the same data type, xmsHMsg. For more information about how to use handles for messages, see "The body of an XMS message" on page 95.

Table 11. Data types for object handles

Type of object	Object handle data type
Connection	xmsHConn
ConnectionFactory	xmsHConnFact
ConnectionMetaData	xmsHConnMetaData
Destination	xmsHDest
ErrorBlock	xmsHErrorBlock
InitialContext	xmsHInitialContext
Iterator	xmsHIterator
Message, BytesMessage, MapMessage, ObjectMessage, StreamMessage, and TextMessage	xmsHMsg

Table 11. Data types for object handles (continued)

Type of object	Object handle data type
MessageConsumer	xmsHMsgConsumer
MessageProducer	xmsHMsgProducer
Property	xmsHProperty
QueueBrowser	xmsHQueueBrowser
Requestor	xmsHRequestor
Session	xmsHSess

Certain functions return a generic object handle, which is not related to the type of object that they create. A generic object handle has data type xmsHObj.

If an application receives a generic object handle from one of these functions, the application can call the xmsGetHandleTypeId() function in the PropertyContext class to determine the related data type object handle for that object. The application can then call any function to perform an operation on the object by casting, if necessary, the generic object handle to the data type required by the function.

Object Properties in C

A C application uses the functions in the PropertyContext class to get and set the properties of objects.

For each XMS data type, the PropertyContext class contains a function to get the value of a property with that data type and a function to set its value. For example, a C application can call the function xmsGetIntProperty() to get the value of an integer property and the function xmsSetIntProperty() to set its value.

Functions in the PropertyContext class can operate on any object that can have properties. Each individual class does not contain its own functions to get and set the properties of objects of that class. As a result, functions in the PropertyContext class accept only generic object handles as input. If an application is currently accessing an object using a handle with a data type that is related to the type of the object, the application must cast the handle to the generic object handle data type, xmsHObj, in order to get or set the properties of the object. For more information about generic object handles, see “Object handles in C” on page 57.

All objects except ErrorBlock, Iterator, and Property objects can have properties.

If an application sets the value of a property, the new value replaces any previous value the property had.

C functions that return a string by value

This section describes the interface used by C functions that return a string by value.

In the C API, certain functions return a string as a parameter. Each of these functions uses the same interface for retrieving a string. The following example C code illustrates the function, xmsGetStringProperty() in the PropertyContext class:

```
xmsRC xmsGetStringProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsINT *actualLength,
                           xmsHErrorBlock errorBlock);
```

Three parameters control the retrieval of a 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. If you specify `XMSC_QUERY_SIZE` instead, the string is not returned, but its length is returned in the actualLength parameter.

actualLength
This output parameter is the length of the string that XMS copies into the buffer. If data conversion is required, this is the length after conversion. The length is measured in bytes. XMS always returns a null terminated string, and the length reported to the application includes the terminating null character. If you specify a null pointer for this parameter on input, the length of the string is not returned.

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

If an XMS application receives a message sent by a JMS application, strings in the header fields, properties, and body of the message might contain embedded null characters. Strings containing embedded nulls cannot be manipulated using the standard C string manipulator because they read the first null character to be the end of the string.

C functions that return a byte array by value

This section describes the interface used by C functions that return a byte array by value.

In the C API, certain functions return a byte array as a parameter. Each of these functions uses the same interface for retrieving a byte array. The following example written in C code illustrates the function, xmsGetByteArrayProperty() in the PropertyContext class:

```
xmsRC xmsGetByteArrayProperty(xmsHObj object
                              xmsCHAR *propertyName,
                              xmsSBYTE *propertyValue,
                              xmsINT length,
                              xmsINT *actualLength
                              xmsHErrorBlock errorBlock) const;
```

Three parameters control the retrieval of a byte array:

propertyValue
This parameter is a pointer to a buffer provided by the application into which XMS copies the bytes in the array.

length This parameter is a pointer to the length of the buffer in bytes. This is an input parameter that must be set by the application before the call. If you specify XMSC_QUERY_SIZE instead, the byte array is not returned, but its length is returned in the actualLength parameter.

actualLength
This output parameter is the number of bytes in the array that XMS copies into buffer. If you specify a null pointer for this parameter on input, the length of the array is not returned.

If the buffer is not large enough to store the whole array, XMS returns the array truncated to the length of the buffer, sets the actualLength parameter to the length of the whole array, and returns an error.

Two functions, xmsBytesMsgReadBytes() and xmsStreamMsgReadBytes(), have a slightly different interface. Using one of these functions, an application can retrieve a byte array in stages by successive calls to the function. Each call reads bytes into the buffer provided by the application starting from the current position of an internal cursor, and an output parameter, returnedLength, to determine how many bytes have been read into the buffer. Neither function has the equivalent of the actualLength parameter in its interface, but an application can specify XMSC_QUERY_SIZE to determine the number of bytes remaining in an array starting from the current position of the cursor.

C functions that return a string or byte array by reference

This section describes the interface used by 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 by value" on page 58 or "C functions that return a byte array by value" on page 59, 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 affect performance.

The C API has functions to deliver better performance. 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. Examples of these functions are xmsBytesMsgReadBytesByRef() and xmsGetStringPropertyByRef().

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.

Note: These functions return a pointer to memory owned by XMS. You must not attempt to free or modify the contents of this memory as doing so will cause unpredictable results.

The pointer returned to the application remains valid until the next time that the application adds a new piece of data to the bytes message.

C functions that accept a string as input

This section describes the interface used by C functions that accept a string as an input parameter.

In the C API, certain functions accept a string as an input parameter. Each of these functions uses the same interface for passing a string to XMS. The following example of C code illustrates the function, xmsSetStringProperty() in the PropertyContext class:

```
xmsRC xmsSetStringProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsHErrorBlock errorBlock);
```

Two input parameters control passing a string to XMS:

propertyValue
A pointer to a character array that contains the string to be passed to XMS.

length The length of the string in bytes. If the string is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and XMS calculates its length.

Error handling 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 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 12 lists the possible return codes and gives their meaning.

Table 12. 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 return code is the same as the exception code that is returned in the error block.

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.

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 it 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. 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 140.

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.

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 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.

Note: The application must 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, by passing a null pointer as a parameter instead of a pointer to a message listener function.

A new message listener function and context data can be registered with a message consumer without cancelling the registration of an existing message listener function. If an existing message listener function is running when a new message listener function is registered, the active message listener function completes as usual, and any subsequent messages are processed by calls to the new message listener function. If a transaction is in progress when a message listener function is changed, the transaction is completed by calls to the new message listener function.

For more information about the message listener function, including its signature, see "MessageListener" on page 184.

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 requires when processing an error block. In all other respects, the context data is used with an exception listener function in the same way as it is used with a message listener function.

For more information about the exception listener function, including its signature, see "MessageListener" on page 328.

Note: The application is required 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, by passing a null pointer as a parameter instead of a pointer to an exception listener function.

Chapter 7. Writing XMS applications in C++

This chapter provides information to help you when writing XMS applications in C++.

About this task

This chapter provides information that is specific to writing XMS applications in C++. For general information about writing XMS applications, see Chapter 5, "Developing XMS applications," on page 25.

The chapter contains the following sections:

v "Namespaces in C++"
v "String objects in C++" on page 66
v "C++ methods that return a byte array" on page 67
v "Properties in C++" on page 67
v "Assignment of XMS objects to variables in C++" on page 67
v "Error handling in C++" on page 70
v "Message and exception listeners in C++" on page 72
v "Use of C APIs in a C++ application" on page 74

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:

v The application can qualify the names of XMS classes with the name of the namespace, xms, as show in the following C++ code fragment:

```
#include <xms.hpp>

using namespace std;

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

  connFact.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
  connFact.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
  connFact.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
  connFact.setIntProperty(XMSC_WMQ_PORT, 1414);
  connFact.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
  connFact.setStringProperty(XMSC_WMQ_CHANNEL, "XMS.SVR.CHL");


  conn = connFact.createConnection();

  // Other code here

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

  return(0);
}
```

v The application can use a using directive to make the names of XMS classes available without having to qualify them. For example:

```
#include <xms.hpp>

using namespace std;
using namespace xms;

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

  connFact.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
  connFact.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
  connFact.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
  connFact.setIntProperty(XMSC_WMQ_PORT, 1414);
  connFact.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
  connFact.setStringProperty(XMSC_WMQ_CHANNEL, " XMS.SVR.CHL ");

  conn = connFact.createConnection();

  // Other code here

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

  return(0);
}
```

String objects 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 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 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 394.

C++ methods that return a byte array

This section describes the interface used by C++ methods that return a byte array.

In the C++ API, certain methods return a byte array as a parameter. Each of these methods uses the same interface for retrieving a byte array. The following example illustrates one of these methods, PropertyContext.getBytesProperty():

```
xmsINT getBytesProperty(const String & propertyName,
                        xmsSBYTE *propertyValue,
                        const xmsINT length,
                        xmsINT *actualLength) const;
```

The parameters propertyValue, length, and actualLength control the retrieval of the byte array in the same way as described in "C functions that return a byte array by value" on page 59.

Other examples of these methods are MapMessage.getBytes(), MapMessage.getObject(), Property.getByteArray(), and String.get().

Properties in C++

A C++ application uses the methods in the PropertyContext class to get and set the properties of objects.

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 classes:

- BytesMessage
- Connection
- ConnectionFactory
- ConnectionMetaData
- Destination
- InitialContext
- MapMessage
- Message
- MessageConsumer
- MessageProducer
- ObjectMessage
- QueueBrowser
- Requestor
- Session
- StreamMessage
- TextMessage

If an application sets the value of a property, the new value replaces any previous value the property had.

Assignment of XMS objects to variables in C++

This section describes how XMS objects are assigned to variables in C++.

The assignment operator is overloaded on each of the XMS classes listed in Table 13. 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 objects, 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 objects, 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 13 indicates, for each type of object, whether the overloaded assignment operator makes a shallow or a deep copy of an object.

Table 13. The XMS classes on which the assignment operator is overloaded

Class	Shallow copy	Deep copy
BytesMessage	UYes	
Connection	UYes	
ConnectionFactory	UYes	
ConnectionMetaData	UYes	
Destination	UYes	
Exception		UYes
IllegalStateException		UYes
InitialContext	UYes	
InvalidClientIDException		UYes
InvalidDestinationException		UYes
InvalidSelectorException		UYes
Iterator	UYes	
MapMessage	UYes	
Message	UYes	
MessageConsumer	UYes	
MessageEOFException		UYes
MessageFormatException		UYes
MessageNotReadableException		UYes
MessageNotWritableException		UYes
MessageProducer	UYes	
ObjectMessage	UYes	
Property		UYes
QueueBrowser	UYes	
Requestor	UYes	
ResourceAllocationException		UYes
SecurityException		UYes

Table 13. The XMS classes on which the assignment operator is overloaded (continued)

Class	Shallow copy	Deep copy
Session	UYes	
StreamMessage	UYes	
String		UYes
TextMessage	UYes	
TransactionInProgressException		UYes
TransactionRolledBackException		UYes

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 concepts:

```
#include <xms.hpp>

using namespace std;

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


  cf.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
  cf.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
  cf.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
  cf.setIntProperty(XMSC_WMQ_PORT, 1414);
  cf.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
  cf.setStringProperty(XMSC_WMQ_CHANNEL, " XMS.SVR.CHL");


  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);
}
```

Error handling in C++

XMS throws an exception when it detects an error while processing a call to a method.

An XMS exception is an object of one of the following types:

- Exception
- IllegalStateException
- InvalidClientIDException
- InvalidDestinationException
- InvalidSelectorException
- MessageEOFException
- MessageFormatException
- MessageNotReadableException
- MessageNotWritableException
- ResourceAllocationException
- SecurityException
- TransactionInProgressException
- TransactionRolledBackException

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 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.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
    connFact.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
    connFact.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
    connFact.setIntProperty(XMSC_WMQ_PORT, 1414);
    connFact.setIntProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
    connFact.setIntProperty(XMSC_WMQ_CHANNEL, "XMS.SVR.CHL");
```

```
conn = connFact.createConnection();
```

```
    // Other code here
  }
  catch(xms::Exception & ex)
  {
    // Error handling code here

    nRC = -1;
  }

  return(nRC);
}
```

If an application uses this technique to catch XMS exceptions, it must catch an exception by reference, and not by value. This ensures that an exception is not sliced and that 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++ runtime environment, an application can specify the std::exception class in the exception declaration of the catch construct. The following code fragment illustrates this concept:

```
#include <xms.hpp>

using namespace std;

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

  try
  {
    xms::ConnectionFactory connFact;

    connFact.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
    connFact.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
    connFact.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
    connFact.setIntProperty(XMSC_WMQ_PORT, 1506);
    connFact.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
    connFact.setStringProperty(XMSC_WMQ_CHANNEL, "XMS.SVR.CHL");


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

    nRC = -1;
  }

  return(nRC);
}
```

After an application catches an XMS exception, it 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 288. 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 61.

XMS can create an exception for each error it detects during a call and link the exceptions to form a chain. After an application has caught the first exception, it 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.

Because the getLinkedException() method returns a pointer to a linked exception, the application must 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.

Message and exception listeners in C++

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

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 of the message listener, and the application must ensure that the message listener is still available when XMS calls the onMessage() method.

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 requires concurrent delivery of messages, see "Asynchronous message delivery" on page 31.

To stop the asynchronous delivery of messages to a message consumer, the application can call the setMessageListener() method again, by 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.

A new message listener can be registered with a message consumer without cancelling the registration of an existing message listener. If the onMessage() method of an existing message listener is running when a new message listener is registered, the active method completes normally, and any subsequent messages are processed by calls to the onMessage() method of the new message listener. If a transaction is in progress when a message listener is changed, the transaction is completed by calls to the onMessage() method of the new message listener.

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

```
#include <xms.hpp>

using namespace std;

class MyMsgListener : public xms::MessageListener
{
public:

  xmsVOID onMessage(const xms::Message * pMsg);
};
```

```
-------------------------------------------

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

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

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

    delete pMsg;
  }
}
```

Because XMS delivers a pointer to a message when it calls the onMessage() method, the application must release the message using the 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_CONNECTION_TYPE, XMSC_CT_WMQ);
    cf.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
    cf.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
    cf.setIntProperty(XMSC_WMQ_PORT, 1414);
    cf.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
    cf.setStringProperty(XMSC_WMQ_CHANNEL, "XMS.SVR.CHL");


    conn    = cf.createConnection();
    sess    = conn.createSession();
    dest    = xms::Destination(XMS_DESTINATION_TYPE_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);
}
```

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 of 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, by 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.

Because XMS passes a pointer to an exception when it calls the onException() method, the application must release the exception by using the C++ delete operator.

Use of C APIs 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;

  cf.setIntProperty(XMSC_CONNECTION_TYPE, XMSC_CT_WMQ);
  cf.setIntProperty(XMSC_WMQ_CONNECTION_MODE, XMSC_WMQ_CM_CLIENT);
  cf.setStringProperty(XMSC_WMQ_HOST_NAME, "localhost");
  cf.setIntProperty(XMSC_WMQ_PORT, 1414);
  cf.setStringProperty(XMSC_WMQ_QUEUE_MANAGER, "TESTQM");
  cf.setStringProperty(XMSC_WMQ_CHANNEL, "XMS.SVR.CHL");


  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 does that it can no longer use the object using the C++ API.

Being able to use the C API is useful because some functions are available only in the C API. An example of such a function is described in "C functions that return a string or byte array by reference" on page 60.

Chapter 8. Working with administered objects

This chapter provides information about administered objects. XMS applications can retrieve object definitions from a central administered objects repository, and use them to create connection factories and destinations.

About this task

This chapter provides information to help with creating and managing administered objects, describing the types of administered object repository that XMS supports. The chapter also explains how an XMS application makes a connection to an administered objects repository to retrieve the required administered objects.

The chapter contains the following sections:

- "Supported types of administered object repository"
- "Property mapping for administered objects" on page 78
- "Required properties for administered ConnectionFactory objects" on page 78
- "Required properties for administered Destination objects" on page 79
- "Creating administered objects" on page 80
- "InitialContext objects" on page 81
- "InitialContext properties" on page 81
- "URI format for XMS initial contexts" on page 81
- "JNDI Lookup Web service" on page 83
- "Retrieval of administered objects" on page 83

Supported types of administered object repository

XMS supports two types of administered object directory: File System and Lightweight Directory Access Protocol (LDAP).XMS supports three types of administered object directory: File System, Lightweight Directory Access Protocol (LDAP), and COS naming.

File System object directories take the form of serialized Java and Naming Directory Interface (JNDI) objects. LDAP object directories are directories that contain JNDI objects. File System and LDAP object directories can both be administered using the JMSAdmin tool available with IBM MQ. Both these object directories can be used to administer client connections by centralizing IBM MQ connection factories and destinations. This allows the network administrator to deploy multiple applications that all refer to the same central repository, and that are automatically updated to reflect changes to connection settings made in the central repository.

A COS naming directory contains WebSphere service integration bus connection factories and destinations and can be administered using the WebSphere Application Server administrative console. In order for an XMS application to be able to retrieve objects from the COS naming directory, a JNDI lookup Web service must be deployed. This Web service is not available on all WebSphere service integration technologies. Refer to the product documentation for details.

Note: It is necessary to restart application connections for changes to the object directory to take effect.

Property mapping for administered objects

To enable applications to use IBM MQ JMS and WebSphere Application Server connection factory and destination object definitions, the properties retrieved from these definitions must be mapped on to the corresponding XMS properties that can be set on XMS connection factories and destinations.

In order to create, for example, an XMS connection factory with properties retrieved from a IBM MQ JMS connection factory, the properties must be mapped between the two.

All property mappings are performed automatically.

The following table demonstrates the mappings between some of the most common properties of connection factories and destinations. The properties shown in this table are just a small set of examples, and not all properties shown are relevant to all connection types and servers.

Table 14. Examples of name mapping for connection factory and destination properties

IBM MQ JMS property name	**XMS property name**
PERSISTENCE (PER)	XMSC_DELIVERY_MODE
EXPIRY (EXP)	XMSC_TIME_TO_LIVE
PRIORITY (PRI)	XMSC_PRIORITY

Table 15. Examples of name mapping for connection factory and destination properties

IBM MQ JMS property name	**XMS property name**	**WebSphere service integration bus property name**
PERSISTENCE (PER)	XMSC_DELIVERY_MODE	
EXPIRY (EXP)	XMSC_TIME_TO_LIVE	
PRIORITY (PRI)	XMSC_PRIORITY	
	XMSC_WPM_HOST_NAME	serverName
	XMSC_WPM_BUS_NAME	busName
	XMSC_WPM_TOPIC_SPACE	topicName

Required properties for administered ConnectionFactory objects

When an application creates a connection factory, a number of properties must be defined to create a connection to a messaging server.

The properties listed in the following tables are the minimum required for an application to set to create a connection to a messaging server. If you want to customize the way that a connection is created, then your application can set any additional properties of the ConnectionFactory object as necessary. For further information, and a complete list of available properties, see “Properties of ConnectionFactory” on page 402.

Connection to a IBM MQ queue manager

Table 16. Property settings for administered ConnectionFactory objects for connections to a IBM MQ queue manager

Required XMS	Equivalent IBM MQ JMS property required
XMSC_CONNECTION_TYPE	XMS works this out from the connection factory class name and TRANSPORT (TRAN) property.
XMSC_WMQ_HOST_NAME	HOSTNAME (HOST)
XMSC_WMQ_PORT	PORT

Connection to a WebSphere service integration bus

Table 18. Property settings for administered ConnectionFactory objects for connections to a WebSphere service integration bus

XMS property	Description
XMSC_CONNECTION_TYPE	The type of messaging server to which an application connects. This is determined from the connection factory class name.
XMSC_WPM_BUS_NAME	For a connection factory, the name of the service integration bus that the application connects to or, for a destination, the name of the service integration bus in which the destination exists.

Required properties for administered Destination objects

An application that creates a Destination object must set several properties as compared to an application using an administered Destination object.

Table 19. IBM MQ JMS property settings for administered Destination objects

Type of connection	Property	Description
IBM MQ queue manager	QUEUE (QU)	The queue that you wish to connect to
	TOPIC (TOP)	The topic that the application uses as a destination

Table 20. Property settings for administered Destination objects

Type of connection	Property	Description
IBM MQ queue manager	QUEUE (QU)	The queue that you wish to connect to
	TOPIC (TOP)	The topic that the application uses as a destination
WebSphere service integration bus	topicName	If your application is connecting to a topic
	queueName	If your application is connecting to a queue

Creating administered objects

The ConnectionFactory and Destination object definitions that XMS applications require to make a connection to a messaging server must be created using the appropriate administrative tools.

Before you begin

For further details about the different types of administered object repository that XMS supports, see "Supported types of administered object repository" on page 77.

About this task

To create the administered objects for IBM MQ, use the IBM MQ Explorer or IBM MQ JMS administration (JMSAdmin) tool.

To create the administered objects for IBM MQ, use the IBM MQ JMS administration (JMSAdmin) tool.

To create administered objects for WebSphere service integration bus, use the WebSphere Application Server administrative console.

The following steps summarize what you do to create administered objects.

Procedure

1. Create a connection factory and define the properties needed to create a connection from your application to your chosen server. The minimum properties that XMS requires to make a connection are defined in "Required properties for administered ConnectionFactory objects" on page 78.
2. Create the required destination on the messaging server to which your application will connect:
   - For a connection to a IBM MQ queue manager, create a queue or topic.
   - For a connection to a WebSphere service integration bus, create a queue or a topic.

   The minimum properties that XMS requires to make a connection are defined in "Required properties for administered Destination objects" on page 79.

InitialContext objects

An application must create an initial context to be used to make a connection to the administered objects repository to retrieve the required administered objects.

About this task

An InitialContext object encapsulates a connection to the repository. The XMS API provides methods to perform the following tasks:

- Create an InitialContext object.
- Delete the InitialContext object when it is no longer required (applies to C and C++ only).
- Lookup an administered object in the administered object repository.

For further details about creating an InitialContext object, see "InitialContext" on page 145 for C, "InitialContext" on page 293 for C++ and "Properties of InitialContext" on page 408.

InitialContext properties

The parameters of the InitialContext constructor include the location of the repository of administered objects, given as a uniform resource indicator (URI). In order for an application to establish a connection to the repository, it may be necessary to provide more information than the information contained in the URI.

In JNDI implementation of XMS, the additional information is provided in an environment Hashtable to the constructor.

In the C and C++ implementations of XMS, the information is provided by setting properties on the InitialContext object after it has been constructed. For C and C++, therefore, the creation of the InitialContext object and the connection to the directory (for the lookup) are done separately so that the properties can be set on the InitialContext object before an application connects to the directory to retrieve administered objects.

The location of the administered object repository is defined in the XMSC_IC_URL property. This property is typically passed on the Create call, but can be modified to connect to a different naming directory before the lookup. For FileSystem or LDAP contexts, this property defines the address of the directory. For COS naming, this is the address of the Web service that uses these properties to connect to the JNDI directory.

The following properties are passed unmodified to the Web service which will use them to use to connect to the JNDI directory.

- XMSC_IC_PROVIDER_URL
- XMSC_IC_SECURITY_CREDENTIALS
- XMSC_IC_SECURITY_AUTHENTICATION
- XMSC_IC_SECURITY_PRINCIPAL
- XMSC_IC_SECURITY_PROTOCOL

URI format for XMS initial contexts

The location of the repository of administered objects is provided as a uniform resource indicator (URI). The format of the URI depends on the context type.

FileSystem context

For the FileSystem context, the URL gives the location of the file system based directory. The structure of the URL is as defined by RFC 1738, *Uniform Resource Locators (URL)*: the URL has the prefix `file://`, and the syntax following this prefix is a valid definition of a file that can be opened on the system on which XMS is running.

This syntax can be platform-specific, and can use either '/ separators or '\' separators. If you use '\', then each separator needs to be escaped by using an additional '\'. This prevents the C runtime from trying to interpret the separator as an escape character for what follows. Furthermore, if the URI is coded as literal C strings in source code, the compiler also requires each '\' character to be escaped.

These examples illustrate this syntax:

```
file://myBindings
file:///admin/.bindings
file://\\admin\\.bindings
file://c:/admin/.bindings
file://c:\\admin\\.bindings
file://\\\\madison\\shared\\admin\\.bindings
file:///usr/admin/.bindings
```

The following examples show the syntax written as literal C strings within source code:

```
"file://c:\\\\admin\\\\.bindings"
"file://\\\\\\\\madison\\\\shared\\\\admin\\\\.bindings"
```

LDAP context

For the LDAP context, the basic structure of the URL is as defined by RFC 2255, *The LDAP URL Format*, with the case-insensitive prefix `ldap://`

The precise syntax is illustrated in the following example:

```
LDAP://[Hostname][:Port][ "/" [DistinguishedName]]
```

This syntax is as defined in the RFC but without support for any attributes, scope, filters, or extensions.

Examples of this syntax include:

```
ldap://madison:389/cn=JMSData,dc=IBM,dc=UK
ldap://madison/cn=JMSData,dc=IBM,dc=UK
LDAP:///cn=JMSData,dc=IBM,dc=UK
```

WSS context

For the WSS context, the URL is in the form of a Web services endpoint, with the prefix `http://`.

Alternatively, you can use the prefix `cosnaming://` or `wsvc://`.

These two prefixes are interpreted as meaning that you are using a WSS context with the URL accessed over http, which enables the initial context type to be derived easily directly from the URL.

Examples of this syntax include the following:

```
http://madison.ibm.com:9080/xmsjndi/services/JndiLookup
cosnaming://madison/jndilookup
```

JNDI Lookup Web service

To access a COS naming directory from XMS, a JNDI Lookup Web service must be deployed on a WebSphere service integration bus server. This Web service translates the Java information from the COS naming service into a form that XMS applications can read.

The Web service is provided in the enterprise archive file SIBXJndiLookupEAR.ear, located within the install directory. This can be installed within a WebSphere service integration bus server by using either the administrative console or the wsaadmin scripting tool. Refer to the product documentation for further information on deploying Web service applications.

To define the Web service within XMS applications, you simply need to set the XMSC_IC_URL property of the InitialContext object to the Web service endpoint URL. For example, if the Web service is deployed on a server host called MyServer, an example of a Web service endpoint URL:

```
wsvc://MyHost:9080/SIBXJndiLookup/services/JndiLookup
```

Setting the XMSC_IC_URL property allows InitialContext Lookup calls to invoke the Web service at the defined endpoint, which in turn looks up the required administered object from the COS naming service.

C and C++ applications can use the Web service.

Retrieval of administered objects

XMS retrieves an administered object from the repository using the address provided when the InitialContext object is created, or in the InitialContext properties.

Objects to be retrieved can have the following types of names:

- A simple name describing the Destination object, for example, a queue destination called SalesOrders
- A composite name, which can be made up of SubContexts, separated by '/', and it must end with the object name. An example of a composite name is "Warehouse/PickLists/DispatchQueue2" where Warehouse and Picklists are SubContexts in the naming directory, and DispatchQueue2 is the name of a Destination object.

Chapter 9. Securing communications for XMS applications

This chapter provides information about setting up secure communications to enable XMS applications to connect via Secure Sockets Layer (SSL) to a WebSphere service integration bus messaging engine or IBM MQ queue manager.

About this task

This chapter provides information about configuring XMS ConnectionFactory properties to enable applications to make secure connections.

The chapter contains the following sections:

- "Secure connections to a IBM MQ queue manager"
- "CipherSuite and CipherSpec name mappings for connections to a WebSphere MQ queue manager" on page 86
- "Secure connections to a WebSphere service integration bus messaging engine" on page 88
- "CipherSuite and CipherSpec name mappings for connections to a WebSphere service integration bus" on page 90

Secure connections to a IBM MQ queue manager

To enable an XMS C or C++ application to make secure connections to a IBM MQ queue manager, the relevant properties must be defined in the ConnectionFactory object.To enable an XMS C or C++ application to make secure connections to a IBM MQ queue manager, the relevant properties must be defined in the ConnectionFactory object.

The protocol used in the encryption negotiation can be either Secure Sockets Layer (SSL) or Transport Layer Security (TLS), depending on which CipherSuite you specify in the ConnectionFactory object.

If you use the IBM MQ Version 7.0.0.1 and above client libraries and connect to a WebSphere MQ Version 7 queue manager, then you can create multiple connections to same queue manager in XMS application. However connection to different queue manager is not permitted. If you attempt you get the `MQRC_SSL_ALREADY_INITIALIZED` error.

If you use the IBM MQ Version 6 and above client libraries, then you can create a new SSL connection only if you close any previous SSL connection first. Multiple concurrent SSL connections from the same process to the same or different queue managers are not permitted. If you attempt more than one request, you get the warning `MQRC_SSL_ALREADY_INITIALIZED`, which might mean that some requested parameters for the SSL connection were ignored.

ConnectionFactory properties for connections via SSL to a IBM MQ manager, with a brief description, are shown in the following table:

Table 21. Properties of ConnectionFactory for connections to a IBM MQ queue manager via SSL

Name of property	Description
XMSC_WMQ_SSL_CERT_STORES	The locations of the servers that hold the certificate revocation lists (CRLs) to be used on an SSL connection to a queue manager.
XMSC_WMQ_SSL_CIPHER_SPEC	The name of the cipher spec to be used on a secure connection to a queue manager.
XMSC_WMQ_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on an SSL or TLS connection to a queue manager. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WMQ_SSL_CRYPTO_HW	Configuration details for the cryptographic hardware connected to the client system.
XMSC_WMQ_SSL_FIPS_REQUIRED	The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.
XMSC_WMQ_SSL_KEY_REPOSITORY	The location of the key database file in which keys and certificates are stored.
XMSC_WMQ_SSL_KEY_RESETCOUNT	The KeyResetCount represents the total number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated.
XMSC_WMQ_SSL_PEER_NAME	The peer name to be used on an SSL connection to a queue manager.
"XMSC_WMQ_SSL_ENCRYPTION_POLICY_SUITE_B" on page 461	The value of this property determines whether an application can use the Suite B compliant cipher suites.

CipherSuite and CipherSpec name mappings for connections to a IBM MQ queue manager

The InitialContext translates between the JMSAdmin Connection Factory property SSLCIPHERSUITE and the XMS near-equivalent XMSC_WMQ_SSL_CIPHER_SPEC. A similar translation is necessary if you specify a value for XMSC_WMQ_SSL_CIPHER_SUITE but omit value for XMSC_WMQ_SSL_CIPHER_SPEC.

Table 22 lists the available CipherSpecs and their JSSE CipherSuite equivalents.

Table 22. Available CipherSpecs and their JSSE CipherSuite equivalents

CipherSpec	Equivalent JSSE CipherSuite	Fips	SuiteB 128 bit	SuiteB 192 bit
DES_SHA_EXPORT	SSL_RSA_WITH_DES_CBC_SHA	No	No	No
DES_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA	No	No	No
FIPS_WITH_3DES_EDE_CBC_SHA	SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	No	No	No
FIPS_WITH_DES_CBC_SHA	SSL_RSA_FIPS_WITH_DES_CBC_SHA	No	No	No
NULL_MD5	SSL_RSA_WITH_NULL_MD5	No	No	No
NULL_SHA	SSL_RSA_WITH_NULL_SHA	No	No	No
RC2_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	No	No	No
RC4_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC4_40_MD5	No	No	No

Table 22. Available CipherSpecs and their JSSE CipherSuite equivalents (continued)

CipherSpec	Equivalent JSSE CipherSuite	Fips	SuiteB 128 bit	SuiteB 192 bit
RC4_MD5_US	SSL_RSA_WITH_RC4_128_MD5	No	No	No
RC4_SHA_US	SSL_RSA_WITH_RC4_128_SHA	No	No	No
TLS_RSA_WITH_3DES_EDE_CBC_SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA	Yes	No	No
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA	Yes	No	No
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA	Yes	No	No
TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	No	No	No
TRIPLE_DES_SHA_US	SSL_RSA_WITH_3DES_EDE_CBC_SHA	No	No	No
ECDHE_ECDSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	Yes	No	No
ECDHE_ECDSA_AES_128_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	Yes	No	No
ECDHE_ECDSA_AES_128_GCM_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	Yes	Yes	No
ECDHE_ECDSA_AES_256_CBC_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	Yes	No	No
ECDHE_ECDSA_AES_256_GCM_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	Yes	No	Yes
ECDHE_ECDSA_NULL_SHA256	SSL_ECDHE_ECDSA_WITH_NULL_SHA	No	No	No
ECDHE_ECDSA_RC4_128_SHA256	SSL_ECDHE_ECDSA_WITH_RC4_128_SHA	No	No	No
ECDHE_RSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	Yes	No	No
ECDHE_RSA_AES_128_CBC_SHA256	SSL_ECDHE_RSA_WITH_AES_128_CBC_SHA256	Yes	No	No
ECDHE_RSA_AES_128_GCM_SHA256	SSL_ECDHE_RSA_WITH_AES_128_GCM_SHA25	Yes	No	No
ECDHE_RSA_AES_256_CBC_SHA384	SSL_ECDHE_RSA_WITH_AES_256_CBC_SHA384	Yes	No	No
ECDHE_RSA_AES_256_GCM_SHA384	SSL_ECDHE_RSA_WITH_AES_256_GCM_SHA384	Yes	No	No
ECDHE_RSA_NULL_SHA256	SSL_ECDHE_RSA_WITH_NULL_SHA	No	No	No
ECDHE_RSA_RC4_128_SHA256	SSL_ECDHE_RSA_WITH_RC4_128_SHA	No	No	No
TLS_RSA_WITH_AES_128_CBC_SHA256	SSL_RSA_WITH_AES_128_CBC_SHA256	Yes	No	No
TLS_RSA_WITH_AES_128_GCM_SHA256	SSL_RSA_WITH_AES_128_GCM_SHA256	Yes	No	No
TLS_RSA_WITH_AES_256_CBC_SHA256	SSL_RSA_WITH_AES_256_CBC_SHA256	Yes	No	No
TLS_RSA_WITH_AES_256_GCM_SHA384	SSL_RSA_WITH_AES_256_GCM_SHA384	Yes	No	No
TLS_RSA_WITH_NULL_SHA256	SSL_RSA_WITH_NULL_SHA256	No	No	No
TLS_RSA_WITH_RC4_128_SHA256	SSL_RSA_WITH_RC4_128_SHA	No	No	No

Note: A one-to-one mapping for the CipherSuite name SSL_RSA_WITH_3DES_EDE_CBC_SHA or SSL_RSA_WITH_DES_CBC_SHA must account for the setting of the property XMSC_WMQ_SSL_FIPSREQUIRED and apply an heuristic.

Note: For additional information about these values, see IBM MQ documentation.

If you specify SSL_RSA_WITH_3DES_EDE_CBC_SHA or SSL_RSA_WITH_DES_CBC_SHA for the property XMSC_WMQ_SSL_CIPHER_SUITE, and there is no value for XMSC_WMQ_SSL_CIPHER_SPEC, a value for XMSC_WMQ_SSL_CIPHER_SPEC is chosen according to the following tables.

The values used for XMSC_WMQ_SSL_CIPHER_SPEC when you specify SSL_RSA_WITH_3DES_EDE_CBC_SHA for the XMSC_WMQ_SSL_CIPHER_SUITE property are shown in the following table:

Table 23. Values used for XMSC_WMQ_SSL_CIPHER_SPEC when you specify SSL_RSA_WITH_3DES_EDE_CBC_SHA for the XMSC_WMQ_SSL_CIPHER_SUITE property

Input: XMSC_WMQ_SSL_FIPSREQUIRED value	**Output: XMSC_WMQ_SSL_CIPHER_SPEC chosen**
xmsFALSE (that is, MQSSL_FIPS_NO)	TRIPLE_DES_SHA_US
xmsTRUE (that is, MQSSL_FIPS_YES)	TLS_RSA_WITH_3DES_EDE_CBC_SHA

The values used for XMSC_WMQ_SSL_CIPHER_SPEC when you specify SSL_RSA_WITH_DES_CBC_SHA for the XMSC_WMQ_SSL_CIPHER_SUITE property are shown in the following table:

Table 24. Values used for XMSC_WMQ_SSL_CIPHER_SPEC when you specify SSL_RSA_WITH_DES_CBC_SHA for the XMSC_WMQ_SSL_CIPHER_SUITE property

Input: XMSC_WMQ_SSL_FIPSREQUIRED value	**Output: XMSC_WMQ_SSL_CIPHER_SPEC chosen**
xmsFALSE (that is, MQSSL_FIPS_NO)	DES_SHA_EXPORT
xmsTRUE (that is, MQSSL_FIPS_YES)	TLS_RSA_WITH_DES_CBC_SHA

Secure connections to a WebSphere service integration bus messaging engine

To enable an XMS C/C++ application to make secure connections to a WebSphere service integration bus messaging engine, the relevant properties must be defined in the ConnectionFactory object.

XMS provides SSL and HTTPS support for connections to a WebSphere service integration bus. SSL and HTTPS provide secure connections for authentication and confidentiality.

Like WebSphere security, XMS security is configured with respect to JSSE security standards and naming conventions, which include the use of CipherSuites to specify the algorithms that are used when negotiating a secure connection. The protocol used in the encryption negotiation can be either SSL or TLS, depending on which CipherSuite you specify in the ConnectionFactory object.

The security capabilities for XMS C/C++ application are provided by IBM's standard security enablement component, Global Security Kit (GSKit). XMS configures the relevant GSKit options by means of properties set on the XMS ConnectionFactory object. These properties must be specified regardless of whether the ConnectionFactory object is an administered object.

Table 25 on page 89 lists the properties that must be defined in the ConnectionFactory object.

Table 25. Properties of ConnectionFactory for secure connections to a WebSphere service integration bus messaging engine

Name of property	Description
XMSC_WPM_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on an SSL or TLS connection to a WebSphere service integration bus messaging engine. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WPM_SSL_KEY_REPOSITORY	A path to the file that is the keyring file containing the public or private keys to be used in the secure connection.
XMSC_WPM_SSL_KEYRING_LABEL	The certificate to be used when authenticating with the server.
XMSC_WPM_SSL_KEYRING_PW	The password for the keyring file.
XMSC_WPM_SSL_KEYRING_STASH_FILE	The name of a binary file containing the password of the key repository file.
XMSC_WPM_SSL_FIPS_REQUIRED	The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.
"XMSC_WPM_SSL_ENCRYPTION_POLICY_SUITE_B" on page 472	The value of this property determines whether an application can use the Suite B compliant cipher suites.

Note: You must specify the XMSC_WPM_SSL_CIPHER_SUITE properties for all applications, and the XMSC_WPM_SSL_KEY_REPOSITORY properties for C/C++ applications only. You can specify all the other properties listed in the table according to requirements.

The following is an example of ConnectionFactory properties for secure connections to a WebSphere integration messaging engine:

```
cf.setStringProperty(XMSC_WPM_PROVIDER_ENDPOINTS, host_name:port_number:chain_name);
cf.setStringProperty(XMSC_WPM_SSL_KEY_REPOSITORY, key_repository_pathname);
cf.setStringProperty(XMSC_WPM_TARGET_TRANSPORT_CHAIN, transport_chain);
cf.setStringProperty(XMSC_WPM_SSL_CIPHER_SUITE, cipher_suite);
cf.setStringProperty(XMSC_WPM_SSL_KEYRING_STASH_FILE, stash_file_pathname);
```

Where chain_name should be set to either BootstrapTunneledSecureMessaging or BootstrapSecureMessaging, and port_number is the number of the port on which the bootstrap server listens for incoming requests.

The following is an example of ConnectionFactory properties for secure connections to a WebSphere integration messaging engine with sample values inserted:

```
/* CF properties needed for an SSL connection */
  cf.setStringProperty(XMSC_WPM_PROVIDER_ENDPOINTS,"localhost:7286:BootstrapSecureMessaging");
  cf.setStringProperty(XMSC_WPM_TARGET_TRANSPORT_CHAIN,"InboundSecureMessaging");
  cf.setStringProperty(XMSC_WPM_SSL_KEY_REPOSITORY,"C:\\Program Files\\IBM\\gsk7\\bin\\XMSkey.kdb");
  cf.setStringProperty(XMSC_WPM_SSL_KEYRING_STASH_FILE,"C:\\Program Files\\IBM\\gsk7\\bin\\XMSkey.sth");
  cf.setStringProperty(XMSC_WPM_SSL_CIPHER_SUITE,"SSL_RSA_EXPORT_WITH_RC4_40_MD5");
```

CipherSuite and CipherSpec name mappings for connections to a WebSphere service integration bus

Because GSKit uses CipherSpecs rather than CipherSuites, the JSSE-style CipherSuite names specified in the XMSC_WPM_SSL_CIPHER_SUITE property must be mapped to the GSKit-style CipherSpec names.

Table 26 lists the equivalent CipherSpec for each recognized CipherSuite.

Table 26. Available CipherSuites and their equivalent CipherSpecs

CipherSuite	CipherSpec equivalent
SSL_RSA_WITH_NULL_MD5	NULL_MD5
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RC4_MD5_EXPORT
SSL_RSA_WITH_RC4_128_MD5	RC4_MD5_US
SSL_RSA_WITH_NULL_SHA	NULL_SHA
SSL_RSA_WITH_RC4_128_SHA	RC4_SHA_US
SSL_RSA_WITH_DES_CBC_SHA	DES_SHA_EXPORT
SSL_RSA_FIPS_WITH_DES_CBC_SHA	FIPS_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA	TRIPLE_DES_SHA_US
SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	FIPS_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA	TLS_RSA_WITH_DES_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA	TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA	TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_RSA_WITH_NULL_SHA256	TLS_RSA_WITH_NULL_SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS_ECDHE_RSA_WITH_RC4_128_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_RC4_128_SHA	TLS_RSA_WITH_RC4_128_SHA

Chapter 10. XMS messages

This sectionchapter describes the structure and content of XMS messages and explains how applications process XMS messages.

This chapter contains the following sections:

v "Parts of an XMS message"

v "Header fields in an XMS message"

v "Properties of an XMS message" on page 92 v

"The body of an XMS message" on page 95 v

"Message selectors" on page 100

v "Mapping XMS messages onto IBM MQ messages" on page 101

Parts of an XMS message

An XMS message consists of a header, a set of properties, and a body.

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."

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, see "Properties of an XMS message" on page 92.

Body The body of a message contains application data. For more information, see "The body of an XMS message" on page 95.

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 100.

Header fields in an XMS message

To allow an XMS application to exchange messages with a WebSphere 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 164 for C, or "Message" on page 311 for C++. A header field is always readable and writable.

Table 27 lists the JMS message header fields and indicates how the value of each field is set for a transmitted message. 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 27. 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]
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

XMS supports three kinds of message property: JMS defined properties, IBM defined properties, and application-defined properties.

An XMS application can exchange messages with a WebSphere JMS application because XMS supports the following predefined properties of a Message object:

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

Each predefined property has two names:

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

  This is the name by which the property is known in JMS or WebSphere JMS, and it 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 to identify the property in all situations except in a message selector expression. Each XMS name is defined as a named constant in one of the header files, `xmsc.h`, `xmsc_wmq.h`, or `xmsc_wpm.h`. The value of the named constant is the corresponding JMS or WebSphere 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*.

For information about getting and setting the properties of messages, see "Object Properties in C" on page 58 or "Properties in C++" on page 67.

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.

The received message, when forwarded after clearing up the message properties, will behave in a manner consistent with the behavior of forwarding a standard WMQ XMS for C/C++ BytesMessage with message properties cleared up.

This is, however, not recommended since the following properties will be lost:

- JMS_IBM_Encoding property value, implying that the message data cannot be decoded meaningfully.
- JMS_IBM_Format property value, implying that the header chaining between the (MQMD or the new MQRFH2) message header and existing headers would be broken.

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 it can 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 45.

JMS-defined properties of a message

Several JMS-defined properties of a message are supported by both XMS and WebSphere JMS.

Table 28 lists the JMS-defined properties of a message that are supported by both XMS and WebSphere JMS. For a description of the JMS-defined properties, see *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. 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 28. 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	Send [MessageProducer]
JMSX_DELIVERY_COUNT	JMSXDeliveryCount	xmsINT	Receive [MessageConsumer]
JMSX_GROUPID	JMSXGroupID	String	Set String Property [PropertyContext]
JMSX_GROUPSEQ	JMSXGroupSeq	xmsINT	Set Integer Property [PropertyContext]
JMSX_USERID	JMSXUserID	String	Send [MessageProducer]

IBM-defined properties of a message

Several IBM-defined properties of a message are supported by XMS and WebSphere JMS.

Table 29 lists the IBM defined properties of a message that are supported by both XMS and WebSphere JMS. For more information about the IBM-defined properties, see *IBM MQ Using Java* or the WebSphere Application Server Information Center.

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

Table 29. IBM-defined properties of a message

XMS name of the IBM defined property	**WebSphere JMS name**	**Data type**	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMS_IBM_CHARACTER_SET	JMS_IBM_Character_Set	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_ENCODING	JMS_IBM_Encoding	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_EXCEPTIONMESSAGE	JMS_IBM_ExceptionMessage	String	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONREASON	JMS_IBM_ExceptionReason	xmsINT	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONTIMESTAMP	JMS_IBM_ExceptionTimestamp	xmsLONG	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONPROBLEM DESTINATION	JMS_IBM_ExceptionProblemDestination	String	Receive [MessageConsumer]
JMS_IBM_FEEDBACK	JMS_IBM_Feedback	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_FORMAT	JMS_IBM_Format	String	Set String Property [PropertyContext]
JMS_IBM_LAST_MSG_IN_GROUP	JMS_IBM_Last_Msg_In_Group	xmsBOOL	Set Integer Property [PropertyContext]
JMS_IBM_MSGTYPE	JMS_IBM_MsgType	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_PUTAPPLTYPE	JMS_IBM_PutApplType	xmsINT	Send [MessageProducer]
JMS_IBM_PUTDATE	JMS_IBM_PutDate	String	Send [MessageProducer]
JMS_IBM_PUTTIME	JMS_IBM_PutTime	String	Send [MessageProducer]
JMS_IBM_REPORT_COA	JMS_IBM_Report_COA	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_COD	JMS_IBM_Report_COD	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_DISCARD_MSG	JMS_IBM_Report_Discard_Msg	xmsINT	Set Integer Property [PropertyContext]

Table 29. IBM-defined properties of a message (continued)

XMS name of the IBM defined property	WebSphere JMS name	Data type	How the value is set for a transmitted message (in the format *method* [*class*])
JMS_IBM_REPORT_EXCEPTION	JMS_IBM_Report_Exception	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_EXPIRATION	JMS_IBM_Report_Expiration	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_NAN	JMS_IBM_Report_NAN	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PAN	JMS_IBM_Report_PAN	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PASS_CORREL_ ID	JMS_IBM_Report_Pass_Correl_ID	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PASS_MSG_ID	JMS_IBM_Report_Pass_Msg_ID	xmsINT	Set Integer Property [PropertyContext]
JMS_IBM_SYSTEM_MESSAGEID	JMS_IBM_System_MessageID	String	Send [MessageProducer]

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 WebSphere JMS application to select and process messages sent by an XMS application, the name of an application-defined property must conform to the rules for forming identifiers in message selector expressions, as documented in *IBM MQ Using Java*. The value of an application-defined property must have one of the following data types: xmsBOOL, xmsSBYTE, 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.

XMS supports five 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 for C or C++ contains the methods to process the body of a bytes message. For more information, see "Bytes messages" on page 97.

Map The body contains a set of name-value pairs, where each value has an associated data type. A message with this type of body is called a *map message*. The MapMessage class for C or C++contains the methods to process the body of a map message. For more information, see "Map messages" on page 98.

Object
The body contains a serialized Java object. A message with this type of

body is called an *object message*. The ObjectMessage class for C or C++contains the methods to process the body of an object message. For more information, see "Object messages" on page 98.

Stream
The body contains a stream of values, where each value has an associated data type. A message with this type of body is called a *stream message*. The StreamMessage class for C or C++contains the methods to process the body of a stream message. For more information, see "Stream messages" on page 99.

Text
The body contains a string. A message with this type of body is called a *text message*. The TextMessage class for C or C++ and contains the methods to process the body of a text message. For more information, see "Text messages" on page 100.

In the C interface, XMS returns a message handle to an application when the application creates a 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, MapMessage, ObjectMessage, StreamMessage, or TextMessage 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 an error.

A C application can call the xmsMsgGetTypeId() function to determine the body type of a message. The function returns one of the following values:

XMS_MESSAGE_TYPE_BASE
If the message has no body

XMS_MESSAGE_TYPE_BYTES
If the message is a bytes message

XMS_MESSAGE_TYPE_MAP
If the message is a map message

XMS_MESSAGE_TYPE_OBJECT
If the message is an object message

XMS_MESSAGE_TYPE_STREAM
If the message is a stream message

XMS_MESSAGE_TYPE_TEXT
If the message is a text message

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

For information about the size and maximum and minimum values of each of these data types, see Table 8 on page 42.

For more information about the required data types for elements of application data written in the body of a message and about the five types of body message, see the subtopics.

Data types for elements of application data

To ensure that an XMS application can exchange messages with a IBM MQ JMS application, both the applications 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 30. For each XMS data type, the table shows the compatible Java data type. XMS provides the methods to write elements of application data only with these data types.

Table 30. XMS data types that are compatible with Java data types

XMS data type	Represents	Compatible Java data type
xmsBOOL	The boolean value xmsTRUE or xmsFALSE	boolean
xmsCHAR16	Double byte character	char
xmsSBYTE	Signed 8-bit integer	byte
xmsSHORT	Signed 16-bit integer	short
xmsINT	Signed 32-bit integer	int
xmsLONG	Signed 64-bit integer	long
xmsFLOAT	Signed floating point number	float
xmsDOUBLE	Signed double precision floating point number	double
String	String of characters	String

For information about the size, maximum value and minimum value of each of these data types, see "XMS primitive types" on page 42.

Bytes messages

The body of a bytes message contains a stream of bytes. The body contains only the actual data, and it is 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 (for C or C++). 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 repeatedly.

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 or IBytesMessage interface 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.

In the case of C only, an application can skip over bytes without reading them by calling a read function with a null pointer for the `value` parameter or by calling xmsBytesMsgReadBytes(). For information about how to skip over a string, see "xmsBytesMsgReadUTF – Read UTF String" on page 122.

If an application calls the Reset method of the BytesMessage class or IBytesMessage interface 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 for C or C++ 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, where each value has an associated data type.

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 30 on page 97. 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, a C or C++ 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 it can 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. CONFIRM THAT WHOLE OF THIS APPLIES TO C/C++

When an application gets the value of a name-value pair, the value can be converted by XMS into another data type. For example, to get an integer from the body of a map message, an application can call the Get String method of the MapMessage class, which returns the integer as a string. The supported conversions are the same as those that are supported when XMS converts a property value from one data type to another. For more information about the supported conversions, see "Implicit conversion of a property value from one data type to another" on page 43.

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.

Object messages

The body of an object message contains a serialized Java object.

An XMS application can receive an object message, change its header fields and properties, and then send it to another destination. An application can also copy the body of an object message and use it to form another object message. XMS treats the body of an object message as an array of bytes.

After an application creates an object 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 an object message, the body of the message is read-only. If an application calls the Clear Body method of the Message class for C or C++ when the body of an object message is read-only, the body becomes readable and writable. The method also clears the body.

Stream messages

The body of a stream message contains a stream of values, where each value has an associated data type.

The data type of a value is one of the XMS data types listed in Table 30 on page 97.

After an application creates a stream 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 StreamMessage class for C or C++. Each time the application writes a value to the message stream, the value and its data type are assembled immediately after the previous value written by the application. XMS maintains an internal cursor to remember the position of the last value 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 stream message, the body of the message is read-only. The application can use the appropriate read methods of the StreamMessage class for C or C++ to read the contents of the message stream. The application reads the values in sequence, and XMS maintains an internal cursor to remember the position of the last value that was read.

Using the C interface only, an application can skip over a value without reading it by calling a read function with a null pointer for the `value` parameter or by calling xmsStreamMsgReadBytes() or xmsStreamMsgReadObject() for the buffer parameter. For information about how to skip over a value that is a string, see "xmsStreamMsgReadString – Read String" on page 252.

When an application reads a value from the message stream, the value can be converted by XMS into another data type. For example, to read an integer from the message stream, an application can call the Read String method, which returns the integer as a string. The supported conversions are the same as those that are supported when XMS converts a property value from one data type to another. For more information about the supported conversions, see "Implicit conversion of a property value from one data type to another" on page 43.

If an error occurs while an application is attempting to read a value from the message stream, the cursor is not advanced. The application can recover from the error by attempting to read the value as another data type.

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

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

Text messages

The body of a text message contains a string.

After an application creates a text 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 text message, the body of the message is read-only. If an application calls the Clear Body method of the Message class for C or C++ when the body of a text 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 the 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.

When an application is connecting to IBM MQ V7.0 and above queue manager, the message selection is done at the queue manager side. XMS without doing any selection delivers the message it received from the queue manager, thus providing better performance.

Except for selecting messages by message identifier or correlation identifier, all message selection in previous releases of IBM MQ was done by Message Service Client for C/C++. In IBM MQ V7.0 and above, all message selection is done by the queue manager on all platforms except z/OS®. For an application connected to a z/OS queue manager, message selection is done by the queue manager in the publish/subscribe domain, but is still done by Message Service Client for C/C++ in the point-to-point domain.

As a result, message throughput is increased for applications that consume messages using message selection, where the message selection is done by the queue manager. The performance improvement is greater for an application that connects in the client mode because only those messages that satisfy the selection criteria are transported over the network, and Message Service Client for C/C++ sees only those messages that it delivers to the application.

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

A message selector expression can see the following properties of a message:

- JMS-defined properties
- IBM-defined properties

v Application-defined properties

It can also see the following message header fields:

v JMSCorrelationID
v JMSDeliveryMode
v JMSMessageID
v JMSPriority
v JMSTimestamp
v 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 the one in IBM MQ JMS. For information about how to construct a message selector expression, see *IBM MQ Using Java*. Note 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 IBM MQ JMS names. You cannot use the XMS names in a message selector expression.

Mapping XMS messages onto IBM MQ messages

The JMS header fields and properties of an XMS message are mapped onto fields in the header structures of a IBM MQ message.

When an XMS application is connected to a IBM MQ queue manager, messages sent to the queue manager are mapped onto IBM MQ messages in the same way that IBM MQ JMS messages are mapped onto IBM MQ messages in similar circumstances.

If the XMSC_WMQ_TARGET_CLIENT property of a Destination object is set to XMSC_WMQ_TARGET_DEST_JMS, the JMS header fields and properties of a message sent to the destination are mapped onto fields in the MQMD and MQRFH2 header structures of the IBM MQ message. Setting the XMSC_WMQ_TARGET_CLIENT property in this way assumes that the application that receives the message can handle an MQRFH2 header. The receiving application might therefore be another XMS application, a IBM MQ JMS application, or a native IBM MQ application that has been designed to handle an MQRFH2 header.

If the XMSC_WMQ_TARGET_CLIENT property of a Destination object is set to XMSC_WMQ_TARGET_DEST_MQ instead, the JMS header fields and properties of a message sent to the destination are mapped onto fields in the MQMD header structure of the IBM MQ message. The message does not contain an MQRFH2 header, and any JMS header fields and properties that cannot be mapped

onto fields in the MQMD header structure are ignored. The application that receives the message can therefore be a native IBM MQ that has not been designed to handle an MQRFH2 header.

IBM MQ messages received from a queue manager are mapped onto XMS messages in the same way that IBM MQ messages are mapped onto IBM MQ JMS messages in similar circumstances.

If an incoming IBM MQ message has an MQRFH2 header, the resulting XMS message has a body whose type is determined by the value of the Msd property contained in the mcd folder of the MQRFH2 header. If the Msd property is not present in the MQRFH2 header, or if the IBM MQ message has no MQRFH2 header, the resulting XMS message has a body whose type is determined by the value of the *Format* field in the MQMD header. If the *Format* field is set to MQFMT_STRING, the XMS message is a text message. Otherwise, the XMS message is a bytes message. If the IBM MQ message has no MQRFH2 header, only those JMS header fields and properties that can be derived from fields in the MQMD header are set.

For more information about mapping IBM MQ JMS messages onto IBM MQ messages, see *IBM MQ Using Java*.

Using the XMS sample applications

This chapter provides information about how to use the sample applications provided with XMS.

About this task

A number of sample applications are supplied with XMS. The samples provide an overview of the common features of each API. You can use these sample applications to verify your installation and messaging server setup, and also for guidance in building your own applications.

This chapter contains the following topics sections:

v "The sample applications"

v "Running the sample applications" on page 104

v "Building the C or C++ sample applications" on page 105

The sample applications

The sample applications provide an overview of the common features of each API. You can use them to verify your installation and messaging server setup and your own applications.

These samples do not cover the whole of the API, but rather provide an overview of how to use some of the most common features. They are subject to change in future releases of XMS.

If you require guidance on how to create your own applications, use the sample applications as a starting point. Look through the sample source code and identify the key steps to create each required object for your application (ConnectionFactory, Connection, Session, Destination, and a Producer, or a Consumer, or both), and to set any specific properties that are needed to specify how you want your application to work. For additional information, see Chapter 5, "Developing XMS applications," on page 25.

Table 31 shows the three sets of sample applications (one for each API) that are supplied with XMS.

Table 31. XMS sample applications

Name of sample	Description
SampleConsumerC SampleConsumerCPP SampleConsumerCS	A message consumer application that consumes messages from a queue or topic.
SampleProducerC SampleProducerCPP SampleConsumerCS	A message producer application that produces messages to a queue or on a topic.
SampleConfigC SampleConsumerCS	A configuration application that you can use to create a file-based administered object repository containing a connection factory and destination for your particular connection settings. This administered object repository can then be used with each of the sample consumer and producer applications.

The samples that support the same functionality in the various APIs have syntactical differences.

- The sample message consumer and producer applications both support the following:
  - Connections to IBM MQ, and a WebSphere service integration bus
  - Administered object repository lookups via the initial context interface
  - Connections to queues (IBM MQ and WebSphere service integration bus) and topics (IBM MQ, and WebSphere service integration bus)
  - Base, bytes, map, object, stream, and text messages.
- The sample message consumer application supports synchronous and asynchronous receive modes, and SQL Selector statements.
- The sample message producer application supports persistent and non-persistent delivery modes.

Both the source and a compiled version are provided for each application.

Operating modes

The samples can operate in one of two modes:

- Simple mode - you can run the samples with the minimum user input.
- Advanced mode - you can customize more finely the way in which the samples operate.

All the samples are compatible and can therefore operate across languages. For example, the SampleConsumerCPP application can run in conjunction with the Sample ProducerCS application.

Where to find the samples

To find out where sample applications for Message Service Client for C/C++ are installed:

- For AIX and Linux see Table 2 on page 16"What is installed on AIX, Linux, and Solaris" on page 16.
- For Windows, see Table 3 on page 17"What is installed on Windows (C/C++)" on page 17.

Running the sample applications

You can run the C and C++ sample applications interactively in either simple or advanced mode, or noninteractively using auto-generated or custom response files.

Before you begin

Before running any of the supplied sample applications, you must first set up the messaging server environment so that the applications can connect to a server as described in Chapter 4, "Setting up the messaging server environment," on page 21.

Setup IBM MQ Client environment

From XMS v3.0.0.0 onwards, IBM MQ client 9.2.2 and above is the pre-req for running any XMS application including the sample application. So you must setup MQ Client environment.
By default, applications use the primary installation. If there is no primary installation, or you do not want to use the primary installation, you must use the **setmqenv** command to specify which IBM MQ client installation to use.

On UNIX and Linux systems, using the **setmqenv** command set the **LD_LIBRARY_PATH**(**LIBPATH** on AIX), with the **-n** and **-k** option using the following command.

.<INSTALLATION PATH>/bin/setmqenv -n InstallationName -k, where **-n InstallationName**.

The **-k** parameter updates the **LD_LIBRARY_PATH**, or **LIBPATH** environment variable, with the path to the IBM MQ libraries at the start.

On UNIX platforms the leading "**.** "is critical. The dot, followed by a space, instructs the command shell to run **setmqenv** in the same command shell, and therefore inherit the environment set by **setmqenv**.

On Windows platforms, set the PATH environment variable using the **setmqenv -n** option.

INSTALLATION_PATH\bin\setmqenv -n InstallationName, where **-n InstallationName** sets up the environment for the installation named **InstallationName**.

If you are using Redistributable IBM MQ client on UNIX and Linux systems, using the **setmqenv** command set the **LD_LIBRARY_PATH**(**LIBPATH** on AIX), with the **-s** and **-k** option using the following command.

.<INSTALLATION PATH>/bin/setmqenv -s -k,

where the **-s** parameter sets up the environment for the installation that runs the setmqenv command.

The **-k** parameter updates the **LD_LIBRARY_PATH**, or **LIBPATH** environment variable, with the path to the IBM MQ libraries at the start.

On UNIX platforms the leading "**.** "is critical. The dot, followed by a space, instructs the command shell to run **setmqenv** in the same command shell, and therefore inherit the environment set by **setmqenv**.

For Redistributable client on Windows platforms, set the PATH environment variable using the **setmqenv -s**

INSTALLATION_PATH\bin\setmqenv -s,
where the **-s** parameter sets up the environment for the installation that runs the setmqenv command.

For more information about the other options for **setmqenv**, see the IBM MQ documentation.

Setup XMS environment

For C or C++ sample applications, you must have set up one of the following environment variables:

- On AIX, the <*install_dir*>/lib directory on your LIBPATH
- On Linux, the <*install_dir*>/lib directory on your LD_LIBRARY_PATH
- On Windows, the <*install_dir*>\bin directory on your PATH

About this task

The operation of the C and C++ sample applications is identical for all platforms.

Tip: When you are running a sample application, type ? at any time for help on what to do next.

The following steps summarize what you need to do to run the C and C++ sample applications.

Procedure

1. Select the mode in which you want to run the sample application. Type either `Advanced` or `Simple`.
2. Answer the questions. To select the default value, which is shown in the square brackets at the end of the question, press Enter. To select a different value, type the appropriate value, and press Enter.

   Here is an example question:

   ```
   Enter connection type [wpm]:
   ```

   In this case, the default value is wpm (connection to a WebSphere service integration bus).

Results

When you run the sample applications, response files are generated automatically in the current working directory. Response file names are in the format

<*connectiontype*>-<*sampletype*>.rsp; (for example, wpm-producer.rsp). If required, you can use a generated response file to rerun the sample application with the same options, without re-entering these options manually.

Building the C or C++ sample applications

When you build a sample C or C++ application, an executable version is created.

Before you begin

To build the C or C++ samples, you must have the appropriate compiler installed as described in "Operating environments" on page 8.

About this task

This section provides the information that you need to build the C and C++ applications.

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.
3. Type one of the following commands:
   a. If you are using AIX or Linux type `make`.
   b. If you are using Windows, type `nmake`.

   The command builds an executable version of the application in the current directory. This application has the same name as the folder; for example, if you are building the C version of the sample message producer application, SampleProducerC.exe is created in the SampleProducerC folder.
4. Before running the samples, make sure that the directory where you have installed XMS is specified by the appropriate environment variable:
   a. On AIX, the <install_dir>/lib directory must be in the path specified by the LIBPATH environment variable.
   b. On Linux, the <install_dir>/lib directory must be in the path specified by the LD_LIBRARY_PATH environment variable.
   c. On Windows, the <install_dir>\bin directory must be in the path specified by the PATH environment variable.

   Note: If the application is built in 64bit mode then on Windows <install_dir>/bin64 should be added to the PATH environment variable, in place of <install_dir>/bin and on all other platforms <install_dir>/lib64 should be added to the appropriate environment variable, in place of <install_dir>/lib.

Chapter 11. Troubleshooting

This chapter provides information to help you to detect and deal with problems.

About this task

This chapter provides information to help you with problem determination for XMS applications, and describes how to configure First Failure Data Capture (FFDC) and trace for C/C++ applications.

This chapter contains the following sections:

v "Problem determination for C/C++ applications"

v "FFDC and trace configuration for C/C++ applications" on page 109

v "Tips for troubleshooting" on page 111

Problem determination for C/C++ applications

This topic provides information to help you to detect and deal with problems in XMS C/C++ applications.

This topic contains the following subtopics:

v "Error conditions that can be handled at run time"

v "Error conditions that cannot be handled at run time" on page 108

v "Repeatable failures" on page 108

Error conditions that can be handled at run time

Return codes from API calls are error conditions that can be handled at run time. The way in which you deal with this type of error depends on whether you are using the C or C++ API.

How to detect errors at run time

If an application calls a C API function and the call fails, a response with a return code other than XMS_OK is returned with an XMS error block containing more information about the reason for the failure. For further details, see "Return codes" on page 61 and "ErrorBlock" on page 140.

The C++ API throws an exception when a method is used.

An application uses an exception listener to be notified asynchronously of a problem with a connection. The exception listener is supplied to, and is initialized using, the XMS C or C++ API. For further information, see "Message and exception listener functions in C" on page 62 and "Message and exception listeners in C++" on page 72.

How to handle errors at run time

Some error conditions are an indication that some resource is unavailable, and the action that an application can take depends on the XMS function that the application is calling. For example, if a connection fails to connect to the server, then the application may wish to retry periodically until a connection is made. An

XMS error block or exception might not contain enough information to determine what action to take, and, in these situations, there is often a linked error block or exception that contains more specific diagnostic information.

In the C API, always test for a response with a return code other than XMS_OK, and always pass an error block on the API call. The action taken usually depends on which API function is the application using. For further details, see "Error handling in C" on page 61.

In the C++ API, always include calls to methods in a try block and, to catch all types of XMS exception, specify the Exception class in the catch construct. For further details, see "Error handling in C++" on page 70.

The exception listener is an asynchronous error condition path that can be started at any time. When the exception listener function is started, on its own thread, it is usually an indication of a more severe failure than a normal XMS API error condition. Any appropriate action may be taken, but you must be careful to follow the rules for the XMS threading model as described in "The threading model" on page 26.

Error conditions that cannot be handled at run time

First Failure Data Capture (FFDC) records can be written in the current working directory if the XMS library code finds a condition that it cannot handle. If an FFDC record is written, this often indicates a serious condition, and it is likely that XMS functions incorrectly because of the same unhandleable condition.

The type of FFDC record that XMS generates depends on the type of failure that has occurred. There are two distinct types of FFDC record:

- The first type of FFDC record is sometimes, but not always, generated as a result of a user's own application causing a failure, and usually results in the application being terminated. This type of failure is often characterized in the FFDC record as 'Unhandled Exception detected' or 'SIGNAL xx received'. The FFDC record contains detailed information describing the cause of the failure and also contains a function stack back-trace which shows the failing function stack.
- The second type of FFDC record is generated by XMS itself in cases where it has detected an unexpected condition. Generally, the application continues to run but, depending upon the reason for which the FFDC record was generated, XMS API function calls may return negative responses.

Repeatable failures

If you are dealing with a repeatable failure, it might be necessary for you to capture product trace over an extended period of time to allow the problem to be diagnosed.

If you need to provide a product trace, either enable trace as advised by the IBM Support Center representative or as described in "FFDC and trace configuration for C/C++ applications" on page 109.

It is important that the size of the trace file is large enough to capture the trace while the repeatable problem occurs. To set the size of the trace file, either use environment variable XMS_TRACE_FILE_SIZE or use the gxisc executable command as follows:

```
alter trace(enabled) tracesize(xxxx)
```

Refer to Table 32 for the descriptions of various environment variable settings for C/C++ trace.

After the failure that you are tracing has occurred, you must either copy the trace files or disable the trace using the following command:

```
gxisc trace(disabled)
```

This is because trace wraps, which means that leaving trace on would eventually cause the trace at the point of failure to be lost.

XMS product trace is written in compressed form to gain a performance advantage. You can format the trace files using the gxitrcfmt tool.

If you are experiencing problems where you do not have access to the information provided in the error block, you may want to enable the XMS_FFDC_EXCEPTIONS environment variable. This produces an FFDC record whenever the XMS API returns an error from a function. The FFDC record contains full details of the XMS error block to assist in debugging failures.

Refer to "FFDC and trace configuration for C/C++ applications" for more information on the gxisc and gxitrcfmt commands.

FFDC and trace configuration for C/C++ applications

First Failure Data Capture (FFDC) records are stored in human readable text files with names that start with the prefix xmsffdc. Trace files are binary and can be formatted. Trace file names start with the prefix xms.

XMS creates FFDC records and trace files in the current working directory, unless you specify an alternative location by configuring an XMS environment variable as described below.

Trace configuration using XMS environment variables

To configure trace for an XMS C or C++ application, set the following XMS environment variables before running the application:

Table 32. Environment variable settings for C/C++ trace

Environment variables	Default	Settings	Meaning
XMS_TRACE_ON	Not applicable	normal	Selected components are traced.
		full	All components are traced.

Table 32. Environment variable settings for C/C++ trace (continued)

Environment variables	Default	Settings	Meaning
		partial	A comma separated list of component identifiers to trace. For example, "partial,osa,cal" only traces XMS components gxiosa and gxical. Use full trace to show the components that can be traced.
XMS_TRACE_FILE_PATH	Current working directory	/dirpath/	The directory path that trace and FFDC records are written to. XMS creates FFDC and trace files 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 FFDC and trace files. You must set this 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 FFDC and trace files.
XMS_TRACE_FILE_SIZE	200000	integer	The maximum size that XMS product trace can grow to (in kilobytes), that is, 10 represents 10,000 bytes.
XMS_TRACE_FILE_NUMBER	4	integer	The number of files that can be used to store trace records. (200000 / 4 = 50000 bytes per file.)

Dynamic trace configuration

To configure trace dynamically, use the executable gxisc. You can use gxisc to enable and disable trace in a running XMS C or C++ application, and to modify the trace size. You must run gxisc on the same machine as the XMS application.

To invoke gxisc, use the process id of the XMS application for which you want to alter the trace configuration, as shown in the example below.

```
gxisc 1234                              <enter>
display all                             <enter>
alter trace(enabled) tracesize(100)     <enter>
help                                    <enter>
alter trace(disabled)                   <enter>
alter                                   <enter>
end

gxisc                                   <enter>
alter pid(1234) trace(enabled)          <enter>
end

cat a.file                              <enter>

        alter pid(1234) trace(enabled)
        end

cat a.file | gxisc                      <enter>


cat b.file                              <enter>

        alter trace(disabled) tracesize(1000)
        end

cat b.file | gxisc 1234                 <enter>
```

Note: Trace settings are not retained after the XMS C or C++ application terminates.

Trace file formatting

To minimize processing and disk overheads at runtime, XMS outputs trace in a binary format into one or more trace files with the extension .trc. You can format trace files by using the executable gxitrcfmt, as shown in the following example:

```
gxitrcfmt xms01234.trc
```

A formatted file has the suffix txt, for example:

```
cat xms01234.trc.txt
```

Tips for troubleshooting

Use these tips to help you troubleshoot problems with using XMS.

An XMS application cannot connect to a queue manager (not authorized)

The XMS C/C++ clients may have different behavior from that of the IBM MQ JMS client. Therefore, you may find that your XMS application cannot connect to your queue manager, although your JMS application can.

The XMS C/C++ client may have different behavior from that of theWebSphere MQ JMS client. Therefore, you may find that your XMS application cannot connect to your queue manager, although your JMS application can.

- A simple solution to this problem is to try using a userid that is no more than 12 characters long and is authorized completely in the queue manager's authority list. If this solution is not ideal, a different but more complex approach would be to use security exits. If you need further help on this issue, contact IBM Support for assistance.
- If you set the XMSC_USERID property of the connection factory, it must match the userid and password of the logged on user. If you do not set this property, the queue manager will use the userid of the logged on user by default.

Chapter 12. C classes

This topic documents the C classes and their functions.

The following table summarizes all the classes.

Table 33. Summary of the C classes

Class	Description
"BytesMessage" on page 114	A bytes message is a message whose body comprises a stream of bytes.
"Connection" on page 127	A Connection object represents an application's active connection.
"ConnectionFactory for the C class" on page 132	An application uses a connection factory to create a connection.
"ConnectionMetaData" on page 135	A ConnectionMetaData object provides information about a connection.
"Destination for the C class" on page 136	A destination is where an application sends messages, or it is a source from which an application receives messages, or both.
"ErrorBlock" on page 140	If a C function call fails, XMS can store information about why the call failed in an error block.
"ExceptionListener" on page 144	An application uses an exception listener to be notified asynchronously of a problem with a connection.
"InitialContext" on page 145	An application uses an InitialContext object to create objects from object definitions that are retrieved from a repository of administered objects.
"Iterator" on page 147	An iterator encapsulates a list of objects. An application uses an iterator to access each object in turn.
"MapMessage" on page 149	A map message is a message whose body comprises a set of name-value pairs, where each value has an associated data type.
"Message" on page 164	A Message object represents a message that an application sends or receives.
"MessageConsumer" on page 180	An application uses a message consumer to receive messages sent to a destination.
"MessageListener" on page 184	An application uses a message listener to receive messages asynchronously.
"MessageProducer" on page 185	An application uses a message producer to send messages to a destination.
"ObjectMessage" on page 194	An object message is a message whose body comprises a serialized Java object.
"Property" on page 196	A Property object represents a property of an object.

Table 33. Summary of the C classes (continued)

Class	Description
"PropertyContext" on page 211	The PropertyContext class contains functions that get and set properties. These functions can operate on any object that can have properties.
"QueueBrowser" on page 228	An application uses a queue browser to browse messages on a queue without removing them.
"Requestor" on page 230	An application uses a requestor to send a request message and then wait for, and receive, the reply.
"Session" on page 232	A session is a single threaded context for sending and receiving messages.
"StreamMessage" on page 245	A stream message is a message whose body comprises a stream of values, where each value has an associated data type.
"TextMessage" on page 259	A text message is a message whose body comprises a string.

The definition of each function lists the exception codes that XMS might return if it detects an error while processing a call to the function. Each exception code is represented by its named constant.

BytesMessage

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

Functions

Summary of functions:

Function	Description
xmsBytesMsgGetBodyLength	Get the length of the body of the message when the body of the message is read-only.
xmsBytesMsgReadBoolean	Read a boolean value from the bytes message stream.
xmsBytesMsgReadByte	Read the next byte from the bytes message stream as a signed 8-bit integer.
xmsBytesMsgReadBytes	Read an array of bytes from the bytes message stream starting from the current position of the cursor.
xmsBytesMsgReadBytesByRef	Get a pointer to the start of the bytes message stream and get the length of the stream.
xmsBytesMsgReadChar	Read the next 2 bytes from the bytes message stream as a character.
xmsBytesMsgReadDouble	Read the next 8 bytes from the bytes message stream as a double precision floating point number.
xmsBytesMsgReadFloat	Read the next 4 bytes from the bytes message stream as a floating point number.
xmsBytesMsgReadInt	Read the next 4 bytes from the bytes message stream as a signed 32-bit integer.
xmsBytesMsgReadLong	Read the next 8 bytes from the bytes message stream as a signed 64-bit integer.
xmsBytesMsgReadShort	Read the next 2 bytes from the bytes message stream as a signed 16-bit integer.

Function	Description
xmsBytesMsgReadUnsignedByte	Read the next byte from the bytes message stream as an unsigned 8-bit integer. xmsBytesMsgReadUnsignedShort Read the next 2 bytes from the bytes message stream as an unsigned 16-bit integer.
xmsBytesMsgReadUTF	Read a string, encoded in UTF-8, from the bytes message stream.
xmsBytesMsgReset	Put the body of the message into read-only mode and reposition the cursor at the beginning of the bytes message stream.
xmsBytesMsgWriteBoolean	Write a boolean value to the bytes message stream.
xmsBytesMsgWriteByte	Write a byte to the bytes message stream.
xmsBytesMsgWriteBytes	Write an array of bytes to the bytes message stream.
xmsBytesMsgWriteChar	Write a character to the bytes message stream as 2 bytes, high order byte first.
xmsBytesMsgWriteDouble	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.
xmsBytesMsgWriteFloat	Convert a floating point number to an integer and write the integer to the bytes message stream as 4 bytes, high order byte first.
xmsBytesMsgWriteInt	Write an integer to the bytes message stream as 4 bytes, high order byte first.
xmsBytesMsgWriteLong	Write a long integer to the bytes message stream as 8 bytes, high order byte first.
xmsBytesMsgWriteShort	Write a short integer to the bytes message stream as 2 bytes, high order byte first.
xmsBytesMsgWriteUTF	Write a string, encoded in UTF-8, to the bytes message stream.

xmsBytesMsgGetBodyLength – Get Body Length

Interface:

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

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 function 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

xmsBytesMsgReadBoolean – Read Boolean Value

Interface:

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

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 function skips over the boolean value without reading it.

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

xmsBytesMsgReadByte – Read Byte

Interface:

```
xmsRC xmsBytesMsgReadByte(xmsHMsg message,
                          xmsSBYTE *value,
                          xmsHErrorBlock errorBlock);
```

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 function skips over the byte without reading it.

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

xmsBytesMsgReadBytes – Read Bytes

Interface:

```
xmsRC xmsBytesMsgReadBytes(xmsHMsg message,
                           xmsSBYTE *buffer,
                           xmsINT bufferLength,
                           xmsINT *returnedLength,
                           xmsHErrorBlock errorBlock);
```

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 function 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. If you specify `XMSC_QUERY_SIZE` instead, no bytes are read into the buffer, but the number of bytes remaining in the stream, starting from the current position of the cursor, is returned in the returnedLength parameter, and the cursor is not advanced.

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`.

If you specify a null pointer on input, the function returns no value.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

xmsBytesMsgReadBytesByRef – Read Bytes by Reference

Interface:

```
xmsRC xmsBytesMsgReadBytesByRef(xmsHMsg message,
                                xmsSBYTE **stream,
                                xmsINT *length,
                                xmsHErrorBlock errorBlock);
```

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 60.

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

xmsBytesMsgReadChar – Read Character

Interface:

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

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 function skips over the bytes without reading them.

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

xmsBytesMsgReadDouble – Read Double Precision Floating Point Number

Interface:

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

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 function skips over the bytes without reading them.

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

xmsBytesMsgReadFloat – Read Floating Point Number

Interface:

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

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 function skips over the bytes without reading them.

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

xmsBytesMsgReadInt – Read Integer

Interface:

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

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 function skips over the bytes without reading them.

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

xmsBytesMsgReadLong – Read Long Integer

Interface:

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

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 function skips over the bytes without reading them.

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

xmsBytesMsgReadShort – Read Short Integer

Interface:

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

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 function skips over the bytes without reading them.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

v XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
v XMS_X_MESSAGE_EOF_EXCEPTION

xmsBytesMsgReadUnsignedByte – Read Unsigned Byte

Interface:

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

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 function skips over the byte without reading it.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
v XMS_X_MESSAGE_EOF_EXCEPTION

xmsBytesMsgReadUnsignedShort – Read Unsigned Short Integer

Interface:

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

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 function skips over the bytes without reading them.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
v XMS_X_MESSAGE_EOF_EXCEPTION

xmsBytesMsgReadUTF – Read UTF String

Interface:

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

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 function, see "C functions that return a string by value" on page 58.

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 XMSC_QUERY_SIZE, the string is not returned, but its length is returned in the actualLength parameter, and the cursor is not advanced.

If you specify XMSC_SKIP, the function skips over the string without reading it.

actualLength (output)
The length of the string in bytes. If data conversion is required, this is the length of the string after conversion. If you specify a null pointer on input, the length is not returned.

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

Notes:

1. 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 an error. XMS does not advance the internal cursor.
2. 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.

xmsBytesMsgReset – Reset

Interface:

```
xmsRC xmsBytesMsgReset(xmsHMsg message,
                       xmsHErrorBlock errorBlock);
```

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

xmsBytesMsgWriteBoolean – Write Boolean Value

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteByte – Write Byte

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteBytes – Write Bytes

Interface:

```
xmsRC xmsBytesMsgWriteBytes(xmsHMsg message,
                            xmsSBYTE *value,
                            xmsINT length,
                            xmsHErrorBlock errorBlock);
```

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteChar – Write Character

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteDouble – Write Double Precision Floating Point Number

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteFloat – Write Floating Point Number

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteInt – Write Integer

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteLong – Write Long Integer

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteShort – Write Short Integer

Interface:

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

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsBytesMsgWriteUTF – Write UTF String

Interface:

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

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)
A character array containing the string to be written.

length (input)
The length of the string in bytes. If the string is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Connection

A Connection object represents an application's active connection.

For a list of the XMS defined properties of a Connection object, see "Properties of Connection" on page 401.

Functions

Summary of functions:

Function	Description
xmsConnClose	Close the connection.
xmsConnCreateSession	Create a session.
xmsConnGetClientID	Get the client identifier for the connection.
xmsConnGetExceptionListener	Get pointers to the exception listener function and context data that are registered with the connection.
xmsConnGetMetaData	Get the metadata for the connection.
xmsConnSetClientID	Set a client identifier for the connection.
xmsConnSetExceptionListener	Register an exception listener function and context data with the connection.

Function	Description
xmsConnStart	Start, or restart the delivery of incoming messages for the connection.
xmsConnStop	Stop the delivery of incoming messages for the connection.

xmsConnClose – Close Connection

Interface:

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

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsConnCreateSession – Create Session

Interface:

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

Create a session.

Parameters:

connection (input)
The handle for the connection.

transacted (input)
The value `xmsTRUE` means that the session is transacted. The value `xmsFALSE` means that the session is not transacted.

acknowledgeMode (input)
Indicates how messages received by an application are acknowledged. The value must be one of the following acknowledgement modes:

`XMSC_AUTO_ACKNOWLEDGE`

`XMSC_CLIENT_ACKNOWLEDGE`

`XMSC_DUPS_OK_ACKNOWLEDGE`

This parameter is ignored if the session is transacted. For more information about acknowledgement modes, see "Message acknowledgement" on page 29.

session (output)
The handle for the session.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsConnGetClientID – Get Client ID

Interface:

```
xmsRC xmsConnGetClientID(xmsHConn connection,
                         xmsCHAR *clientID,
                         xmsINT length,
                         xmsINT *actualLength,
                         xmsHErrorBlock errorBlock);
```

Get the client identifier for the connection.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

connection (input)
The handle for the connection.

clientID (output)
The buffer to contain the client identifier.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the client identifier is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the client identifier in bytes. If data conversion is required, this is the length of the client identifier after conversion. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsConnGetExceptionListener – Get Exception Listener

Interface:

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

Get pointers to the exception listener function and context data that are registered with the connection.

For more information about using exception listener functions, see "Exception listener functions in C" on page 63.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsConnGetMetaData – Get Metadata

Interface:

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

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsConnSetClientID – Set Client ID

Interface:

```
xmsRC xmsConnSetClientID(xmsHConn connection,
                         xmsCHAR *clientID,
                         xmsINT length,
                         xmsHErrorBlock errorBlock)
```

Set a client identifier for the connection. A client identifier is used only to support durable subscriptions in the publish/subscribe domain, and is ignored in the point-to-point domain.

If an application calls this function to set a client identifier for a connection, the application must do so immediately after creating the connection, and before performing any other operation on the connection. If the application tries to call the function after this point, the function returns exception XMS_X_ILLEGAL_STATE_EXCEPTION.

Parameters:

connection (input)
The handle for the connection.

clientID (input)
The client identifier as a character array.

length (input)
The length of the client identifier in bytes. If the client identifier is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION
- XMS_X_INVALID_CLIENTID_EXCEPTION

xmsConnSetExceptionListener – Set Exception Listener

Interface:

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

Register an exception listener function and context data with the connection.

For more information about using exception listener functions, see "Exception listener functions in C" on page 63.

Parameters:

connection (input)
The handle for the connection.

lsr (input)
A pointer to the exception listener function. If an exception listener function 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsConnStart – Start Connection

Interface:

```
xmsRC xmsConnStart(xmsHConn connection,
                   xmsHErrorBlock errorBlock);
```

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsConnStop – Stop Connection

Interface:

```
xmsRC xmsConnStop(xmsHConn connection,
                  xmsHErrorBlock errorBlock);
```

Stop 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

ConnectionFactory for the C class

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 402.

Functions

Summary of functions:

Function	Description
xmsConnFactCreate	Create a connection factory with the default properties.
xmsConnFactCreateConnection	Create a connection using the default user identity.
xmsConnFactCreateConnectionForUser	Create a connection using a specified user identity.
xmsConnFactDispose	Delete the connection factory.

xmsConnFactCreate – Create Connection Factory

Interface:

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

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:

v XMS_X_GENERAL_EXCEPTION

xmsConnFactCreateConnection – Create Connection (using the default user identity)

Interface:

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

Create a connection using the default user identity.

If you are connecting to IBM MQ, and you set the XMSC_USERID property of the connection factory, it must match the userid of the logged on user. If you do not set these properties, the queue manager will use the userid of the logged on user by default. If you require further connection-level authentication of individual users you can write a client authentication exit which is configured in IBM MQ. You can learn more about creating a client authentication exit in the Authentication topic in the IBM MQ documentation manual.

The connection is created in stopped mode. No messages are delivered until the application calls xmsConnStart().

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_SECURITY_EXCEPTION

xmsConnFactCreateConnectionForUser – Create Connection (using a specified user identity)

Interface:

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

Create a connection using a specified user identity.

If you are connecting to IBM MQ, and you set the XMSC_USERID property of the connection factory, it must match the userid of the logged on user. If you do not set these properties, the queue manager will use the userid of the logged on user by default. If you require further connection-level authentication of individual users you can write a client authentication exit which is configured in IBM MQ. You can learn more about creating a client authentication exit in the Authentication topic in the IBM MQ documentation manual.

The connection is created in stopped mode. No messages are delivered until the application calls xmsConnStart().

Parameters:

factory (input)
The handle for the connection factory.

userID (input)
The user identifier to be used to authenticate the application. The user identifier is in the format of a null terminated string. If the user identifier is null, the connection factory property XMSC_USERID is used instead.

password (input)
The password to be used to authenticate the application. The password is in the format of a null terminated string. If the password is null, the connection factory property XMSC_PASSWORD is used instead.

connection (output)
The handle for the connection.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_SECURITY_EXCEPTION

xmsConnFactDispose – Delete Connection Factory

Interface:

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

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

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 406.

Functions

Summary of functions:

Function	Description
xmsConnMetaDataGetJMSXProperties	Get a list of the names of the JMS defined message properties supported by the connection.

xmsConnMetaDataGetJMSXProperties – Get JMS Defined Message Properties

Interface:

```
xmsRC
xmsConnMetaDataGetJMSXProperties(xmsHConnMetaData connectionMetaData,
                                 xmsHIterator *iterator,
                                 xmsHErrorBlock errorBlock);
```

Get a list of the names of the JMS defined message properties supported by the connection.

The function returns an iterator that encapsulates a list of Property objects, where each Property object encapsulates the name of a JMS defined message property. The application can then use the iterator to retrieve the name of each JMS defined message property in turn.

Note: The equivalent JMS method performs a slightly different function. The JMS method returns an enumeration of the names of the JMS defined message properties.

Parameters:

connectionMetaData (input)
The handle for the connection metadata.

iterator (output)
The handle for the iterator.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Destination for the C class

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 407.

Functions

Summary of functions:

Function	Description
xmsDestCreate	Create a destination using the specified uniform resource identifier (URI).
xmsDestCreateByType	Create a destination using the specified destination type and name.
xmsDestCreateTemporaryByType	Create a temporary destination.
xmsDestDispose	Delete the destination.
xmsDestGetName	Get the name of the destination.
xmsDestGetTypeId	Get the type of the destination.
xmsDestToString	Get the name of the destination in the format of a uniform resource identifier (URI).

xmsDestCreate – Create Destination (using a URI)

Interface:

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

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

For a destination that is a queue, this function does not create the queue in the messaging server. You must create the queue before an application can call this function.

Parameters:

URI (input)
The URI in the format of a null terminated string.

destination (output)
The handle for the destination.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestCreateByType – Create Destination (specifying a type and name)

Interface:

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

Create a destination using the specified destination type and name.

For a destination that is a queue, this function does not create the queue in the messaging server. You must create the queue before an application can call this function.

Parameters:

destinationType (input)
The type of the destination, which must be one of the following values:

XMS_DESTINATION_TYPE_QUEUE

XMS_DESTINATION_TYPE_TOPIC

destinationName (input)
The name of the destination, which can be the name of a queue or the name of a topic. The name is in the format of a null terminated string.

If the destination is a IBM MQ queue, you can specify the name of the destination in either of the following ways:

QName

QMgrName/QName

where *QName* is the name of a IBM MQ queue, and *QMgrName* is the name of a IBM MQ queue manager. The IBM MQ queue name resolution process uses the values of *QName* and *QMgrName* to determine the actual destination queue. For more information about the queue name resolution process, see the *IBM MQ documentation*.

destination (output)
The handle for the destination.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestCreateTemporaryByType – Create Temporary Destination

Interface:

```
xmsRC xmsDestCreateTemporaryByType(xmsDESTINATION_TYPE destinationType,
                                   xmsHSess session,
                                   xmsHDest *destination,
                                   xmsHErrorBlock errorBlock);
```

Create a temporary destination.

The scope of the temporary destination is the connection. Only the sessions created by the connection can use the temporary destination.

The temporary destination remains until it is explicitly deleted, or the connection ends, whichever is the sooner.

For more information about temporary destinations, see "Temporary destinations" on page 36.

Parameters:

destinationType (input)
The type of the temporary destination, which must be one of the following values:

XMS_DESTINATION_TYPE_QUEUE

XMS_DESTINATION_TYPE_TOPIC

session (input)
The handle for the session.

destination (output)
The handle for the temporary destination.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestDispose – Delete Destination

Interface:

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

Delete the destination.

For a destination that is a queue, this function does not delete the queue in the messaging server unless the queue was created for an XMS temporary queue.

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestGetName – Get Destination Name

Interface:

```
xmsRC xmsDestGetName(xmsHDest destination,
                     xmsCHAR *destinationName,
                     xmsINT length,
                     xmsINT *actualLength,
                     xmsHErrorBlock errorBlock);
```

Get the name of the destination.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

destination (input)
The handle for the destination.

destinationName (output)
The buffer to contain the name of the destination. The name is either the name of a queue or the name of a topic.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestGetTypeId – Get Destination Type

Interface:

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

Get the type of the destination.

Parameters:

destination (input)
The handle for the destination.

destinationType (output)
The type of the destination, which is one of the following values:

XMS_DESTINATION_TYPE_QUEUE

XMS_DESTINATION_TYPE_TOPIC

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsDestToString – Get Destination Name as URI

Interface:

```
xmsRC xmsDestToString(xmsHDest destination,
                      xmsCHAR *destinationName,
                      xmsINT length,
                      xmsINT *actualLength,
                      xmsHErrorBlock errorBlock);
```

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

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

destination (input)
The handle for the destination.

destinationName (output)
The buffer to contain the URI. The URI is either a queue URI or a topic URI.

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

actualLength (output)
The length of the URI in bytes. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

ErrorBlock

If a C function call fails, XMS can store information about why the call failed in an error block.

For more information about the error block and its contents, see "The error block" on page 61.

Functions in this class return the following return codes:

Return code	Meaning
XMS_OK	The call completed successfully.
Any other value	The call failed, for example, because the error block that was passed to the function was not valid.

This class is a helper class.

Functions

Summary of functions:

Function	Description
xmsErrorClear	Clear the contents of the error block.
xmsErrorCreate	Create an error block.
xmsErrorDispose	Delete the error block.
xmsErrorGetErrorCode	Get the error code.
xmsErrorGetErrorData	Get the error data.
xmsErrorGetErrorString	Get the error string.
xmsErrorGetJMSException	Get the exception code.
xmsErrorGetLinkedError	Get the handle for the next error block in the chain of error blocks.

xmsErrorClear – Clear Error Block

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 an API function call.

Parameters:

errorBlock (input)
The handle for the error block.

Thread context:
Any

xmsErrorCreate – Create Error Block

Interface:

```
xmsRC xmsErrorCreate(xmsHErrorBlock *errorBlock);
```

Create an error block.

In a newly created error block, the exception code is XMS_X_NO_EXCEPTION.

Parameters:

errorBlock (output)
The handle for the error block.

Thread context:
Any

xmsErrorDispose – Delete Error Block

Interface:

```
xmsRC xmsErrorDispose(xmsHErrorBlock *errorBlock);
```

Delete the error block.

Only the first error block in a chain of error blocks can be explicitly deleted. By deleting the first error block in a chain, all subsequent error blocks in the chain are also deleted.

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 function returns a null handle.

Thread context:
Any

xmsErrorGetErrorCode – Get Error Code

Interface:

```
xmsRC xmsErrorGetErrorCode(xmsHErrorBlock errorBlock,
                           xmsINT *errorCode);
```

Get the error code.

For more information about the error code, see "The error block" on page 61.

Parameters:

errorBlock (input)
The handle for the error block.

errorCode (output)
The error code.

Thread context:
Any

xmsErrorGetErrorData – Get Error Data

Interface:

```
xmsRC xmsErrorGetErrorData(xmsHErrorBlock errorBlock,
                           xmsCHAR *buffer,
                           xmsINT bufferLength,
                           xmsINT *actualLength);
```

Get the error data.

For more information about the error data, see "The error block" on page 61.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

Thread context:
Any

xmsErrorGetErrorString – Get Error String

Interface:

```
xmsRC xmsErrorGetErrorString(xmsHErrorBlock errorBlock,
                             xmsCHAR *buffer,
                             xmsINT bufferLength,
                             xmsINT *actualLength);
```

Get the error string.

For more information about the error string, see "The error block" on page 61.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

Thread context:
Any

xmsErrorGetJMSException – Get Exception Code

Interface:

```
xmsRC xmsErrorGetJMSException(xmsHErrorBlock errorBlock,
                              xmsJMSEXP_TYPE *exceptionCode);
```

Get the exception code.

For more information about the exception code, see "The error block" on page 61.

Parameters:

errorBlock (input)
The handle for the error block.

exceptionCode (output)
The exception code. If the error block is in a chain of error blocks, but is not the first in the chain, the exception code is always XMS_X_GENERAL_EXCEPTION.

Thread context:
Any

xmsErrorGetLinkedError – Get Linked Error

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 function returns a null handle if there are no more error blocks in the chain.

Thread context:
Any

ExceptionListener

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. In other situations, an

exception listener can provide a more immediate way of learning about a problem with a connection than waiting until the next synchronous call to XMS.

Functions

Summary of functions:

Function	Description
onException	Notify the application of a problem with a connection.

onException – On Exception

Interface:

```
xmsVOID onException(xmsCONTEXT context,
                    xmsHErrorBlock errorBlock);
```

Notify the application of a problem with a connection.

onException() is the exception listener function that is registered with the connection. The name of the function does not have to be onException.

For more information about using exception listener functions, see "Exception listener functions in C" on page 63.

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.

InitialContext

An application uses an InitialContext object to create objects from object definitions that are retrieved from a repository of administered objects.

For a list of the XMS defined properties of an InitialContext object, see "Properties of InitialContext" on page 408.

Functions

Summary of functions:

Function	Description
xmsInitialContextCreate	Create an InitialContext object.
xmsInitialContextDispose	Delete the InitialContext object.
xmsInitialContextLookup	Create an object from an object definition that is retrieved from the repository of administered objects.

xmsInitialContextCreate – Create Initial Context

Interface:

```
xmsRC xmsInitialContextCreate(xmsCHAR *URL,
                              xmsHInitialContect *initalContext,
                              xmsHErrorBlock errorBlock);
```

Create an InitialContext object.

Parameters:

URL (input)
A uniform resource locator (URL) that identifies the name and location of a repository containing administered objects. The URL is in the format of a null terminated string.

initialContext (output)
The handle for the InitialContext object.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsInitialContextDispose – Delete Initial Context

Interface:

```
xmsRC xmsInitialContextDispose(xmsHInitialContext *initialContext,
                               xmsHErrorBlock errorBlock);
```

Delete the InitialContext object.

If an application tries to delete an InitialContext object that is already deleted, the call is ignored.

Parameters:

initialContext (input/output)
On input, the handle for the InitialContext object. On output, the function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsInitialContextLookup – Look Up Object in Initial Context

Interface:

```
xmsRC xmsInitialContextLookup(xmsHInitialContext initialContext,
                              xmsCHAR *objectName,
                              xmsHObj *returnedObject,
                              xmsHANDLE_TYPE *handleType,
                              xmsHErrorBlock errorBlock);
```

Create an object from an object definition that is retrieved from the repository of administered objects.

Parameters:

initialContext (input)
The handle for the InitialContext object.

objectName (input)
The name of the administered object in the format of a null terminated string.

returnedObject (output)
The handle for the object that is created.

handleType (output)
The type of the handle for the object that is created, which is one of following values:

```
XMS_HANDLE_TYPE_CONNFACT
XMS_HANDLE_TYPE_DEST
```

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Iterator

An iterator encapsulates a list of objects. An application uses an iterator to access each object in turn.

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

An application cannot create an iterator directly. An iterator is created only by certain functions in order to pass a list of objects back to the application.

This class is a helper class.

Functions

Summary of functions:

Function	Description
xmsIteratorDispose	Delete the iterator.
xmsIteratorGetNext	Move the cursor to the next object and get the object at the new position of the cursor.
xmsIteratorHasNext	Check whether there are any more objects beyond the current position of the cursor.
xmsIteratorReset	Move the cursor back to a position before the first object.

xmsIteratorDispose – Delete Iterator

Interface:

```
xmsRC xmsIteratorDispose(xmsHIterator *iterator,
                         xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsIteratorGetNext – Get Next Object

Interface:

```
xmsRC xmsIteratorGetNext(xmsHIterator iterator,
                         xmsHObj *object,
                         xmsHErrorBlock errorBlock);
```

Move the cursor to the next object and get the object at the new position of the cursor.

Parameters:

iterator (input)
The handle for the iterator.

object (output)
The handle for the object.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsIteratorHasNext – Check for More Objects

Interface:

```
xmsRC xmsIteratorHasNext(xmsHIterator iterator,
                         xmsBOOL *moreProperties,
                         xmsHErrorBlock errorBlock);
```

Check whether there are any more 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)
The value is `xmsTRUE` if there are more objects beyond the current position of the cursor. The value is `xmsFALSE` if there are no more objects beyond the current position of the cursor.

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

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsIteratorReset – Reset Iterator

Interface:

```
xmsRC xmsIteratorReset(xmsHIterator iterator,
                       xmsHErrorBlock errorBlock);
```

Move the cursor back to a position before the first object.

Parameters:

iterator (input)
The handle for the iterator.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

MapMessage

A map message is a message whose body comprises a set of name-value pairs, where each value has an associated data type.

When an application gets the value of name-value pair, the value can be converted by XMS into another data type. For more information about this form of implicit conversion, see “Map messages” on page 98.

Functions

Summary of functions:

Function	Description
xmsMapMsgGetBoolean	Get the boolean value identified by name from the body of the map message.
xmsMapMsgGetByte	Get the byte identified by name from the body of the map message.
xmsMapMsgGetBytes	Get the array of bytes identified by name from the body of the map message.
xmsMapMsgGetBytesByRef	Get a pointer to an array of bytes in the body of the map message and get the length of the array.
xmsMapMsgGetChar	Get the character identified by name from the body of the map message.
xmsMapMsgGetDouble	Get the double precision floating point number identified by name from the body of the map message.
xmsMapMsgGetFloat	Get the floating point number identified by name from the body of the map message.
xmsMapMsgGetInt	Get the integer identified by name from the body of the map message.
xmsMapMsgGetLong	Get the long integer identified by name from the body of the map message.
xmsMapMsgGetMap	Get a list of the name-value pairs in the body of the map message.

Function	Description
xmsMapMsgGetObject	Get the value of a name-value pair, and its data type, from the body of the map message.
xmsMapMsgGetShort	Get the short integer identified by name from the body of the map message.
xmsMapMsgGetString	Get the string identified by name from the body of the map message.
xmsMapMsgGetStringByRef	Get a pointer to the string identified by name and get the length of the string.
xmsMapMsgItemExists	Check whether the body of the map message contains a name-value pair with the specified name.
xmsMapMsgSetBoolean	Set a boolean value in the body of the map message.
xmsMapMsgSetByte	Set a byte in the body of the map message.
xmsMapMsgSetBytes	Set an array of bytes in the body of the map message.
xmsMapMsgSetChar	Set a 2-byte character in the body of the map message.
xmsMapMsgSetDouble	Set a double precision floating point number in the body of the map message.
xmsMapMsgSetFloat	Set a floating point number in the body of the map message.
xmsMapMsgSetInt	Set an integer in the body of the map message.
xmsMapMsgSetLong	Set a long integer in the body of the map message.
xmsMapMsgSetObject	Set a value, with a specified data type, in the body of the map message.
xmsMapMsgSetShort	Set a short integer in the body of the map message.
xmsMapMsgSetString	Set a string in the body of the map message.

xmsMapMsgGetBoolean – Get Boolean Value

Interface:

```
xmsRC xmsMapMsgGetBoolean(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL *value,
                          xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetByte – Get Byte

Interface:

```
xmsRC xmsMapMsgGetByte(xmsHMsg message,
                       xmsCHAR *name,
                       xmsSBYTE *value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetBytes – Get Bytes

Interface:

```
xmsRC xmsMapMsgGetBytes(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSBYTE *buffer,
                        xmsINT bufferLength,
                        xmsINT *actualLength,
                        xmsHErrorBlock errorBlock);
```

Get the array of bytes identified by name from the body of the map message.

For more information about how to use this function, see “C functions that return a byte array by value” on page 59.

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.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length of the array is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetBytesByRef – Get Bytes by Reference

Interface:

```
xmsRC xmsMapMsgGetBytesByRef(xmsHMsg message,
                             xmsCHAR *name,
                             xmsSBYTE **array,
                             xmsINT *length,
                             xmsHErrorBlock errorBlock);
```

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 60.

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:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetChar – Get Character

Interface:

```
xmsRC xmsMapMsgGetChar(xmsHMsg message,
                       xmsCHAR *name,
                       xmsCHAR16 *value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (output)
The character retrieved from the body of the map message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetDouble – Get Double Precision Floating Point Number

Interface:

```
xmsRC xmsMapMsgGetDouble(xmsHMsg message,
                         xmsCHAR *name,
                         xmsDOUBLE *value,
                         xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetFloat – Get Floating Point Number

Interface:

```
xmsRC xmsMapMsgGetFloat(xmsHMsg message,
                        xmsCHAR *name,
                        xmsFLOAT *value,
                        xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetInt – Get Integer

Interface:

```
xmsRC xmsMapMsgGetInt(xmsHMsg message,
                      xmsCHAR *name,
                      xmsINT *value,
                      xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (output)
The integer retrieved from the body of the map message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetLong – Get Long Integer

Interface:

```
xmsRC xmsMapMsgGetLong(xmsHMsg message,
                       xmsCHAR *name,
                       xmsLONG *value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetMap – Get Name-Value Pairs

Interface:

```
xmsRC xmsMapMsgGetMap(xmsHMsg message,
                      xmsHIterator *iterator,
                      xmsHErrorBlock errorBlock);
```

Get a list of the name-value pairs in the body of the map message.

The function returns an iterator that encapsulates a list of Property objects, where each Property object encapsulates a name-value pair. The application can then use the iterator to access each name-value pair in turn.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetObject – Get Object

Interface:

```
xmsRC xmsMapMsgGetObject(xmsHMsg message,
                         xmsCHAR *name,
                         xmsSBYTE *buffer,
                         xmsINT bufferLength,
                         xmsINT *actualLength,
                         xmsOBJECT_TYPE *objectType,
                         xmsHErrorBlock errorBlock);
```

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 function, see "C functions that return a byte array by value" on page 59.

Parameters:

message (input)
The handle for the message.

name (input)
The name of the name-value pair in the format of a null terminated string.

buffer (output)
The buffer to contain the value, which is returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value in bytes. If the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

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_BYTEARRAY
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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetShort – Get Short Integer

Interface:

```
xmsRC xmsMapMsgGetShort(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSHORT *value,
                        xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetString – Get String

Interface:

```
xmsRC xmsMapMsgGetString(xmsHMsg message,
                         xmsCHAR *name,
                         xmsCHAR *buffer,
                         xmsINT bufferLength,
                         xmsINT *actualLength,
                         xmsHErrorBlock errorBlock);
```

Get the string identified by name from the body of the map message.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgGetStringByRef – Get String by Reference

Interface:

```
xmsRC xmsMapMsgGetStringByRef(xmsHMsg message,
                              xmsCHAR *name,
                              xmsCHAR **string,
                              xmsINT *length,
                              xmsHErrorBlock errorBlock);
```

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 60.

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:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgItemExists – Check Name-Value Pair Exists

Interface:

```
xmsRC xmsMapMsgItemExists(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL *pairExists,
                          xmsHErrorBlock errorBlock);
```

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 format of a null terminated string.

pairExists (output)
The value is `xmsTRUE` if the body of the map message contains a

name-value pair with the specified name. The value is xmsFALSE if 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetBoolean – Set Boolean Value

Interface:

```
xmsRC xmsMapMsgSetBoolean(xmsHMsg message,
                          xmsCHAR *name,
                          xmsBOOL value,
                          xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The boolean value to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetByte – Set Byte

Interface:

```
xmsRC xmsMapMsgSetByte(xmsHMsg message,
                       xmsCHAR *name,
                       xmsSBYTE value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The byte to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetBytes – Set Bytes

Interface:

```
xmsRC xmsMapMsgSetBytes(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSBYTE *value,
                        xmsINT length,
                        xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetChar – Set Character

Interface:

```
xmsRC xmsMapMsgSetChar(xmsHMsg message,
                       xmsCHAR *name,
                       xmsCHAR16 value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The character to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetDouble – Set Double Precision Floating Point Number

Interface:

```
xmsRC xmsMapMsgSetDouble(xmsHMsg message,
                         xmsCHAR *name,
                         xmsDOUBLE value,
                         xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetFloat – Set Floating Point Number

Interface:

```
xmsRC xmsMapMsgSetFloat(xmsHMsg message,
                        xmsCHAR *name,
                        xmsFLOAT value,
                        xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

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

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetInt – Set Integer

Interface:

```
xmsRC xmsMapMsgSetInt(xmsHMsg message,
                      xmsCHAR *name,
                      xmsINT value,
                      xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The integer to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetLong – Set Long Integer

Interface:

```
xmsRC xmsMapMsgSetLong(xmsHMsg message,
                       xmsCHAR *name,
                       xmsLONG value,
                       xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The long integer to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetObject – Set Object

Interface:

```
xmsRC xmsMapMsgSetObject(xmsHMsg message,
                         xmsCHAR *name,
                         xmsSBYTE *value,
                         xmsINT length,
                         xmsOBJECT_TYPE objectType,
                         xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
An array of bytes containing 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 object types:

XMS_OBJECT_TYPE_BOOL
XMS_OBJECT_TYPE_BYTE
XMS_OBJECT_TYPE_BYTEARRAY
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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetShort – Set Short Integer

Interface:

```
xmsRC xmsMapMsgSetShort(xmsHMsg message,
                        xmsCHAR *name,
                        xmsSHORT value,
                        xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
The short integer to be set.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMapMsgSetString – Set String

Interface:

```
xmsRC xmsMapMsgSetString(xmsHMsg message,
                         xmsCHAR *name,
                         xmsCHAR *value,
                         xmsINT length,
                         xmsHErrorBlock errorBlock);
```

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. The name is in the format of a null terminated string.

value (input)
A character array containing the string to be set.

length (input)
The length of the string in bytes. If the string is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

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 91. For a list of the JMS defined properties of a Message object, see "JMS-defined properties of a message" on page 93. For a list of the IBM defined properties of a Message object, see "IBM-defined properties of a message" on page 94.

Functions

Summary of functions:

Function	Description
xmsMsgAcknowledge	Acknowledge this message and all previously unacknowledged messages received by the session.
xmsMsgClearBody	Clear the body of the message.
xmsMsgClearProperties	Clear the properties of the message.
xmsMsgDispose	Delete the message.
xmsMsgGetJMSCorrelationID	Get the correlation identifier of the message.
xmsMsgGetJMSDeliveryMode	Get the delivery mode of the message.
xmsMsgGetJMSDestination	Get the destination of the message.
xmsMsgGetJMSExpiration	Get the expiration time of the message.
xmsMsgGetJMSMessageID	Get the message identifier of the message.
xmsMsgGetJMSPriority	Get the priority of the message.
xmsMsgGetJMSRedelivered	Get an indication of whether the message is being re-delivered.
xmsMsgGetJMSReplyTo	Get the destination where a reply to the message is to be sent.
xmsMsgGetJMSTimestamp	Get the time when the message was sent.
xmsMsgGetJMSType	Get the type of the message.
xmsMsgGetProperties	Get a list of the properties of the message.
xmsMsgGetTypeId	Get the body type of the message.
xmsMsgPropertyExists	Check whether the message has a property with the specified name.
xmsMsgSetJMSCorrelationID	Set the correlation identifier of the message.
xmsMsgSetJMSDeliveryMode	Set the delivery mode of the message.
xmsMsgSetJMSDestination	Set the destination of the message.
xmsMsgSetJMSExpiration	Set the expiration time of the message.
xmsMsgSetJMSMessageID	Set the message identifier of the message.
xmsMsgSetJMSPriority	Set the priority of the message.
xmsMsgSetJMSRedelivered	Indicate whether the message is being re-delivered.
xmsMsgSetJMSReplyTo	Set the destination where a reply to the message is to be sent.
xmsMsgSetJMSTimestamp	Set the time when the message is sent.
xmsMsgSetJMSType	Set the type of the message.

xmsMsgAcknowledge – Acknowledge

Interface:

```
xmsRC xmsMsgAcknowledge(xmsHMsg message,
                        xmsHErrorBlock errorBlock);
```

Acknowledge this message and all previously unacknowledged messages received by the session.

An application can call this function if the acknowledgement mode of the session is XMSC_CLIENT_ACKNOWLEDGE. Calls to the function are ignored if the session has any other acknowledgement mode or is transacted.

Messages that have been received but not acknowledged might be re-delivered.

For more information about acknowledging messages, see "Message acknowledgement" on page 29.

Parameters:

message (input)
The handle for the message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

xmsMsgClearBody – Clear Body

Interface:

```
xmsRC xmsMsgClearBody(xmsHMsg message,
                      xmsHErrorBlock errorBlock);
```

Clear the body of the message. The header fields and message properties are not cleared.

If an application clears a message body, the body is left in the same state as an empty body in a newly created message. The state of an empty body in a newly created message depends on the type of message body. For more information, see "The body of an XMS message" on page 95.

An application can clear a message body at any time, no matter what state the body is in. If a message body is read-only, the only way that an application can write to the body is for the application to clear the body first.

Parameters:

message (input)
The handle for the message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsMsgClearProperties – Clear Properties

Interface:

```
xmsRC xmsMsgClearProperties(xmsHMsg message,
                            xmsHErrorBlock errorBlock);
```

Clear the properties of the message. The header fields and the message body are not cleared.

If an application clears the properties of a message, the properties become readable and writable.

An application can clear the properties of a message at any time, no matter what state the properties are in. If the properties of a message are read-only, the only way that the properties can become writable is for the application to clear the properties first.

Parameters:

message (input)
The handle for the message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgDispose – Delete Message

Interface:

```
xmsRC xmsMsgDispose(xmsHMsg *message,
                    xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSCorrelationID – Get JMSCorrelationID

Interface:

```
xmsRC xmsMsgGetJMSCorrelationID(xmsHMsg message,
                                xmsCHAR *correlID,
                                xmsINT length,
                                xmsINT *actualLength,
                                xmsHErrorBlock errorBlock);
```

Get the correlation identifier of the message.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 0, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSDeliveryMode – Get JMSDeliveryMode

Interface:

```
xmsRC xmsMsgGetJMSDeliveryMode(xmsHMsg message,
                               xmsINT *deliveryMode,
                               xmsHErrorBlock errorBlock);
```

Get the delivery mode of the message. The delivery mode is set by the xmsMsgProducerSend() 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_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

For a newly created message that has not been sent, the delivery mode is XMSC_DELIVERY_PERSISTENT For a message that has been received, the function returns the delivery mode that was set by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the delivery mode by calling xmsMsgSetJMSDeliveryMode().

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSDestination – Get JMSDestination

Interface:

```
xmsRC xmsMsgGetJMSDestination(xmsHMsg message,
                              xmsHDest *destination,
                              xmsHErrorBlock errorBlock);
```

Get the destination of the message. The destination is set by the xmsMsgProducerSend() 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 newly created message that has not been sent, the function returns a null handle and an error unless the sending application sets a destination by calling xmsMsgSetJMSDestination(). For a message that has been received, the function returns a handle for the destination that was set by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the destination by calling xmsMsgSetJMSDestination().

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSExpiration – Get JMSExpiration

Interface:

```
xmsRC xmsMsgGetJMSExpiration(xmsHMsg message,
                             xmsLONG *expiration,
                             xmsHErrorBlock errorBlock);
```

Get the expiration time of the message.

The expiration time is set by the xmsMsgProducerSend() 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 0, the xmsMsgProducerSend() call sets the expiration time to 0 to indicate that the message does not expire.

XMS discards expired messages and does not deliver them to applications.

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 0 unless the sending application sets a different expiration time by calling xmsMsgSetJMSExpiration(). For a message that has been received, the function returns the expiration time that was set

by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the expiration time by calling xmsMsgSetJMSExpiration().

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSMessageID – Get JMSMessageID

Interface:

```
xmsRC xmsMsgGetJMSMessageID(xmsHMsg message,
                            xmsCHAR *msgID,
                            xmsINT length,
                            xmsINT *actualLength,
                            xmsHErrorBlock errorBlock);
```

Get the message identifier of the message. The message identifier is set by the xmsMsgProducerSend() call when the message is sent.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 function returns the message identifier that was set by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the message identifier by calling xmsMsgSetJMSMessageID().

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

XMS_X_GENERAL_EXCEPTION

Notes:

1. If a message has no message identifier, the function leaves the contents of the buffer unchanged, sets the actualLength parameter to 0, and returns an error.

xmsMsgGetJMSPriority – Get JMSPriority

Interface:

```
xmsRC xmsMsgGetJMSPriority(xmsHMsg message,
                           xmsINT *priority,
                           xmsHErrorBlock errorBlock);
```

Get the priority of the message. The priority is set by the xmsMsgProducerSend() 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 xmsMsgSetJMSPriority(). For a message that has been received, the function returns the priority that was set by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the priority by calling xmsMsgSetJMSPriority().

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSRedelivered – Get JMSRedelivered

Interface:

```
xmsRC xmsMsgGetJMSRedelivered(xmsHMsg message,
                              xmsBOOL *redelivered,
                              xmsHErrorBlock errorBlock);
```

Get an indication of whether the message is being re-delivered. The indication is set by the xmsMsgConsumerReceive() call when the message is received.

Parameters:

message (input)
The handle for the message.

redelivered (output)
The value is xmsTRUE if the message is being re-delivered. The value is xmsFALSE if the message is not being re-delivered.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSReplyTo – Get JMSReplyTo

Interface:

```
xmsRC xmsMsgGetJMSReplyTo(xmsHMsg message,
                          xmsHDest *destination,
                          xmsHErrorBlock errorBlock);
```

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. A null handle means that no reply is expected.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetJMSTimestamp – Get JMSTimestamp

Interface:

```
xmsRC xmsMsgGetJMSTimestamp(xmsHMsg message,
                            xmsLONG *timeStamp,
                            xmsHErrorBlock errorBlock);
```

Get the time when the message was sent. The time stamp is set by the xmsMsgProducerSend() 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 time stamp is 0 unless the sending application sets a different time stamp by calling xmsMsgSetJMSTimestamp(). For a message that has been received, the function returns the time stamp that was set by the xmsMsgProducerSend() call when the message was sent unless the receiving application changes the time stamp by calling xmsMsgSetJMSTimestamp().

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Note:

1. If the time stamp is undefined, the function sets the timeStamp parameter to 0 but returns no error.

xmsMsgGetJMSType – Get JMSType

Interface:

```
xmsRC xmsMsgGetJMSType(xmsHMsg message,
                       xmsCHAR *type,
                       xmsINT length,
                       xmsINT *actualLength,
                       xmsHErrorBlock errorBlock);
```

Get the type of the message.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetProperties – Get Properties

Interface:

```
xmsRC xmsMsgGetProperties(xmsHMsg message,
                          xmsHIterator *iterator,
                          xmsHErrorBlock errorBlock);
```

Get a list of the properties of the message.

The function returns an iterator that encapsulates a list of Property objects. The application can then use the iterator to access each property 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 (output)
The handle for the iterator.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgGetTypeId – Get Type

Interface:

```
xmsRC xmsMsgGetTypeId(xmsHMsg message,
                      xmsMESSAGE_TYPE *type,
                      xmsHErrorBlock errorBlock);
```

Get the body type of the message.

For information about message body types, see "The body of an XMS message" on page 95.

Parameters:

message (input)
The handle for the message.

type (output)
The body type of the message, which is one of the following values:

XMS_MESSAGE_TYPE_BASE (the message has no body)
XMS_MESSAGE_TYPE_BYTES
XMS_MESSAGE_TYPE_MAP
XMS_MESSAGE_TYPE_OBJECT
XMS_MESSAGE_TYPE_STREAM
XMS_MESSAGE_TYPE_TEXT

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgPropertyExists – Check Property Exists

Interface:

```
xmsRC xmsMsgPropertyExists(xmsHMsg message,
                           xmsCHAR *propertyName,
                           xmsBOOL *propertyExists,
                           xmsHErrorBlock errorBlock);
```

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 format of a null terminated string.

propertyExists (output)
The value is xmsTRUE if the message has a property with the specified name. The value is xmsFALSE if the message does not have a property with the specified name.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSCorrelationID – Set JMSCorrelationID

Interface:

```
xmsRC xmsMsgSetJMSCorrelationID(xmsHMsg message,
                                xmsCHAR *correlID,
                                xmsINT length,
                                xmsHErrorBlock errorBlock);
```

Set the correlation identifier of the message.

Parameters:

message (input)
The handle for the message.

correlID (input)
The correlation identifier as a character array.

length (input)
The length of the correlation identifier in bytes. If the correlation identifier is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSDeliveryMode – Set JMSDeliveryMode

Interface:

```
xmsRC xmsMsgSetJMSDeliveryMode(xmsHMsg message,
                               xmsINT deliveryMode,
                               xmsHErrorBlock errorBlock);
```

Set the delivery mode of the message.

A delivery mode set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function 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, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSDestination – Set JMSDestination

Interface:

```
xmsRC xmsMsgSetJMSDestination(xmsHMsg message,
                              xmsHDest destination,
                              xmsHErrorBlock errorBlock);
```

Set the destination of the message.

A destination set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function to change the destination of a message that has been received.

Parameters:

message (input)
The handle for the message.

destination (input)
The handle for the destination of the message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSExpiration – Set JMSExpiration

Interface:

```
xmsRC xmsMsgSetJMSExpiration(xmsHMsg message,
                             xmsLONG expiration,
                             xmsHErrorBlock errorBlock);
```

Set the expiration time of the message.

An expiration time set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSMessageID – Set JMSMessageID

Interface:

```
xmsRC xmsMsgSetJMSMessageID(xmsHMsg message,
                            xmsCHAR *msgID,
                            xmsINT length,
                            xmsHErrorBlock errorBlock);
```

Set the message identifier of the message.

A message identifier set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function 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 as a character array.

length (input)
The length of the message identifier in bytes. If the message identifier is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSPriority – Set JMSPriority

Interface:

```
xmsRC xmsMsgSetJMSPriority(xmsHMsg message,
                           xmsINT priority,
                           xmsHErrorBlock errorBlock);
```

Set the priority of the message.

A priority set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSRedelivered – Set JMSRedelivered

Interface:

```
xmsRC xmsMsgSetJMSRedelivered(xmsHMsg message,
                              xmsBOOL redelivered,
                              xmsHErrorBlock errorBlock);
```

Indicate whether the message is being re-delivered.

An indication of re-delivery set by this function before the message is sent is ignored by the xmsMsgProducerSend() call when the message is sent, and is ignored and replaced by the xmsMsgConsumerReceive() call when the message is received. However, you can use this function 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSReplyTo – Set JMSReplyTo

Interface:

```
xmsRC xmsMsgSetJMSReplyTo(xmsHMsg message,
                          xmsHDest destination,
                          xmsHErrorBlock errorBlock);
```

Set the destination where a reply to the message is to be sent.

Parameters:

message (input)
The handle for the message.

destination (input)
The handle for the destination where a reply to the message is to be sent. A null handle means that no reply is expected.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSTimestamp – Set JMSTimestamp

Interface:

```
xmsRC xmsMsgSetJMSTimestamp(xmsHMsg message,
                            xmsLONG timeStamp,
                            xmsHErrorBlock errorBlock);
```

Set the time when the message is sent.

A time stamp set by this function before the message is sent is ignored and replaced by the xmsMsgProducerSend() call when the message is sent. However, you can use this function to change the time stamp 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgSetJMSType – Set JMSType

Interface:

```
xmsRC xmsMsgSetJMSType(xmsHMsg message,
                       xmsCHAR *type,
                       xmsINT length,
                       xmsHErrorBlock errorBlock);
```

Set the type of the message.

Parameters:

message (input)
The handle for the message.

type (input)
The type of the message as a character array.

length (input)
The length of the type of the message in bytes. If the type of the message is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

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 414.

Functions

Summary of functions:

Function	Description
xmsMsgConsumerClose	Close the message consumer.
xmsMsgConsumerGetMessageListener	Get pointers to the message listener function and context data that are registered with the message consumer.
xmsMsgConsumerGetMessageSelector	Get the message selector for the message consumer.
xmsMsgConsumerReceive	Receive the next message for the message consumer. The call waits indefinitely for a message, or until the message consumer is closed.
xmsMsgConsumerReceiveNoWait	Receive the next message for the message consumer if one is available immediately.
xmsMsgConsumerReceiveWithWait	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.
xmsMsgConsumerSetMessageListener	Register a message listener function and context data with the message consumer.

xmsMsgConsumerClose – Close Message Consumer

Interface:

```
xmsRC xmsMsgConsumerClose(xmsHMsgConsumer *consumer,
                          xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerGetMessageListener – Get Message Listener

Interface:

```
xmsRC xmsMsgConsumerGetMessageListener(xmsHMsgConsumer consumer,
                                       fpXMS_MESSAGE_CALLBACK *lsr,
                                       xmsCONTEXT *context,
                                       xmsHErrorBlock errorBlock);
```

Get pointers to the message listener function and context data that are registered with the message consumer.

For more information about using message listener functions, see "Message listener functions in C" on page 62.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerGetMessageSelector – Get Message Selector

Interface:

```
xmsRC xmsMsgConsumerGetMessageSelector(xmsHMsgConsumer consumer,
                                       xmsCHAR *messageSelector,
                                       xmsINT length,
                                       xmsINT *actualLength,
                                       xmsHErrorBlock errorBlock);
```

Get the message selector for the message consumer.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

consumer (input)
The handle for the message consumer.

messageSelector (output)
The buffer to contain the message selector expression. If data conversion is required, this is the message selector expression after conversion.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If data conversion is required, this is the length of the message selector expression after conversion. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerReceive – Receive

Interface:

```
xmsRC xmsMsgConsumerReceive(xmsHMsgConsumer consumer,
                            xmsHMsg *message,
                            xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerReceiveNoWait – Receive with No Wait

Interface:

```
xmsRC xmsMsgConsumerReceiveNoWait(xmsHMsgConsumer consumer,
                                  xmsHMsg *message,
                                  xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerReceiveWithWait – Receive (with a wait interval)

Interface:

```
xmsRC xmsMsgConsumerReceiveWithWait(xmsHMsgConsumer consumer,
                                    xmsLONG waitInterval,
                                    xmsHMsg *message,
                                    xmsHErrorBlock errorBlock);
```

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 0, 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 function returns a null handle but no error.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgConsumerSetMessageListener – Set Message Listener

Interface:

```
xmsRC xmsMsgConsumerSetMessageListener(xmsHMsgConsumer consumer,
                                       fpXMS_MESSAGE_CALLBACK lsr,
                                       xmsCONTEXT context,
                                       xmsHErrorBlock errorBlock);
```

Register a message listener function and context data with the message consumer.

For more information about using message listener functions, see "Message listener functions in C" on page 62.

Parameters:

consumer (input)
The handle for the message consumer.

lsr (input)
A pointer to the message listener function. If a message listener function 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

MessageListener

An application uses a message listener to receive messages asynchronously.

Functions

Summary of functions:

Function	Description
onMessage	Deliver a message asynchronously to the message consumer.

onMessage – On Message

Interface:

```
xmsVOID onMessage(xmsCONTEXT context,
                  xmsHMsg message);
```

Deliver a message asynchronously to the message consumer.

onMessage() is the message listener function that is registered with the message consumer. The name of the function does not have to be onMessage.

For more information about using message listener functions, see "Message listener functions in C" on page 62.

Parameters:

context (input)
A pointer to the context data that is registered with the message consumer.

message (input)
The handle for the message.

MessageProducer

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

For a list of the XMS defined properties of a MessageProducer object, see "Properties of MessageProducer" on page 414.

Functions

Summary of functions:

Function	Description
xmsMsgProducerClose	Close the message producer.
xmsMsgProducerGetDeliveryMode	Get the default delivery mode for messages sent by the message producer.
xmsMsgProducerGetDestination	Get the destination for the message producer.
xmsMsgProducerGetDisableMsgID	Get an indication of whether a receiving application requires message identifiers to be included in messages sent by the message producer.
xmsMsgProducerGetDisableMsgTS	Get an indication of whether a receiving application requires time stamps to be included in messages sent by the message producer.
xmsMsgProducerGetPriority	Get the default priority for messages sent by the message producer.
xmsMsgProducerGetTimeToLive	Get the default length of time that a message exists before it expires.
xmsMsgProducerSend	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.
xmsMsgProducerSendDest	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.
xmsMsgProducerSendDestWithASte	trnd 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.
xmsMsgProducerSendWithAttr	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.
xmsMsgProducerSetDeliveryMode	Set the default delivery mode for messages sent by the message producer.
xmsMsgProducerSetDisableMsgID	Indicate whether a receiving application requires message identifiers to be included in messages sent by the message producer.
xmsMsgProducerSetDisableMsgTS	Indicate whether a receiving application requires time stamps to be included in messages sent by the message producer.
xmsMsgProducerSetPriority	Set the default priority for messages sent by the message producer.
xmsMsgProducerSetTimeToLive	Set the default length of time that a message exists before it expires.

xmsMsgProducerClose – Close Message Producer

Interface:

```
xmsRC xmsMsgProducerClose(xmsHMsgProducer *producer,
                          xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetDeliveryMode – Get Default Delivery Mode

Interface:

```
xmsRC xmsMsgProducerGetDeliveryMode(xmsHMsgProducer producer,
                                    xmsINT *deliveryMode,
                                    xmsHErrorBlock errorBlock);
```

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, which is one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetDestination – Get Destination

Interface:

```
xmsRC xmsMsgProducerGetDestination(xmsHMsgProducer producer,
                                   xmsHDest *destination,
                                   xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetDisableMsgID – Get Disable Message ID Flag

Interface:

```
xmsRC xmsMsgProducerGetDisableMsgID(xmsHMsgProducer producer,
                                    xmsBOOL *msgIDDisabled,
                                    xmsHErrorBlock errorBlock);
```

Get an indication of whether a receiving application requires message identifiers to be included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

msgIDDisabled (output)
The value is xmsTRUE if a receiving application does not require message identifiers to be included in messages sent by the message producer. The value is xmsFALSE if a receiving application does require message identifiers.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetDisableMsgTS – Get Disable Time Stamp Flag

Interface:

```
xmsRC xmsMsgProducerGetDisableMsgTS(xmsHMsgProducer producer,
                                    xmsBOOL *timeStampDisabled,
                                    xmsHErrorBlock errorBlock);
```

Get an indication of whether a receiving application requires time stamps to be included in messages sent by the message producer.

Parameters:

producer (input)
The handle for the message producer.

timeStampDisabled (output)
The value is xmsTRUE if a receiving application does not require

time stamps to be included in messages sent by the message producer. The value is xmsFALSE if a receiving application does require time stamps.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetPriority – Get Default Priority

Interface:

```
xmsRC xmsMsgProducerGetPriority(xmsHMsgProducer producer,
                                xmsINT *priority,
                                xmsHErrorBlock errorBlock);
```

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerGetTimeToLive – Get Default Time to Live

Interface:

```
xmsRC xmsMsgProducerGetTimeToLive(xmsHMsgProducer producer,
                                  xmsLONG *timeToLive,
                                  xmsHErrorBlock errorBlock);
```

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 0 means that a message never expires.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerSend – Send

Interface:

```
xmsRC xmsMsgProducerSend(xmsHMsgProducer producer,
                         xmsHMsg message,
                         xmsHErrorBlock errorBlock);
```

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)
The handle for the message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsMsgProducerSendDest – Send (to a specified destination)

Interface:

```
xmsRC xmsMsgProducerSendDest(xmsHMsgProducer producer,
                             xmsHDest destination,
                             xmsHMsg message,
                             xmsHErrorBlock errorBlock);
```

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)
The handle for the destination.

message (input)
The handle for the message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsMsgProducerSendDestWithAttr – Send (to a specified destination, specifying a delivery mode, priority, and time to live)

Interface:

```
xmsRC xmsMsgProducerSendDestWithAttr(xmsHMsgProducer producer,
                                     xmsHDest destination,
                                     xmsHMsg message,
                                     xmsINT deliveryMode,
                                     xmsINT priority,
                                     xmsLONG timeToLive,
                                     xmsHErrorBlock errorBlock);
```

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)
The handle for the destination.

message (input)
The handle for the message.

deliveryMode (input)
The delivery mode for the message, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_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.

timeToLive (input)
The time to live for the message in milliseconds. A value of 0 means that the message never expires.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION

v XMS_X_INVALID_DESTINATION_EXCEPTION
v XMS_X_ILLEGAL_STATE_EXCEPTION

xmsMsgProducerSendWithAttr – Send (specifying a delivery mode, priority, and time to live)

Interface:

```
xmsRC xmsMsgProducerSendWithAttr(xmsHMsgProducer producer,
                                 xmsHMsg message,
                                 xmsINT deliveryMode,
                                 xmsINT priority,
                                 xmsLONG timeToLive,
                                 xmsHErrorBlock errorBlock);
```

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)
The handle for the message.

deliveryMode (input)
The delivery mode for the message, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_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.

timeToLive (input)
The time to live for the message in milliseconds. A value of 0 means that the message never expires.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_FORMAT_EXCEPTION
v XMS_X_INVALID_DESTINATION_EXCEPTION
v XMS_X_ILLEGAL_STATE_EXCEPTION

xmsMsgProducerSetDeliveryMode – Set Default Delivery Mode

Interface:

```
xmsRC xmsMsgProducerSetDeliveryMode(xmsHMsgProducer producer,
                                    xmsINT deliveryMode,
                                    xmsHErrorBlock errorBlock);
```

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, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

The default value is XMSC_DELIVERY_PERSISTENT.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerSetDisableMsgID – Set Disable Message ID Flag

Interface:

```
xmsRC xmsMsgProducerSetDisableMsgID(xmsHMsgProducer producer,
                                    xmsBOOL msgIDDisabled,
                                    xmsHErrorBlock errorBlock);
```

Indicate whether a receiving application requires message identifiers to be included in messages sent by the message producer.

On a connection to a queue manager, this flag is ignored. On a connection to a service integration bus, the flag is honoured.

Parameters:

producer (input)
The handle for the message producer.

msgIDDisabled (input)
The value xmsTRUE means that a receiving application does not require message identifiers to be included in messages sent by the message producer. The value xmsFALSE means that a receiving application does require message identifiers. The default value is xmsFALSE.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerSetDisableMsgTS – Set Disable Time Stamp Flag

Interface:

```
xmsRC xmsMsgProducerSetDisableMsgTS(xmsHMsgProducer producer,
                                    xmsBOOL timeStampDisabled,
                                    xmsHErrorBlock errorBlock);
```

Indicate whether a receiving application requires time stamps to be included in messages sent by the message producer.

On a connection to a queue manager, or on a connection to a service integration bus, the flag is honoured.

Parameters:

producer (input)
The handle for the message producer.

timeStampDisabled (input)
The value `xmsTRUE` means that a receiving application does not require time stamps to be included in messages sent by the message producer. The value `xmsFALSE` means that a receiving application does require time stamps. The default value is `xmsFALSE`.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerSetPriority – Set Default Priority

Interface:

```
xmsRC xmsMsgProducerSetPriority(xmsHMsgProducer producer,
                                xmsINT priority,
                                xmsHErrorBlock errorBlock);
```

Set the default priority for messages sent by the message producer.

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsMsgProducerSetTimeToLive – Set Default Time to Live

Interface:

```
xmsRC xmsMsgProducerSetTimeToLive(xmsHMsgProducer producer,
                                  xmsLONG timeToLive,
                                  xmsHErrorBlock errorBlock);
```

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. The default value is 0, which means that a message never expires.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

ObjectMessage

An object message is a message whose body comprises a serialized Java object.

Functions

Summary of functions:

Function	Description
xmsObjectMsgGetObjectAsBytes	Get the object that forms the body of the object message.
xmsObjectMsgSetObjectAsBytes	Set the object that forms the body of the object message.

xmsObjectMsgGetObjectAsBytes – Get Object as Bytes

Interface:

```
xmsRC xmsObjectMsgGetObjectAsBytes(xmsHMsg message,
                                   xmsSBYTE *buffer,
                                   xmsINT bufferLength,
                                   xmsINT *actualLength,
                                   xmsHErrorBlock errorBlock);
```

Get the object that forms the body of the object message.

For more information about how to use this function, see “C functions that return a byte array by value” on page 59.

Parameters:

message (input)
The handle for the message.

buffer (output)
The buffer to contain the object, which is returned as an array of bytes.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the object is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the object in bytes. If you specify a null pointer on input, the length is not returned.

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

Notes:

1. If the buffer is not large enough to store the whole object, XMS returns the object truncated to the length of the buffer, sets the actualLength parameter to the actual length of the object, and returns an error.
2. If any other error occurs while attempting to get the object, XMS reports the error but does not set the actualLength parameter.

xmsObjectMsgSetObjectAsBytes – Set Object as Bytes

Interface:

```
xmsRC xmsObjectMsgSetObjectAsBytes(xmsHMsg message,
                                   xmsSBYTE *value,
                                   xmsINT length,
                                   xmsHErrorBlock errorBlock);
```

Set the object that forms the body of the object message.

Parameters:

message (input)
The handle for the message.

value (input)
An array of bytes representing the object to be set.

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

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Property

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

If an application sets the property value attribute of a Property object, the property value replaces any previous value the attribute had.

This class is a helper class.

Functions

Summary of functions:

Function	Description
xmsPropertyCreate	Create a Property object with no property value or property type.
xmsPropertyDispose	Delete the Property object.
xmsPropertyDuplicate	Copy the Property object.
xmsPropertyGetBoolean	Get the boolean property value from the Property object.
xmsPropertyGetByte	Get the byte property value from the Property object.
xmsPropertyGetByteArray	Get the byte array property value from the Property object.
xmsPropertyGetByteArrayByRef	Get a pointer to the byte array property value in the Property object.
xmsPropertyGetChar	Get the 2-byte character property value from the Property object.
xmsPropertyGetDouble	Get the double precision floating point property value from the Property object.
xmsPropertyGetFloat	Get the floating point property value from the Property object.
xmsPropertyGetInt	Get the integer property value from the Property object.
xmsPropertyGetLong	Get the long integer property value from the Property object.
xmsPropertyGetName	Get the property name from the Property object.
xmsPropertyGetShort	Get the short integer property value from the Property object.
xmsPropertyGetString	Get the string property value from the Property object.
xmsPropertyGetStringByRef	Get a pointer to the string property value in the Property object.
xmsPropertyGetTypeId	Get the property type from the Property object.
xmsPropertyIsTypeId	Check whether the Property object has the specified property type.
xmsPropertySetBoolean	Set a boolean property value in the Property object and set the property type.
xmsPropertySetByte	Set a byte property value in the Property object and set the property type.
xmsPropertySetByteArray	Set a byte array property value in the Property object and set the property type.

Function	Description
xmsPropertySetChar	Set a 2-byte character property value in the Property object and set the property type.
xmsPropertySetDouble	Set a double precision floating point property value in the Property object and set the property type.
xmsPropertySetFloat	Set a floating point property value in the Property object and set the property type.
xmsPropertySetInt	Set an integer property value in the Property object and set the property type.
xmsPropertySetLong	Set a long integer property value in the Property object and set the property type.
setShortxmsPropertySetShort	Set a short integer property value in the Property object and set the property type.
xmsPropertySetString	Set a string property value in the Property object and set the property type.

xmsPropertyCreate – Create Property (with no property value or property type)

Interface:

```
xmsRC xmsPropertyCreate(xmsCHAR *propertyName,
                        xmsHProperty *property,
                        xmsHErrorBlock errorBlock);
```

Create a Property object with no property value or property type.

Parameters:

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

property (output)
The handle for the Property object.

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

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyDispose – Delete Property

Interface:

```
xmsRC xmsPropertyDispose(xmsHProperty *property,
                         xmsHErrorBlock errorBlock);
```

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 function returns a null handle.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyDuplicate – Copy Property

Interface:

```
xmsRC xmsPropertyDuplicate(xmsHProperty property,
                           xmsHProperty *copiedProperty,
                           xmsHErrorBlock errorBlock);
```

Copy the Property object.

Parameters:

property (input)
The handle for 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:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetBoolean – Get Boolean Property Value

Interface:

```
xmsRC xmsPropertyGetBoolean(xmsHProperty property,
                            xmsBOOL *propertyValue,
                            xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetByte – Get Byte Property Value

Interface:

```
xmsRC xmsPropertyGetByte(xmsHProperty property,
                         xmsSBYTE *propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetByteArray – Get Byte Array Property Value

Interface:

```
xmsRC xmsPropertyGetByteArray(xmsHProperty property,
                              xmsSBYTE *propertyValue,
                              xmsINT length,
                              xmsINT *actualLength,
                              xmsHErrorBlock errorBlock);
```

Get the byte array property value from the Property object.

For more information about how to use this function, see "C functions that return a byte array by value" on page 59.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetByteArrayByRef – Get Byte Array Property Value by Reference

Interface:

```
xmsRC xmsPropertyGetByteArrayByRef(xmsHProperty property,
                                   xmsSBYTE **propertyValue,
                                   xmsINT *length,
                                   xmsHErrorBlock errorBlock);
```

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 60.

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:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetChar – Get Character Property Value

Interface:

```
xmsRC xmsPropertyGetChar(xmsHProperty property,
                         xmsCHAR16 *propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetDouble – Get Double Precision Floating Point Property Value

Interface:

```
xmsRC xmsPropertyGetDouble(xmsHProperty property,
                           xmsDOUBLE *propertyValue,
                           xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetFloat – Get Floating Point Property Value

Interface:

```
xmsRC xmsPropertyGetFloat(xmsHProperty property,
                          xmsFLOAT *propertyValue,
                          xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetInt – Get Integer Property Value

Interface:

```
xmsRC xmsPropertyGetInt(xmsHProperty property,
                        xmsINT *propertyValue,
                        xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetLong – Get Long Integer Property Value

Interface:

```
xmsRC xmsPropertyGetLong(xmsHProperty property,
                         xmsLONG *propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetName – Get Property Name

Interface:

```
xmsRC xmsPropertyGetName(xmsHProperty property,
                         xmsCHAR *propertyName,
                         xmsINT length,
                         xmsINT *actualLength,
                         xmsHErrorBlock errorBlock);
```

Get the property name from the Property object.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetShort – Get Short Integer Property Value

Interface:

```
xmsRC xmsPropertyGetShort(xmsHProperty property,
                          xmsSHORT *propertyValue,
                          xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetString – Get String Property Value

Interface:

```
xmsRC xmsPropertyGetString(xmsHProperty property,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsINT *actualLength,
                           xmsHErrorBlock errorBlock);
```

Get the string property value from the Property object.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
The buffer to contain the string property value. If data conversion is required, this is the value after conversion.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

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

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertyGetStringByRef – Get String Property Value by Reference

Interface:

```
xmsRC xmsPropertyGetStringByRef(xmsHProperty property,
                                xmsCHAR **propertyValue,
                                xmsINT *length,
                                xmsHErrorBlock errorBlock);
```

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 60.

Parameters:

property (input)
The handle for the Property object.

propertyValue (output)
A pointer to the string property value. If data conversion is required, this is the value after conversion.

Note that the property value must be a string. The function makes no attempt to convert a property value with another data type into a string. If an application calls this function to get a pointer to a property value that is not a string, XMS returns an error.

length (output)
The length of the property value 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.

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

xmsPropertyGetTypeId – Get Property Type

Interface:

```
xmsRC xmsPropertyGetTypeId(xmsHProperty property,
                           xmsPROPERTY_TYPE *propertyType,
                           xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

xmsPropertyIsTypeId – Check Property Type

Interface:

```
xmsRC xmsPropertyIsTypeId(xmsHProperty property,
                          xmsPROPERTY_TYPE propertyType,
                          xmsBOOL *isType,
                          xmsHErrorBlock errorBlock);
```

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)
The value is xmsTRUE if the Property object has the specified property type. The value is xmsFALSE if the Property object does not have the specified property type.

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

Thread context:
Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsPropertySetBoolean – Set Boolean Property Value

Interface:

```
xmsRC xmsPropertySetBoolean(xmsHProperty property,
                            xmsBOOL propertyValue,
                            xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetByte – Set Byte Property Value

Interface:

```
xmsRC xmsPropertySetByte(xmsHProperty property,
                         xmsSBYTE propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetByteArray – Set Byte Array Property Value

Interface:

```
xmsRC xmsPropertySetByteArray(xmsHProperty property,
                              xmsSBYTE *propertyValue,
                              xmsINT length,
                              xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetChar – Set Character Property Value

Interface:

```
xmsRC xmsPropertySetChar(xmsHProperty Property,
                         xmsCHAR16 propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetDouble – Set Double Precision Floating Point Property Value

Interface:

```
xmsRC xmsPropertySetDouble(xmsHProperty property,
                           xmsDOUBLE propertyValue,
                           xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertySetFloat – Set Floating Point Property Value

Interface:

```
xmsRC xmsPropertySetFloat(xmsHProperty property,
                          xmsFLOAT propertyValue,
                          xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertySetInt – Set Integer Property Value

Interface:

```
xmsRC xmsPropertySetInt(xmsHProperty property,
                        xmsINT propertyValue,
                        xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsPropertySetLong – Set Long Integer Property Value

Interface:

```
xmsRC xmsPropertySetLong(xmsHProperty property,
                         xmsLONG propertyValue,
                         xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetShort – Set Short Integer Property Value

Interface:

```
xmsRC xmsPropertySetShort(xmsHProperty property,
                          xmsSHORT propertyValue,
                          xmsHErrorBlock errorBlock);
```

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.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsPropertySetString – Set String Property Value

Interface:

```
xmsRC xmsPropertySetString(xmsHProperty property,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsHErrorBlock errorBlock);
```

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 as a character array.

length (input)
The length of the property value in bytes. If the property value is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

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

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

PropertyContext

The PropertyContext class contains functions that get and set properties. These functions can operate on any object that can have properties.

All objects can have properties except ErrorBlock, Iterator, and Property objects.

Functions

Summary of functions:

Function	Description
xmsGetBooleanProperty	Get the value of the boolean property identified by name.
xmsGetByteArrayProperty	Get the value of the byte array property identified by name.
xmsGetByteArrayPropertyByRef	Get a pointer to the value of the byte array property identified by name.
xmsGetByteProperty	Get the value of the byte property identified by name.
xmsGetCharProperty	Get the value of the 2-byte character property identified by name.
xmsGetDoubleProperty	Get the value of the double precision floating point property identified by name.
xmsGetFloatProperty	Get the value of the floating point property identified by name.
xmsGetHandleTypeId	Get the type of the handle for the object.
xmsGetIntProperty	Get the value of the integer property identified by name.
xmsGetLongProperty	Get the value of the long integer property identified by name.
xmsGetObjectProperty	Get the value and data type of the property identified by name.
xmsGetProperty	Get a Property object for the property identified by name.
xmsGetShortProperty	Get the value of the short integer property identified by name.
xmsGetStringProperty	Get the value of the string property identified by name.
xmsGetStringPropertyByRef	Get a pointer to the value of the string property identified by name.

Function	Description
xmsSetBooleanProperty	Set the value of the boolean property identified by name.
xmsSetByteProperty	Set the value of the byte property identified by name.
xmsSetByteArrayProperty	Set the value of the byte array property identified by name.
xmsSetCharProperty	Set the value of the 2-byte character property identified by name.
xmsSetDoubleProperty	Set the value of the double precision floating point property identified by name.
xmsSetFloatProperty	Set the value of the floating point property identified by name.
xmsSetIntProperty	Set the value of the integer property identified by name.
xmsSetLongProperty	Set the value of the long integer property identified by name.
xmsSetObjectProperty	Set the value and data type of a property identified by name.
xmsSetProperty	Set the value of a property using a Property object.
xmsSetShortProperty	Set the value of the short integer property identified by name.
xmsSetStringProperty	Set the value of the string property identified by name.

xmsGetBooleanProperty – Get Boolean Property

Interface:

```
xmsRC xmsGetBooleanProperty(xmsHObj object,
                            xmsCHAR *propertyName,
                            xmsBOOL *propertyValue,
                            xmsHErrorBlock errorBlock);
```

Get the value of the boolean property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetByteArrayProperty – Get Byte Array Property

Interface:

```
xmsRC xmsGetByteArrayProperty(xmsHObj object
                              xmsCHAR *propertyName,
                              xmsSBYTE *propertyValue,
```

```
                                    xmsINT length,
                                    xmsINT *actualLength
                                    xmsHErrorBlock errorBlock) const;
```

Get the value of the byte array property identified by name.

For more information about how to use this function, see "C functions that return a byte array by value" on page 59.

Parameters:

object (input)
The handle for the object.

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, which is an array of bytes.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length of the array is not returned.

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

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetByteArrayPropertyByRef – Get Byte Array Property by Reference

Interface:

```
xmsRC xmsGetByteArrayPropertyByRef(xmsHObj object
                                   xmsCHAR *propertyName,
                                   xmsSBYTE **propertyValue,
                                   xmsINT *length,
                                   xmsHErrorBlock errorBlock) const;
```

Get a pointer to the value of the byte array 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 60.

Parameters:

object (input)
The handle for the object.

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, which is an array of bytes.

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

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

Thread context:
Determined by the thread context of the object

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsGetByteProperty – Get Byte Property

Interface:

```
xmsRC xmsGetByteProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsSBYTE *propertyValue,
                         xmsHErrorBlock errorBlock);
```

Get the value of the byte property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsGetCharProperty – Get Character Property

Interface:

```
xmsRC xmsGetCharProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsCHAR16 *propertyValue,
                         xmsHErrorBlock errorBlock);
```

Get the value of the 2-byte character property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsGetDoubleProperty – Get Double Precision Floating Point Property

Interface:

```
xmsRC xmsGetDoubleProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsDOUBLE *propertyValue,
                           xmsHErrorBlock errorBlock);
```

Get the value of the double precision floating point property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsGetFloatProperty – Get Floating Point Property

Interface:

```
xmsRC xmsGetFloatProperty(xmsHObj object,
                          xmsCHAR *propertyName,
                          xmsFLOAT *propertyValue,
                          xmsHErrorBlock errorBlock);
```

Get the value of the floating point property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
- XMS_X_GENERAL_EXCEPTION

xmsGetHandleTypeId – Get Handle Type

Interface:

```
xmsRC xmsGetHandleTypeId(xmsHObj object,
                         xmsHANDLE_TYPE *handleType,
                         xmsHErrorBlock errorBlock);
```

Get the type of the handle for the object.

Parameters:

object (input)
The handle for the object.

handleType (output)
The type of the handle for the object, which is one of the following values:

XMS_HANDLE_TYPE_CONN
XMS_HANDLE_TYPE_CONNFACT
XMS_HANDLE_TYPE_CONNMETADATA
XMS_HANDLE_TYPE_DEST
XMS_HANDLE_TYPE_ERRORBLOCK
XMS_HANDLE_TYPE_INITIALCONTEXT
XMS_HANDLE_TYPE_ITERATOR
XMS_HANDLE_TYPE_MSG
XMS_HANDLE_TYPE_MSGCONSUMER
XMS_HANDLE_TYPE_MSGPRODUCER
XMS_HANDLE_TYPE_QUEUEBROWSER
XMS_HANDLE_TYPE_PROPERTY
XMS_HANDLE_TYPE_REQUESTOR
XMS_HANDLE_TYPE_SESS

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

Thread context:
Determined by the thread context of the object

Exceptions:
- XMS_X_GENERAL_EXCEPTION

xmsGetIntProperty – Get Integer Property

Interface:

```
xmsRC xmsGetIntProperty(xmsHObj object,
                        xmsCHAR *propertyName,
                        xmsINT *propertyValue,
                        xmsHErrorBlock errorBlock);
```

Get the value of the integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetLongProperty – Get Long Integer Property

Interface:

```
xmsRC xmsGetLongProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsLONG *propertyValue,
                         xmsHErrorBlock errorBlock);
```

Get the value of the long integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetObjectProperty – Get Object Property

Interface:

```
xmsRC xmsGetObjectProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsSBYTE *propertyValue,
                           xmsINT length,
                           xmsINT *actualLength,
                           xmsOBJECT_TYPE *objectType,
                           xmsHErrorBlock errorBlock);
```

Get the value and data type of the property identified by name.

For more information about how to use this function, see "C functions that return a byte array by value" on page 59.

Parameters:

object (input)
The handle for the object.

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, which is returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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 the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

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_BYTEARRAY
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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetProperty – Get Property

Interface:

```
xmsRC xmsGetProperty(xmsHObj object,
                     xmsCHAR *propertyName,
                     xmsHProperty *property,
                     xmsHErrorBlock errorBlock);
```

Get a Property object for the property identified by name.

Parameters:

object (input)
The handle for the object.

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

property (output)
The handle for the Property object.

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

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetShortProperty – Get Short Integer Property

Interface:

```
xmsRC xmsGetShortProperty(xmsHObj object,
                          xmsCHAR *propertyName,
                          xmsLONG *propertyValue,
                          xmsHErrorBlock errorBlock);
```

Get the value of the short integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsGetStringProperty – Get String Property

Interface:

```
xmsRC xmsGetStringProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsINT *actualLength,
                           xmsHErrorBlock errorBlock);
```

Get 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 by value" on page 58.

Parameters:

object (input)
The handle for the object.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

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

Thread context:
Determined by the thread context of the object

Exceptions:
v XMS_X_GENERAL_EXCEPTION

xmsGetStringPropertyByRef – Get String Property by Reference

Interface:

```
xmsRC xmsMsgGetStringPropertyByRef(xmsHObj object,
                                   xmsCHAR *propertyName,
                                   xmsCHAR **propertyValue,
                                   xmsINT *length,
                                   xmsHErrorBlock errorBlock);
```

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 60.

Parameters:

object (input)
The handle for the object.

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.

Note that the value of the property must be a string. The function makes no attempt to convert a value with another data type into a string. If an application calls this function to get a pointer to a value that is not a string, XMS returns an error.

length (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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSetBooleanProperty – Set Boolean Property

Interface:

```
xmsRC xmsSetBooleanProperty(xmsHObj object,
                            xmsCHAR *propertyName,
                            xmsBOOL propertyValue,
                            xmsHErrorBlock errorBlock);
```

Set the value of the boolean property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetByteProperty – Set Byte Property

Interface:

```
xmsRC xmsSetByteProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsSBYTE propertyValue,
                         xmsHErrorBlock errorBlock);
```

Set the value of the byte property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

v XMS_X_GENERAL_EXCEPTION

v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetByteArrayProperty – Set Byte Array Property

Interface:

```
xmsRC xmsSetByteArrayProperty(xmsHObj object,
                              xmsCHAR *propertyName,
                              xmsSBYTE *propertyValue,
                              xmsINT length
                              xmsHErrorBlock errorBlock);
```

Set the value of the byte array property identified by name.

Parameters:

object (input)
The handle for the object.

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

propertyValue (input)
The value of the property, which is an array of bytes.

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

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

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetCharProperty – Set Character Property

Interface:

```
xmsRC xmsSetCharProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsCHAR16 propertyValue,
                         xmsHErrorBlock errorBlock);
```

Set the value of the 2-byte character property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetDoubleProperty – Set Double Precision Floating Point Property

Interface:

```
xmsRC xmsSetDoubleProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsDOUBLE propertyValue,
                           xmsHErrorBlock errorBlock);
```

Set the value of the double precision floating point property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetFloatProperty – Set Floating Point Property

Interface:

```
xmsRC xmsSetFloatProperty(xmsHObj object,
                          xmsCHAR *propertyName,
                          xmsFLOAT propertyValue,
                          xmsHErrorBlock errorBlock);
```

Set the value of the floating point property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetIntProperty – Set Integer Property

Interface:

```
xmsRC xmsSetIntProperty(xmsHObj object,
                        xmsCHAR *propertyName,
                        xmsINT propertyValue,
                        xmsHErrorBlock errorBlock);
```

Set the value of the integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetLongProperty – Set Long Integer Property

Interface:

```
xmsRC xmsSetLongProperty(xmsHObj object,
                         xmsCHAR *propertyName,
                         xmsLONG propertyValue,
                         xmsHErrorBlock errorBlock);
```

Set the value of the long integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetObjectProperty – Set Object Property

Interface:

```
xmsRC xmsSetObjectProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsSBYTE *propertyValue,
                           xmsINT length,
                           xmsOBJECT_TYPE objectType,
                           xmsHErrorBlock errorBlock);
```

Set the value and data type of a property identified by name.

Parameters:

object (input)
The handle for the object.

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

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_BYTEARRAY
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.

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetProperty – Set Property

Interface:

```
xmsRC xmsSetProperty(xmsHObj object,
                     xmsHProperty property,
                     xmsHErrorBlock errorBlock);
```

Set the value of a property using a Property object.

Parameters:

object (input)
The handle for the object.

property (input)
The handle for the Property object.

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

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetShortProperty – Set Short Integer Property

Interface:

```
xmsRC xmsSetShortProperty(xmsHObj object,
                          xmsCHAR *propertyName,
                          xmsSHORT propertyValue,
                          xmsHErrorBlock errorBlock);
```

Set the value of the short integer property identified by name.

Parameters:

object (input)
The handle for the object.

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.

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsSetStringProperty – Set String Property

Interface:

```
xmsRC xmsSetStringProperty(xmsHObj object,
                           xmsCHAR *propertyName,
                           xmsCHAR *propertyValue,
                           xmsINT length,
                           xmsHErrorBlock errorBlock);
```

Set the value of the string property identified by name.

Parameters:

object (input)
The handle for the object.

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. If the value of the

property is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

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

Thread context:
Determined by the thread context of the object

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

QueueBrowser

An application uses a queue browser to browse messages on a queue without removing them.

Functions

Summary of functions:

Function	Description
xmsQueueBrowserClose	Close the queue browser.
xmsQueueBrowserGetEnumeration	Get a list of the messages on the queue.
xmsQueueBrowserGetMessageSelector	Get the message selector for the queue browser.
xmsQueueBrowserGetQueue	Get the queue associated with the queue browser.

xmsQueueBrowserClose – Close Queue Browser

Interface:

```
xmsRC xmsQueueBrowserClose(xmsHQueueBrowser *browser,
                           xmsHErrorBlock errorBlock);
```

Close the queue browser.

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

Parameters:

browser (input/output)
On input, the handle for the queue browser. On output, the function returns a null handle.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsQueueBrowserGetEnumeration – Get Messages

Interface:

```
xmsRC xmsQueueBrowserGetEnumeration(xmsHQueueBrowser browser,
                                    xmsHIterator *iterator,
                                    xmsHErrorBlock errorBlock);
```

Get a list of the messages on the queue.

The function returns an iterator that encapsulates a list of Message objects. The order of the Message objects in the list is the same as the order in which the messages would be retrieved from the queue. The application can then use the iterator to browse each message in turn.

The iterator is updated dynamically as messages are put on the queue and removed from the queue. Each time the application calls xmsIteratorGetNext() to browse the next message on the queue, the message returned reflects the current contents of the queue.

If an application calls this function more than once for a given queue browser, each call returns a new iterator. The application can therefore use more than one iterator to browse the messages on a queue and maintain multiple positions within the queue.

Parameters:

browser (input)
The handle for the queue browser.

iterator (output)
The handle for the iterator.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsQueueBrowserGetMessageSelector – Get Message Selector

Interface:

```
xmsRC xmsQueueBrowserGetMessageSelector(xmsHQueueBrowser browser,
                                        xmsCHAR *messageSelector,
                                        xmsINT length,
                                        xmsINT *actualLength,
                                        xmsHErrorBlock errorBlock);
```

Get the message selector for the queue browser.

For more information about how to use this function, see “C functions that return a string by value” on page 58.

Parameters:

browser (input)
The handle for the queue browser.

messageSelector (output)
The buffer to contain the message selector expression. If data conversion is required, this is the message selector expression after conversion.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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. If data conversion is required, this is the length of the message selector expression after conversion. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsQueueBrowserGetQueue – Get Queue

Interface:

```
xmsRC xmsQueueBrowserGetQueue(xmsHQueueBrowser browser,
                              xmsHDest *queue,
                              xmsHErrorBlock errorBlock);
```

Get the queue associated with the queue browser.

Parameters:

browser (input)
The handle for the queue browser.

queue (output)
The handle for a Destination object representing the queue.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Requestor

An application uses a requestor to send a request message and then wait for, and receive, the reply.

Functions

Summary of functions:

Function	Description
xmsRequestorClose	Close the requestor.
xmsRequestorCreate	Create a requestor.
xmsRequestorRequest	Send a request message and then wait for, and receive, a reply from the application that receives the request message.

xmsRequestorClose – Close Requestor

Interface:

```
xmsRC xmsRequestorClose(xmsHRequestor *requestor,
                        xmsHErrorBlock errorBlock);
```

Close the requestor.

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

Note: When an application closes a requestor, the associated session does not close as well. In this respect, XMS behaves differently compared to JMS.

Parameters:

requestor (input/output)
On input, the handle for the requestor. On output, the function returns a null handle.

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

Thread context:
Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsRequestorCreate – Create Requestor

Interface:

```
xmsRC xmsRequestorCreate(xmsHSess session,
                         xmsHDest destination,
                         xmsHRequestor *requestor
                         xmsHErrorBlock errorBlock);
```

Create a requestor.

Parameters:

session (input)
The handle for a session. The session must not be transacted and must have one of the following acknowledgement modes:

XMSC_AUTO_ACKNOWLEDGE

XMSC_DUPS_OK_ACKNOWLEDGE

destination (input)
The handle for the destination where the application can send request messages.

requestor (output)
The handle for the requestor.

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

Thread context:
The session associated with the requestor

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsRequestorRequest – Request

Interface:

```
xmsRC xmsRequestorRequest(xmsHRequestor requestor,
                          xmsHMsg requestMessage,
                          xmsHMsg *replyMessage,
                          xmsHErrorBlock errorBlock);
```

Send a request message and then wait for, and receive, a reply from the application that receives the request message.

A call to this function blocks until a reply is received or until the session ends, whichever is the sooner.

Parameters:

requestor (input)
The handle for the requestor.

requestMessage (input)
The handle for the request message.

replyMessage (output)
The handle for the reply message.

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

Thread context:
The session associated with the requestor

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Session

A session is a single threaded context for sending and receiving messages.

For a list of the XMS defined properties of a Session, see "Properties of Session" on page 414.

Functions

Summary of functions:

Function	Description
xmsSessClose	Close the session.
xmsSessCommit	Commit all messages processed in the current transaction.
xmsSessCreateBrowser	Create a queue browser for the specified queue.
xmsSessCreateBrowserSelector	Create a queue browser for the specified queue using a message selector.
xmsSessCreateBytesMessage	Create a bytes message.
xmsSessCreateConsumer	Create a message consumer for the specified destination.
xmsSessCreateConsumerSelector	Create a message consumer for the specified destination using a message selector.
xmsSessCreateConsumerSelectorLocal	Create a message consumer for the specified destination using a message selector and, if the destination is a topic, specifying whether the message consumer receives the messages published by its own connection.
xmsSessCreateDurableSubscriber	Create a durable subscriber for the specified topic.

Function	Description
xmsSessCreateDurableSubscriberSelector	Create a durable subscriber for the specified topic using a message selector and specifying whether the durable subscriber receives the messages published by its own connection.
xmsSessCreateMapMessage	Create a map message.
xmsSessCreateMessage	Create a message that has no body.
xmsSessCreateObjectMessage	Create an object message.
xmsSessCreateProducer	Create a message producer to send messages to the specified destination.
xmsSessCreateStreamMessage	Create a stream message.
xmsSessCreateTextMessage	Create a text message with an empty body.
xmsSessCreateTextMessageInit	Create a text message whose body is initialized with the specified text.
xmsSessGetAcknowledgeMode	Get the acknowledgement mode for the session.
xmsSessGetTransacted	Determine whether the session is transacted.
xmsSessRecover	Recover the session.
xmsSessRollback	Rollback all messages processed in the current transaction.
xmsSessUnsubscribe	Delete a durable subscription.

xmsSessClose – Close Session

Interface:

```
xmsRC xmsSessClose(xmsHSess *session,
                   xmsHErrorBlock errorBlock);
```

Close the session. If the session is transacted, any transaction in progress is rolled back.

All objects dependent on the session are deleted. For information about which objects are deleted, see "Object Deletion" on page 42.

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 function returns a null handle.

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

Thread context:
Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsSessCommit – Commit

Interface:

```
xmsRC xmsSessCommit(xmsHSess session,
                    xmsHErrorBlock errorBlock);
```

Commit all messages processed in the current transaction.

Parameters:

session (input)
The handle for the session. The session must be a transacted session.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION
- XMS_X_TRANSACTION_ROLLED_BACK_EXCEPTION

xmsSessCreateBrowser – Create Queue Browser

Interface:

```
xmsRC xmsSessCreateBrowser(xmsHSess session,
                           xmsHDest queue,
                           xmsHQueueBrowser *browser
                           xmsHErrorBlock errorBlock);
```

Create a queue browser for the specified queue.

Parameters:

session (input)
The handle for the session.

queue (input)
The handle for a Destination object representing the queue.

browser (output)
The handle for the queue browser.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsSessCreateBrowserSelector – Create Queue Browser (with message selector)

Interface:

```
xmsRC xmsSessCreateBrowserSelector(xmsHSess session,
                                   xmsHDest queue,
                                   xmsCHAR *messageSelector,
                                   xmsINT length,
                                   xmsHQueueBrowser *browser
                                   xmsHErrorBlock errorBlock);
```

Create a queue browser for the specified queue using a message selector.

Parameters:

session (input)
The handle for the session.

queue (input)
The handle for a Destination object representing the queue.

messageSelector (input)
A message selector expression as a character array. Only those messages with properties that match the message selector expression are delivered to the queue browser.

A value of null or an empty string means that there is no message selector for the queue browser.

length (input)
The length of the message selector expression in bytes. If the message selector expression is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

browser (output)
The handle for the queue browser.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

xmsSessCreateBytesMessage – Create Bytes Message

Interface:

```
xmsRC xmsSessCreateBytesMessage(xmsHSess session,
                                xmsHMsg *message,
                                xmsHErrorBlock errorBlock);
```

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsSessCreateConsumer – Create Consumer

Interface:

```
xmsRC xmsSessCreateConsumer(xmsHSess session,
                            xmsHDest destination,
                            xmsHMsgConsumer *consumer,
                            xmsHErrorBlock errorBlock);
```

Create a message consumer for the specified destination.

Parameters:

session (input)
The handle for the session.

destination (input)
The handle for the destination.

consumer (output)
The handle for the message consumer.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsSessCreateConsumerSelector – Create Consumer (with message selector)

Interface:

```
xmsRC xmsSessCreateConsumerSelector(xmsHSess session,
                                    xmsHDest destination,
                                    xmsCHAR *messageSelector,
                                    xmsINT length,
                                    xmsHMsgConsumer *consumer,
                                    xmsHErrorBlock errorBlock);
```

Create a message consumer for the specified destination using a message selector.

Parameters:

session (input)
The handle for the session.

destination (input)
The handle for the destination.

messageSelector (input)
A message selector expression as a character array. Only those messages with properties that match the message selector expression are delivered to the message consumer.

A value of null or an empty string means that there is no message selector for the message consumer.

length (input)
The length of the message selector expression in bytes. If the message selector expression is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

consumer (output)
The handle for the message consumer.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

xmsSessCreateConsumerSelectorLocal – Create Consumer (with message selector and local message flag)

Interface:

```
xmsRC xmsSessCreateConsumerSelectorLocal(xmsHSess session,
                                         xmsHDest destination,
                                         xmsCHAR *messageSelector,
                                         xmsINT length,
                                         xmsBOOL noLocal,
                                         xmsHMsgConsumer *consumer,
                                         xmsHErrorBlock errorBlock);
```

Create a message consumer for the specified destination using a message selector and, if the destination is a topic, specifying whether the message consumer receives the messages published by its own connection.

Parameters:

session (input)
The handle for the session.

destination (input)
The handle for the destination.

messageSelector (input)
A message selector expression as a character array. Only those messages with properties that match the message selector expression are delivered to the message consumer.

A value of null or an empty string means that there is no message selector for the message consumer.

length (input)
The length of the message selector expression in bytes. If the message selector expression is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

noLocal (input)
The value `xmsTRUE` means that the message consumer does not receive the messages published by its own connection. The value `xmsFALSE` means that the message consumer does receive the messages published by its own connection. The default value is `xmsFALSE`.

consumer (output)
The handle for the message consumer.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

xmsSessCreateDurableSubscriber – Create Durable Subscriber

Interface:

```
xmsRC xmsSessCreateDurableSubscriber(xmsHSess session,
                                     xmsHDest topic,
                                     xmsCHAR *subscriptionName
                                     xmsHMsgConsumer *subscriber,
                                     xmsHErrorBlock errorBlock);
```

Create a durable subscriber for the specified topic.

For more information about durable subscribers, see "Durable subscribers" on page 37.

Parameters:

session (input)
The handle for the session.

topic (input)
The handle for a Destination object representing the topic. The topic must not be a temporary topic.

subscriptionName (input)
A name that identifies the durable subscription. The name must be unique within the client identifier for the connection, and is in the format of a null terminated string.

subscriber (output)
The handle for the MessageConsumer object representing the durable subscriber.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsSessCreateDurableSubscriberSelector – Create Durable Subscriber (with message selector and local message flag)

Interface:

```
xmsRC xmsSessCreateDurableSubscriberSelector(xmsHSess session,
                                             xmsHDest topic,
                                             xmsCHAR *subscriptionName
                                             xmsCHAR *messageSelector,
                                             xmsINT length,
                                             xmsBOOL noLocal,
                                             xmsHMsgConsumer *subscriber,
                                             xmsHErrorBlock errorBlock);
```

Create a durable subscriber for the specified topic using a message selector and specifying whether the durable subscriber receives the messages published by its own connection.

For more information about durable subscribers, see “Durable subscribers” on page 37.

Parameters:

session (input)
The handle for the session.

topic (input)
The handle for a Destination object representing the topic. The topic must not be a temporary topic.

subscriptionName (input)
A name that identifies the durable subscription. The name must be unique within the client identifier for the connection, and is in the format of a null terminated string.

messageSelector (input)
A message selector expression as a character array. Only those messages with properties that match the message selector expression are delivered to the durable subscriber.

A value of null or an empty string means that there is no message selector for the durable subscriber.

length (input)
The length of the message selector expression in bytes. If the message selector expression is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

noLocal (input)
The value `xmsTRUE` means that the durable subscriber does not receive the messages published by its own connection. The value `xmsFALSE` means that the durable subscriber does receive the messages published by its own connection. The default value is `xmsFALSE`.

subscriber (output)
The handle for the MessageConsumer object representing the durable subscriber.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

xmsSessCreateMapMessage – Create Map Message

Interface:

```
xmsRC xmsSessCreateMapMessage(xmsHSess session,
                              xmsHMsg *message,
                              xmsHErrorBlock errorBlock);
```

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessCreateMessage – Create Message

Interface:

```
xmsRC xmsSessCreateMessage(xmsHSess session,
                           xmsHMsg *message,
                           xmsHErrorBlock errorBlock);
```

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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessCreateObjectMessage – Create Object Message

Interface:

```
xmsRC xmsSessCreateObjectMessage(xmsHSess session,
                                 xmsHMsg *message,
                                 xmsHErrorBlock errorBlock);
```

Create an object message.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the object message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessCreateProducer – Create Producer

Interface:

```
xmsRC xmsSessCreateProducer(xmsHSess session,
                            xmsHDest destination,
                            xmsHMsgProducer *producer,
                            xmsHErrorBlock errorBlock);
```

Create a message producer to send messages to the specified destination.

Parameters:

session (input)
The handle for the session.

destination (input)
The handle for the destination.

If you specify a null handle, the message producer is created without a destination. In this case, the application must specify a destination every time it uses the message producer to send a message.

producer (output)
The handle for the message producer.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

xmsSessCreateStreamMessage – Create Stream Message

Interface:

```
xmsRC xmsSessCreateStreamMessage(xmsHSess session,
                                 xmsHMsg *message,
                                 xmsHErrorBlock errorBlock);
```

Create a stream message.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the stream message.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsSessCreateTextMessage – Create Text Message

Interface:

```
xmsRC xmsSessCreateTextMessage(xmsHSess session,
                               xmsHMsg *message,
                               xmsHErrorBlock errorBlock);
```

Create a text message with an empty body.

Parameters:

session (input)
The handle for the session.

message (output)
The handle for the text message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessCreateTextMessageInit – Create Text Message (initialized)

Interface:

```
xmsRC xmsSessCreateTextMessageInit(xmsHSess session,
                                   xmsCHAR *text
                                   xmsINT length
                                   xmsHMsg *message,
                                   xmsHErrorBlock errorBlock);
```

Create a text message whose body is initialized with the specified text.

Parameters:

session (input)
The handle for the session.

text (input)
A character array containing the text to initialize the body of the text message.

length (input)
The length of the text in bytes. If the text is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

message (output)
The handle for the text message.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessGetAcknowledgeMode – Get Acknowledgement Mode

Interface:

```
xmsRC xmsSessGetAcknowledgeMode(xmsHSess session,
                                xmsINT *acknowledgeMode,
                                xmsHErrorBlock errorBlock);
```

Get the acknowledgement mode for the session. The acknowledgement mode is specified when the session is created.

A session that is transacted has no acknowledgement mode.

For more information about acknowledgement modes, see "Message acknowledgement" on page 29.

Parameters:

session (input)
The handle for the session.

acknowledgeMode (output)
The acknowledgement mode. Provided the session is not transacted, the acknowledgement mode is one of the following values:

`XMSC_AUTO_ACKNOWLEDGE`

`XMSC_CLIENT_ACKNOWLEDGE`

`XMSC_DUPS_OK_ACKNOWLEDGE`

If the session is transacted, the function returns `XMSC_SESSION_TRANSACTED` instead.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSessGetTransacted – Determine Whether Transacted

Interface:

```
xmsRC xmsSessGetTransacted(xmsHSess session,
                           xmsBOOL *transacted,
                           xmsHErrorBlock errorBlock);
```

Determine whether the session is transacted.

Parameters:

session (input)
The handle for the session.

transacted (output)
The value is `xmsTRUE` if the session is transacted. The value is `xmsFALSE` if the session is not transacted.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsSessRecover – Recover

Interface:

```
xmsRC xmsSessRecover(xmsHSess session,
                     xmsHErrorBlock errorBlock);
```

Recover the session. Message delivery is stopped and then restarted with the oldest unacknowledged message.

The session must not be a transacted session.

For more information about recovering a session, see "Message acknowledgement" on page 29.

Parameters:

session (input)
The handle for the session.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

xmsSessRollback – Rollback

Interface:

```
xmsRC xmsSessRollback(xmsHSess session,
                      xmsHErrorBlock errorBlock);
```

Rollback all messages processed in the current transaction.

Parameters:

session (input)
The handle for the session. The session must be a transacted session.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

xmsSessUnsubscribe – Unsubscribe

Interface:

```
xmsRC xmsSessUnsubscribe(xmsHSess session,
                         xmsCHAR *subscriptionName,
                         xmsHErrorBlock errorBlock);
```

Delete a durable subscription. The messaging server deletes the record of the durable subscription that it is maintaining and does not send any more messages to the durable subscriber.

An application cannot delete a durable subscription in any of the following circumstances:

- While there is an active message consumer for the durable subscription
- While a consumed message is part of a pending transaction
- While a consumed message has not been acknowledged

Parameters:

session (input)
The handle for the session.

subscriptionName (input)
The name that identifies the durable subscription. The name is in the format of a null terminated string.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

StreamMessage

A stream message is a message whose body comprises a stream of values, where each value has an associated data type.

The contents of the body are written to and read sequentially.

When an application reads a value from the message stream, the value can be converted by XMS into another data type. For more information about this form of implicit conversion, see "Stream messages" on page 99.

Functions

Summary of functions:

Function	Description
xmsStreamMsgReadBoolean	Read a boolean value from the message stream.
xmsStreamMsgReadByte	Read a signed 8-bit integer from the message stream.
xmsStreamMsgReadBytes	Read an array of bytes from the message stream.
xmsStreamMsgReadBytesByRef	Get a pointer to an array of bytes in the message stream, and get the length of the array.
xmsStreamMsgReadChar	Read a 2-byte character from the message stream.
xmsStreamMsgReadDouble	Read an 8-byte double precision floating point number from the message stream.

Function	Description
xmsStreamMsgReadFloat	Read a 4-byte floating point number from the message stream.
xmsStreamMsgReadInt	Read a signed 32-bit integer from the message stream.
xmsStreamMsgReadLong	Read a signed 64-bit integer from the message stream.
xmsStreamMsgReadObject	Read a value from the message stream, and return its data type.
xmsStreamMsgReadShort	Read a signed 16-bit integer from the message stream.
xmsStreamMsgReadString	Read a string from the message stream.
xmsStreamMsgReset	Put the body of the message into read-only mode and reposition the cursor at the beginning of the message stream.
xmsStreamMsgWriteBoolean	Write a boolean value to the message stream.
xmsStreamMsgWriteByte	Write a byte to the message stream.
xmsStreamMsgWriteBytes	Write an array of bytes to the message stream.
xmsStreamMsgWriteChar	Write a character to the message stream as 2 bytes, high order byte first.
xmsStreamMsgWriteDouble	Convert a double precision floating point number to a long integer and write the long integer to the message stream as 8 bytes, high order byte first.
xmsStreamMsgWriteFloat	Convert a floating point number to an integer and write the integer to the message stream as 4 bytes, high order byte first.
xmsStreamMsgWriteInt	Write an integer to the message stream as 4 bytes, high order byte first.
xmsStreamMsgWriteLong	Write a long integer to the message stream as 8 bytes, high order byte first.
xmsStreamMsgWriteObject	Write a value, with a specified data type, to the message stream.
xmsStreamMsgWriteShort	Write a short integer to the message stream as 2 bytes, high order byte first.
xmsStreamMsgWriteString	Write a string to the message stream.

xmsStreamMsgReadBoolean – Read Boolean Value

Interface:

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

Read a boolean value from the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadByte – Read Byte

Interface:

```
xmsRC xmsStreamMsgReadByte(xmsHMsg message,
                           xmsSBYTE *value,
                           xmsHErrorBlock errorBlock);
```

Read a signed 8-bit integer from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadBytes – Read Bytes

Interface:

```
xmsRC xmsStreamMsgReadBytes(xmsHMsg message,
                            xmsSBYTE *buffer,
                            xmsINT bufferLength,
                            xmsINT *returnedLength,
                            xmsHErrorBlock errorBlock);
```

Read an array of bytes from the message stream.

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 in the array is less than or equal to the length of the buffer, the whole array is read into the buffer. If the number of bytes in the array is greater than the length of the buffer, the buffer is filled with part of the array, and an internal cursor marks the position of the next byte to be read. A subsequent call to xmsStreamMsgReadBytes() reads bytes from the array starting from the current position of the cursor.

If you specify a null pointer on input, the call skips over the array of bytes without reading it.

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 in the array remaining to be read. If there are no bytes remaining to be read from the array before the call, the value is XMSC_END_OF_BYTEARRAY.

If you specify a null pointer on input, the function returns no value.

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

xmsStreamMsgReadBytesByRef – Read Bytes by Reference

Interface:

```
xmsRC xmsStreamMsgReadBytesByRef(xmsHMsg message,
                                 xmsSBYTE **array,
                                 xmsINT *length,
                                 xmsHErrorBlock errorBlock);
```

Get a pointer to an array of bytes in the message stream, and get the length of the array.

For more information about how to use this function, see "C functions that return a string or byte array by reference" on page 60.

Parameters:

message (input)
The handle for the message.

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
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadChar – Read Character

Interface:

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

Read a 2-byte character from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadDouble – Read Double Precision Floating Point Number

Interface:

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

Read an 8-byte double precision floating point number from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadFloat – Read Floating Point Number

Interface:

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

Read a 4-byte floating point number from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadInt – Read Integer

Interface:

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

Read a signed 32-bit integer from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadLong – Read Long Integer

Interface:

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

Read a signed 64-bit integer from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadObject – Read Object

Interface:

```
xmsRC xmsstreamMsgReadObject(xmsHMsg message,
                             xmsSBYTE *buffer,
                             xmsINT bufferLength,
                             xmsINT *actualLength,
                             xmsOBJECT_TYPE *objectType,
                             xmsHErrorBlock errorBlock);
```

Read a value from the message stream, and return its data type.

For more information about how to use this function, see "C functions that return a byte array by value" on page 59.

Parameters:

message (input)
The handle for the message.

buffer (output)
The buffer to contain the value, which is returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

If you specify a null pointer on input, the call skips over the value without reading it.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value in bytes. If the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

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_BYTEARRAY
- 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

xmsStreamMsgReadShort – Read Short Integer

Interface:

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

Read a signed 16-bit integer from the message stream.

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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

xmsStreamMsgReadString – Read String

Interface:

```
xmsRC xmsStreamMsgReadString(xmsHMsg message,
                             xmsCHAR *buffer,
                             xmsINT bufferLength,
                             xmsINT *actualLength,
                             xmsHErrorBlock errorBlock);
```

Read a string from the message stream. If required, XMS converts the characters in the string into the local code page.

For more information about how to use this function, see “C functions that return a string by value” on page 58.

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 XMSC_QUERY_SIZE, the string is not returned, but its length is returned in the actualLength parameter, and the cursor is not advanced.

If you specify XMSC_SKIP, the function skips over the string without reading it.

actualLength (output)
The length of the string in bytes. If data conversion is required, this is the length of the string after conversion. If you specify a null pointer on input, the length is not returned.

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

Notes:

1. 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 an error. XMS does not advance the internal cursor.
2. 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.

xmsStreamMsgReset – Reset

Interface:

```
xmsRC xmsStreamMsgReset(xmsHMsg message,
                        xmsHErrorBlock errorBlock);
```

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

Parameters:

message (input)
The handle for the message.

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

xmsStreamMsgWriteBoolean – Write Boolean Value

Interface:

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

Write a boolean value to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteByte – Write Byte

Interface:

```
xmsRC xmsStreamMsgWriteByte(xmsHMsg message,
                            xmsSBYTE value,
                            xmsHErrorBlock errorBlock);
```

Write a byte to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteBytes – Write Bytes

Interface:

```
xmsRC xmsStreamMsgWriteBytes(xmsHMsg message,
                             xmsSBYTE *value,
                             xmsINT length,
                             xmsHErrorBlock errorBlock);
```

Write an array of bytes to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteChar – Write Character

Interface:

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

Write a character to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteDouble – Write Double Precision Floating Point Number

Interface:

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

Convert a double precision floating point number to a long integer and write the long integer to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteFloat – Write Floating Point Number

Interface:

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

Convert a floating point number to an integer and write the integer to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteInt – Write Integer

Interface:

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

Write an integer to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteLong – Write Long Integer

Interface:

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

Write a long integer to the 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.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteObject – Write Object

Interface:

```
xmsRC xmsStreamMsgWriteObject(xmsHMsg message,
                              xmsSBYTE *value,
                              xmsINT length,
                              xmsOBJECT_TYPE objectType,
                              xmsHErrorBlock errorBlock);
```

Write a value, with a specified data type, to the message stream.

Parameters:

message (input)
The handle for the message.

value (input)
An array of bytes containing the value to be written.

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_BYTEARRAY
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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsStreamMsgWriteShort – Write Short Integer

Interface:

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

Write a short integer to the 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.

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

xmsStreamMsgWriteString – Write String

Interface:

```
xmsRC xmsStreamMsgWriteString(xmsHMsg message,
                              xmsCHAR *value,
                              xmsINT length,
                              xmsHErrorBlock errorBlock);
```

Write a string to the message stream.

Parameters:

message (input)
The handle for the message.

value (input)
A character array containing the string to be written.

length (input)
The length of the string in bytes. If the string is null terminated with no embedded null characters, you can specify XMSC_CALCULATE_STRING_SIZE instead and allow XMS to calculate its length.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

TextMessage

A text message is a message whose body comprises a string.

Functions

Summary of functions:

Function	Description
xmsTextMsgGetText	Get the string that forms the body of the text message.
xmsTextMsgSetText	Set the string that forms the body of the text message.

xmsTextMsgGetText – Get Text

Interface:

```
xmsRC xmsTextMsgGetText(xmsHMsg message,
                        xmsCHAR *buffer,
                        xmsINT bufferLength,
                        xmsINT *actualLength,
                        xmsHErrorBlock errorBlock);
```

Get the string that forms the body of the text message. If required, XMS converts the characters in the string into the local code page.

For more information about how to use this function, see "C functions that return a string by value" on page 58.

Parameters:

message (input)
The handle for the message.

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 `XMSC_QUERY_SIZE` instead, 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. If you specify a null pointer on input, the length is not returned.

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
v XMS_X_MESSAGE_EOF_EXCEPTION

Notes:

1. 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 an error.
2. If any other error occurs while attempting to get the string, XMS reports the error but does not set the actualLength parameter.

xmsTextMsgSetText – Set Text

Interface:

```
xmsRC xmsTextMsgSetText(xmsHMsg message,
                        xmsCHAR *value,
                        xmsINT length,
                        xmsHErrorBlock errorBlock);
```

Set the string that forms the body of the text message.

Parameters:

message (input)
The handle for the message.

value (input)
A character array containing the string to be set.

length (input)
The length of the string in bytes. If the string is null terminated with no embedded null characters, you can specify `XMSC_CALCULATE_STRING_SIZE` instead and allow XMS to calculate its length.

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

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Chapter 13. Additional C functions

This topic documents the C functions that do not belong to any class.

The topic contains the following subtopic:

v "Process CCSID functions"

Process CCSID functions

This topic documents the functions for getting and setting the process coded character set identifier (CCSID) for an application.

For information about how to use these functions, see "Coded character set identifiers" on page 46.

Functions

Summary of functions:

Function	Description
xmsGetClientCCSID	Get the process CCSID for the application.
xmsSetClientCCSID	Set the process CCSID for the application.

xmsGetClientCCSID – Get Process CCSID

Interface:

```
xmsRC xmsGetClientCCSID(xmsINT *ccsid,
                        xmsHErrorBlock errorBlock);
```

Get the process CCSID for the application.

Parameters:

ccsid (output)
The process CCSID.

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

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

xmsSetClientCCSID – Set Process CCSID

Interface:

```
xmsRC xmsSetClientCCSID(xmsINT ccsid,
                        xmsHErrorBlock errorBlock);
```

Set the process CCSID for the application.

Parameters:

ccsid (input)
The process CCSID.

The following named constants are defined for the specified Unicode CCSIDs:

Named constant	CCSID
XMSC_CCSID_UTF8	The UTF-8 representation of Unicode data
XMSC_CCSID_UTF16	The UTF-16 representation of Unicode data
XMSC_CCSID_UTF32	The UTF-32 representation of Unicode data

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

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

Chapter 14. C++ classes

This topic documents the C++ classes and their methods.

The following table summarizes all the classes.

Table 34. Summary of the C++ classes

Class	Description
"BytesMessage" on page 265	A bytes message is a message whose body comprises a stream of bytes.
"Connection" on page 275	A Connection object represents an application's active connection.
"ConnectionFactory for the C++ class" on page 280	An application uses a connection factory to create a connection.
"ConnectionMetaData" on page 283	A ConnectionMetaData object provides information about a connection.
"Destination for the C++ class" on page 285	A destination is where an application sends messages, or it is a source from which an application receives messages, or both.
"Exception" on page 288	If XMS detects an error while processing a call to a 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.
"ExceptionListener" on page 292	An application uses an exception listener to be notified asynchronously of a problem with a connection.
"IllegalStateException" on page 293	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.
"InitialContext" on page 293	An application uses an InitialContext object to create objects from object definitions that are retrieved from a repository of administered objects.
"InvalidClientIDException" on page 296	XMS throws this exception if an application attempts to set a client identifier for a connection, but the client identifier is not valid or is already in use.
"InvalidDestinationException" on page 296	XMS throws this exception if an application specifies a destination that is not valid.
"InvalidSelectorException" on page 296	XMS throws this exception if an application provides a message selector expression whose syntax is not valid.
"Iterator" on page 297	An iterator encapsulates a list of objects. An application uses an iterator to access object in turn.
"MapMessage" on page 299	A map message is a message whose body comprises a set of name-value pairs, where each value has an associated data type.

Table 34. Summary of the C++ classes (continued)

Class	Description
"Message" on page 311	A Message object represents a message that an application sends or receives.
"MessageConsumer" on page 323	An application uses a message consumer to receive messages sent to a destination.
"MessageEOFException" on page 327	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.
"MessageFormatException" on page 328	XMS throws this exception if XMS encounters a message with a format that is not valid.
"MessageListener" on page 328	An application uses a message listener to receive messages asynchronously.
"MessageNotReadableException" on page 329	XMS throws this exception if an application attempts to read the body of a message that is write-only.
"MessageNotWritableException" on page 329	XMS throws this exception if an application attempts to write to the body of a message that is read-only.
"MessageProducer" on page 329	An application uses a message producer to send messages to a destination.
"ObjectMessage" on page 338	An object message is a message whose body comprises a serialized Java object.
"Property" on page 340	A Property object represents a property of an object.
"PropertyContext" on page 353	PropertyContext is an abstract superclass that contains methods that get and set properties. These methods are inherited by other classes.
"QueueBrowser" on page 365	An application uses a queue browser to browse messages on a queue without removing them.
"Requestor" on page 368	An application uses a requestor to send a request message and then wait for, and receive, the reply.
"ResourceAllocationException" on page 370	XMS throws this exception if XMS cannot allocate the resources required by a method.
"SecurityException" on page 371	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.
"Session" on page 371	A session is a single threaded context for sending and receiving messages.
"StreamMessage" on page 384	A stream message is a message whose body comprises a stream of values, where each value has an associated data type.
"String" on page 394	A String object encapsulates a string.
"TextMessage" on page 398	A text message is a message whose body comprises a string.
"TransactionInProgressException" on page 400	XMS throws this exception if an application requests an operation that is not valid because a transaction is in progress.
"TransactionRolledBackException" on page 400	XMS throws this exception if an application calls Session.commit() to commit the current transaction, but the transaction is subsequently rolled back.

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

Table 35. Exception codes and their corresponding C++ exceptions

Exception code	Corresponding C++ exception
XMS_X_GENERAL_EXCEPTION	"Exception" on page 288
XMS_X_ILLEGAL_STATE_EXCEPTION	"IllegalStateException" on page 293
XMS_X_INVALID_CLIENTID_EXCEPTION	"InvalidClientIDException" on page 296
XMS_X_INVALID_DESTINATION_EXCEPTION	"InvalidDestinationException" on page 296
XMS_X_INVALID_SELECTOR_EXCEPTION	"InvalidSelectorException" on page 296
XMS_X_MESSAGE_EOF_EXCEPTION	"MessageEOFException" on page 327
XMS_X_MESSAGE_FORMAT_EXCEPTION	"MessageFormatException" on page 328
XMS_X_MESSAGE_NOT_READABLE_EXCEPTION	"MessageNotReadableException" on page 329
XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION	"MessageNotWritableException" on page 329
XMS_X_RESOURCE_ALLOCATION_EXCEPTION	"ResourceAllocationException" on page 370
XMS_X_SECURITY_EXCEPTION	"SecurityException" on page 371
XMS_X_TRANSACTION_IN_PROGRESS_EXCEPTION	"TransactionInProgressException" on page 400
XMS_X_TRANSACTION_ROLLED_BACK_EXCEPTION	"TransactionRolledBackException" on page 400

BytesMessage

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

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
           |
           +--- xms::BytesMessage
```

Methods

Summary of methods:

Method	Description
getBodyLength	Get the length of the body of the message when the body of the message is read-only.
readBoolean	Read a boolean value from the bytes message stream.
readByte	Read the next byte from the bytes message stream as a signed 8-bit integer.
readBytes	Read an array of bytes from the bytes message stream starting from the current position of the cursor.
readChar	Read the next 2 bytes from the bytes message stream as a character.
readDouble	Read the next 8 bytes from the bytes message stream as a double precision floating point number.
readFloat	Read the next 4 bytes from the bytes message stream as a floating point number.
readInt	Read the next 4 bytes from the bytes message stream as a signed 32-bit integer.

Method	Description
readLong	Read the next 8 bytes from the bytes message stream as a signed 64-bit integer.
readShort	Read the next 2 bytes from the bytes message stream as a signed 16-bit integer.
readUnsignedByte	Read the next byte from the bytes message stream as an unsigned 8-bit integer.
readUnsignedShort	Read the next 2 bytes from the bytes message stream as an unsigned 16-bit integer.
readUTF	Read a string, encoded in UTF-8, from the bytes message stream.
reset	Put the body of the message into read-only mode and reposition the cursor at the beginning of the bytes message stream.
writeBoolean	Write a boolean value to the bytes message stream.
writeByte	Write a byte to the bytes message stream.
writeBytes	Write an array of bytes to the bytes message stream.
writeChar	Write a character to the bytes message stream as 2 bytes, high order byte first.
writeDouble	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.
writeFloat	Convert a floating point number to an integer and write the integer to the bytes message stream as 4 bytes, high order byte first.
writeInt	Write an integer to the bytes message stream as 4 bytes, high order byte first.
writeLong	Write a long integer to the bytes message stream as 8 bytes, high order byte first.
writeShort	Write a short integer to the bytes message stream as 2 bytes, high order byte first.
writeUTF	Write a string, encoded in UTF-8, to the bytes message stream.

getBodyLength – Get Body Length

Interface:

```
xmsLONG getBodyLength() const;
```

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

Parameters:

None

Returns:

The length of the body of the message in bytes. The method returns the length of the whole body regardless of where the cursor for reading the message is currently positioned.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

readBoolean – Read Boolean Value

Interface:

```
xmsBOOL readBoolean() const;
```

Read a boolean value from the bytes message stream.

Parameters:

None

Returns:

The boolean value that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readByte – Read Byte

Interface:

```
xmsSBYTE readByte() const;
```

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

Parameters:

None

Returns:

The byte that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readBytes – Read Bytes

Interface:

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

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

Parameters:

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 method 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. If you specify XMSC_QUERY_SIZE instead, no bytes are read into the buffer, but the number of bytes remaining in the stream, starting from the current position of the cursor, is returned in the returnedLength parameter, and the cursor is not advanced.

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.

If you specify a null pointer on input, the method returns no value.

Returns:
See the description of the returnedLength parameter.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

readChar – Read Character

Interface:

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

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

Parameters:
None

Returns:
The character that is read.

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readDouble – Read Double Precision Floating Point Number

Interface:

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

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

Parameters:
None

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

readFloat – Read Floating Point Number

Interface:

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

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

Parameters:

None

Returns:

The floating point number that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readInt – Read Integer

Interface:

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

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

Parameters:

None

Returns:

The integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readLong – Read Long Integer

Interface:

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

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

Parameters:

None

Returns:

The long integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readShort – Read Short Integer

Interface:

```
xmsSHORT readShort() const;
```

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

Parameters:

None

Returns:

The short integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readUnsignedByte – Read Unsigned Byte

Interface:

```
xmsBYTE readUnsignedByte() const;
```

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

Parameters:

None

Returns:

The byte that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readUnsignedShort – Read Unsigned Short Integer

Interface:

```
xmsUSHORT readUnsignedShort() const;
```

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

Parameters:

None

Returns:

The unsigned short integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readUTF – Read UTF String

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.

Parameters:

None

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

reset – Reset

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:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION

writeBoolean – Write Boolean Value

Interface:

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

Write a boolean value to the bytes message stream.

Parameters:

value (input)

The boolean value to be written.

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeByte – Write Byte

Interface:

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

Write a byte to the bytes message stream.

Parameters:

value (input)
The byte to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeBytes – Write Bytes

Interface:

```
xmsVOID writeBytes(const xmsSBYTE *value,
                   const xmsINT length);
```

Write an array of bytes to the bytes message stream.

Parameters:

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

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeChar – Write Character

Interface:

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

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

Parameters:

value (input)
The character to be written.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeDouble – Write Double Precision Floating Point Number

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:

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

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeFloat – Write Floating Point Number

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:

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

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeInt – Write Integer

Interface:

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

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

Parameters:

value (input)
The integer to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeLong – Write Long Integer

Interface:

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

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

Parameters:

value (input)
The long integer to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeShort – Write Short Integer

Interface:

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

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

Parameters:

value (input)
The short integer to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeUTF – Write UTF String

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:

value (input)
A String object encapsulating the string to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods

The following methods are inherited from the Message class:

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

The following methods are inherited from the PropertyContext class:

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

Connection

A Connection object represents an application's active connection.

Inheritance hierarchy:

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

For a list of the XMS defined properties of a Connection object, see "Properties of Connection" on page 401.

Methods

Summary of methods:

Method	Description
close	Close the connection.
createSession	Create a session.
getClientID	Get the client identifier for the connection.
getExceptionListener	Get a pointer to the exception listener that is registered with the connection.
getHandle	Get the handle that a C application would use to access the connection.

Method	Description
getMetaData	Get the metadata for the connection.
isNull	Determine whether the Connection object is a null object.
setClientID	Set a client identifier for the connection.
setExceptionListener	Register an exception listener with the connection.
start	Start, or restart, the delivery of incoming messages for the connection.
stop	Stop the delivery of incoming messages for the connection.

close – Close Connection

Interface:

```
xmsVOID close();
```

Close the connection.

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

Parameters:

None

Returns:

Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createSession – Create Session

Interface:

```
Session createSession(const xmsBOOL transacted,
                      const xmsINT acknowledgeMode);
```

Create a session.

Parameters:

transacted (input)

The value xmsTRUE means that the session is transacted. The value xmsFALSE means that the session is not transacted.

acknowledgeMode (input)

Indicates how messages received by an application are acknowledged. The value must be one of the following acknowledgement modes:

XMSC_AUTO_ACKNOWLEDGE

XMSC_CLIENT_ACKNOWLEDGE

XMSC_DUPS_OK_ACKNOWLEDGE

This parameter is ignored if the session is transacted. For more information about acknowledgement modes, see "Message acknowledgement" on page 29.

Returns:

The Session object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getClientID – Get Client ID

Interface:

```
String getClientID() const;
```

Get the client identifier for the connection.

Parameters:

None

Returns:

A String object encapsulating the client identifier.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getExceptionListener – Get Exception Listener

Interface:

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

Get a pointer to the exception listener that is registered with the connection.

For more information about using exception listeners, see "Exception listeners in C++" on page 74.

Parameters:

None

Returns:

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

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

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

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

Parameters:

None

Returns:

The handle for the connection.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getMetaData – Get Metadata

Interface:

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

Get the metadata for the connection.

Parameters:

None

Returns:

The ConnectionMetaData object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Connection object is a null object.

Parameters:

None

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

setClientID – Set Client ID

Interface:

```
xmsVOID setClientID(const String & clientID);
```

Set a client identifier for the connection. A client identifier is used only to support durable subscriptions in the publish/subscribe domain, and is ignored in the point-to-point domain.

If an application calls this method to set a client identifier for a connection, the application must do so immediately after creating the connection, and before

performing any other operation on the connection. If the application tries to call the method after this point, the call throws exception XMS_X_ILLEGAL_STATE_EXCEPTION.

Parameters:

clientID (input)
A String object encapsulating the client identifier.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION
- XMS_X_INVALID_CLIENTID_EXCEPTION

setExceptionListener – Set Exception Listener

Interface:

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

Register an exception listener with the connection.

For more information about using exception listeners, see "Exception listeners in C++" on page 74.

Parameters:

lsr (input)
A pointer to the exception listener.

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

start – Start Connection

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:
None

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

stop – Stop Connection

Interface:

```
xmsVOID stop() const;
```

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

Parameters:

None

Returns:

Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

ConnectionFactory for the C++ class

An application uses a connection factory to create a connection.

Inheritance hierarchy:

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

For a list of the XMS defined properties of a ConnectionFactory object, see "Properties of ConnectionFactory" on page 402.

Constructors

Summary of constructors:

Constructor	Description
ConnectionFactory	Create a connection factory with the default properties.

ConnectionFactory – Create Connection Factory

Interface:

```
ConnectionFactory();
```

Create a connection factory with the default properties.

Parameters:

None

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
~ConnectionFactory	Delete the connection factory.
createConnection	Create a connection using the default user identity.
createConnection	Create a connection using a specified user identity.
getHandle	Get the handle that a C application would use to access the connection factory.
isNull	Determine whether the ConnectionFactory object is a null object.

~ConnectionFactory – Delete Connection Factory

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:

None

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createConnection – Create Connection (using the default user identity)

Interface:

```
Connection createConnection();
```

Create a connection using the default user identity.

The connection factory properties XMSC_USERID 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 application, provided the messaging server permits a connection without authentication. The properties are ignored if the application connects to a IBM MQ queue manager in bindings mode.

The connection is created in stopped mode. No messages are delivered until the application calls Connection.start().

Parameters:

None

Returns:

The Connection object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

- XMS_X_SECURITY_EXCEPTION

createConnection – Create Connection (using a specified user identity)

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_USERID and XMSC_PASSWORD, if they are set, are ignored. The user identifier and password are ignored if the application connects to a IBM MQ queue manager in bindings mode.

The connection is created in stopped mode. No messages are delivered until the application calls Connection.start().

Parameters:

userID (input)
A String object encapsulating the user identifier to be used to authenticate the application. If you specify a null String object, the connection factory property XMSC_USERID is used instead.

password (input)
A String object encapsulating the password to be used to authenticate the application. If you specify a null String object, the connection factory property XMSC_PASSWORD is used instead.

Returns:
The Connection object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_SECURITY_EXCEPTION

getHandle – Get Handle

Interface:

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

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

Parameters:
None

Returns:
The handle for the connection factory.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the ConnectionFactory object is a null object.

Parameters:

None

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

Inherited methods

The following methods are inherited from the PropertyContext class:

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

ConnectionMetaData

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 406.

Methods

Summary of methods:

Method	Description
getHandle	Get the handle that a C application would use to access the connection metadata.
getJMSXProperties	Get a list of the names of the JMS defined message properties supported by the connection.
isNull	Determine whether the ConnectionMetaData object is a null object.

getHandle – Get Handle

Interface:

```
xmsHConnMetaData getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the connection metadata.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getJMSXProperties – Get JMS Defined Message Properties

Interface:

```
Iterator getJMSXProperties() const;
```

Get a list of the names of the JMS defined message properties supported by the connection.

The method returns an iterator that encapsulates a list of Property objects, where each Property object encapsulates the name of a JMS defined message property. The application can then use the iterator to retrieve the name of each JMS defined message property in turn.

Note: The equivalent JMS method performs a slightly different function. The JMS method returns an enumeration of the names of the JMS defined message properties.

Parameters:

None

Returns:

The Iterator object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the ConnectionMetaData object is a null object.

Parameters:

None

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

Inherited methods

The following methods are inherited from the PropertyContext class:

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

Destination for the C++ class

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

Inheritance hierarchy:

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

For a list of the XMS defined properties of a Destination object, see "Properties of Destination" on page 407.

Constructors

Summary of constructors:

Constructor	Description
Destination	Create a destination using the specified destination type and name.
Destination	Create a destination using the specified uniform resource identifier (URI).

Destination – Create Destination (specifying a type and name)

Interface:

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

Create a destination using the specified destination type and name.

For a destination that is a queue, this constructor does not create the queue in the messaging server. You must create the queue before an application can call this constructor.

Parameters:

destinationType (input)
The type of the destination, which must be one of the following values:

XMS_DESTINATION_TYPE_QUEUE

XMS_DESTINATION_TYPE_TOPIC

destinationName (input)
A String object encapsulating the name of the destination, which can be the name of a queue or the name of a topic.

If the destination is a IBM MQ queue, you can specify the name of the destination in either of the following ways:

QName

QMgrName/QName

where *QName* is the name of a IBM MQ queue, and *QMgrName* is the name of a IBM MQ queue manager. The IBM MQ queue name resolution process uses the values of *QName* and *QMgrName* to determine the actual destination queue. For more information about the queue name resolution process, see the *IBM MQ documentation*.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Destination – Create Destination (using a URI)

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 URI take the default values.

For a destination that is a queue, this constructor does not create the queue in the messaging server. You must create the queue before an application can call this constructor.

Parameters:

URI (input)
A String object encapsulating the URI.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
~Destination	Delete the destination.
getHandle	Get the handle that a C application would use to access the destination.
getName	Get the name of the destination.
getTypeId	Get the type of the destination.
isNull	Determine whether the Destination object is a null object.
toString	Get the name of the destination in the format of a uniform resource identifier (URI).

~Destination – Delete Destination

Interface:

```
virtual ~Destination();
```

Delete the destination.

For a destination that is a queue, this method does not delete the queue in the

messaging server unless the queue was created for an XMS temporary queue.

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

Parameters:

None

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHDest getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the destination.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getName – Get Destination Name

Interface:

```
String getName() const;
```

Get the name of the destination.

Parameters:

None

Returns:

A String object encapsulating the name of the destination. The name is either the name of a queue or the name of a topic.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getTypeId – Get Destination Type

Interface:

```
xmsDESTINATION_TYPE getTypeId();
```

Get the type of the destination.

Parameters:

None

Returns:

The type of the destination, which is one of the following values:

XMS_DESTINATION_TYPE_QUEUE

XMS_DESTINATION_TYPE_TOPIC

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Destination object is a null object.

Parameters:

None

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

toString – Get Destination Name as URI

Interface:

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

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

Parameters:

None

Returns:

A String object encapsulating the URI. The URI is either a queue URI or a topic URI.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

Exception

If XMS detects an error while processing a call to a method, XMS throws an exception. An exception is an object that encapsulates information about the error.

Inheritance hierarchy:

```
std::exception
    |
    +--- xms::Exception
```

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.

Methods

Summary of methods:

Method	Description
~Exception	Delete the exception and any linked exceptions.
dump	Dump the exception to the specified C++ output stream as formatted text.
getErrorCode	Get the error code.
getErrorData	Get the free format data that provides additional information about the error.
getErrorString	Get the string of characters that describes the error.
getHandle	Get the handle for the internal error block that XMS creates for the exception.
getJMSException	Get the exception code.
getLinkedException	Get a pointer to the next exception in the chain of exceptions.
isNull	Determine whether the Exception object is a null object.

~Exception – Delete Exception

Interface:

```
virtual ~Exception() throw();
```

Delete the exception and any linked exceptions.

Parameters:

None

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

dump – Dump Exception

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.

Returns:

Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getErrorCode – Get Error Code

Interface:

```
xmsINT getErrorCode() const;
```

Get the error code.

Parameters:
None

Returns:
The error code.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getErrorData – Get Error Data

Interface:

```
String getErrorData() const;
```

Get the free format data that provides additional information about the error.

Parameters:
None

Returns:
A String object encapsulating the error data.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getErrorString – Get Error String

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

Returns:
A String object encapsulating the error string.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHErrorBlock getHandle() const;
```

Get the handle for the internal error block that XMS creates for the exception.

Parameters:
None

Returns:
The handle for the error block.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getJMSException – Get Exception Code

Interface:

```
xmsJMSEXP_TYPE getJMSException() const;
```

Get the exception code.

Parameters:
None

Returns:
The exception code.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getLinkedException – Get Linked Exception

Interface:

```
Exception * getLinkedException() const;
```

Get a pointer to the next exception in the chain of exceptions.

Parameters:
None

Returns:

A pointer to an exception. The method returns a null pointer if there are no more exceptions in the chain.

Note: Because the getLinkedException() method returns a pointer to a linked exception, the application must release the object using the C++ delete operator.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Exception object is a null object.

Parameters:

None

Returns:

v xmsTRUE, if the Exception object is a null object.

v xmsFALSE, if the Exception object is not a null object.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

ExceptionListener

An application uses an exception listener to be notified asynchronously of a problem with a connection.

Inheritance hierarchy:

None

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. In other situations, an exception listener can provide a more immediate way of learning about a problem with a connection than waiting until the next synchronous call to XMS.

Methods

Summary of methods:

Method	Description
onException	Notify the application of a problem with a connection.

onException – On Exception

Interface:

```
virtual xmsVOID onException(Exception *exception);
```

Notify the application of a problem with a connection.

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 listeners, see "Exception listeners in C++" on page 74.

Parameters:

exception (input)
A pointer to an exception created by XMS.

Returns:
Void

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.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::IllegalStateException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

InitialContext

An application uses an InitialContext object to create objects from object definitions that are retrieved from a repository of administered objects.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::InitialContext
```

For a list of the XMS defined properties of an InitialContext object, see "Properties of InitialContext" on page 408.

Constructors

Summary of constructors:

Constructor	Description
InitialContext	Create an InitialContext object.

InitialContext – Create Initial Context

Interface:

```
InitialContext( const String & uri);
InitialContext & create( const String & uri);
```

Create an InitialContext object.

Note: The creation of the InitialContext object is done separately from the connection to the repository containing administered objects. This allows properties to be set on the InitialContext object prior to connection. For further details, see "InitialContext properties" on page 81.

Parameters:

uri (input)
A String object encapsulating a URI that identifies the name and location of a repository containing administered objects. The exact syntax of the URI depends on the context type. For further information, see "URI format for XMS initial contexts" on page 81.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
~InitialContext	Delete the InitialContext object. This frees all resources associated with the InitialContext object.
getHandle	Get the handle that a C application would use to access the InitialContext object.
isNull	Determine whether the InitialContext object is a null object.
lookup	Create an object from an object definition that is retrieved from the repository of administered objects.

~InitialContext – Delete Initial Context

Interface:

```
InitialContext:: ~InitialContext();
```

Delete the InitialContext object. This frees all resources associated with the InitialContext object.

If an application tries to delete an InitialContext object that is already deleted, the call is ignored.

Parameters:

None

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHInitialContext getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the InitialContext object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the InitialContext object is a null object.

Parameters:

None

Returns:

v xmsTRUE, if the InitialContext object is a null object.

v xmsFALSE, if the InitialContext object is not a null object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

lookup – Look Up Object in Initial Context

Interface:

```
PropertyContext * lookup(const String & objectName) const;
```

Create an object from an object definition that is retrieved from the repository of administered objects.

Parameters:

objectName (input)

A String object encapsulating the name of the administered object. The name can be either a simple name or a complex name. For further details, see "Retrieval of administered objects" on page 83.

Returns:

A pointer to the object that is created.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

InvalidClientIDException

XMS throws this exception if an application attempts to set a client identifier for a connection, but the client identifier is not valid or is already in use.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::InvalidClientIDException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

InvalidDestinationException

XMS throws this exception if an application specifies a destination that is not valid.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::InvalidDestinationException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

InvalidSelectorException

XMS throws this exception if an application provides a message selector expression whose syntax is not valid.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::InvalidSelectorException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

---

Iterator

An iterator encapsulates a list of objects. An application uses an iterator to access object in turn.

Inheritance hierarchy:

None

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

An application cannot create an iterator directly using a constructor. An iterator is created only by certain methods in order to pass a list of objects back to the application.

This class is a helper class.

Methods

Summary of methods:

Method	Description
~Iterator	Delete the iterator.
getHandle	Get the handle that a C application would use to access the iterator.
getNext	Move the cursor to the next object and get the object at the new position of the cursor.
hasNext	Check whether there are any more objects beyond the current position of the cursor.
isNull	Determine whether the Iterator object is a null object.
reset	Move the cursor back to a position before the first object.

~Iterator – Delete Iterator

Interface:

```
virtual ~Iterator();
```

Delete the iterator.

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

Parameters:

None

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHIterator getHandle() const;
```

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

Parameters:
None

Returns:
The handle for the iterator.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getNext – Get Next Object

Interface:

```
xmsVOID * getNext() const;
```

Move the cursor to the next object and get the object at the new position of the cursor.

Parameters:
None

Returns:
A pointer to the object.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

hasNext – Check for More Objects

Interface:

```
xmsBOOL hasNext();
```

Check whether there are any more objects beyond the current position of the cursor. The call does not move the cursor.

Parameters:
None

Returns:
v xmsTRUE, if there are more objects beyond the current position of the cursor.
v xmsFALSE, if there are no more objects beyond the current position of the cursor.

Thread context:
Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Iterator object is a null object.

Parameters:

None

Returns:

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

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

reset – Reset Iterator

Interface:

```
xmsVOID reset();
```

Move the cursor back to a position before the first object.

Parameters:

None

Returns:

Void

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

MapMessage

A map message is a message whose body comprises a set of name-value pairs, where each value has an associated data type.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
           |
           +--- xms::MapMessage
```

When an application gets the value of name-value pair, the value can be converted by XMS into another data type. For more information about this form of implicit conversion, see “Map messages” on page 98.

Methods

Summary of methods:

Method	Description
getBoolean	Get the boolean value identified by name from the body of the map message.
getByte	Get the byte identified by name from the body of the map message.
getBytes	Get the array of bytes identified by name from the body of the map message.
getChar	Get the character identified by name from the body of the map message.
getDouble	Get the double precision floating point number identified by name from the body of the map message.
getFloat	Get the floating point number identified by name from the body of the map message.
getInt	Get the integer identified by name from the body of the map message.
getLong	Get the long integer identified by name from the body of the map message.
getMap	Get a list of the name-value pairs in the body of the map message.
getObject	Get the value of a name-value pair, and its data type, from the body of the map message.
getShort	Get the short integer identified by name from the body of the map message.
getString	Get the string identified by name from the body of the map message.
itemExists	Check whether the body of the map message contains a name-value pair with the specified name.
setBoolean	Set a boolean value in the body of the map message.
setByte	Set a byte in the body of the map message.
setBytes	Set an array of bytes in the body of the map message.
setChar	Set a 2-byte character in the body of the map message.
setDouble	Set a double precision floating point number in the body of the map message.
setFloat	Set a floating point number in the body of the map message.
setInt	Set an integer in the body of the map message.
setLong	Set a long integer in the body of the map message.
setObject	Set a value, with a specified data type, in the body of the map message.
setShort	Set a short integer in the body of the map message.
setString	Set a string in the body of the map message.

getBoolean – Get Boolean Value

Interface:

```
xmsBOOL getBoolean(const String & name) const;
```

Get the boolean value identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the boolean value.

Returns:
The boolean value retrieved from the body of the map message.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getByte – Get Byte

Interface:

```
xmsSBYTE getByte(const String & name) const;
```

Get the byte identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the byte.

Returns:
The byte retrieved from the body of the map message. No data conversion is performed on the byte.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getBytes – Get Bytes

Interface:

```
xmsINT getBytes(const String & name,
                xmsSBYTE *buffer,
                const xmsINT bufferLength,
                xmsINT *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, see "C++ methods that return a byte array" on page 67.

Parameters:

name (input)
A String object encapsulating the name that identifies the array of bytes.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length of the array is not returned.

Returns:
The number of bytes in the array.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getChar – Get Character

Interface:

```
xmsCHAR16 getChar(const String & name) const;
```

Get the character identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the character.

Returns:
The character retrieved from the body of the map message.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getDouble – Get Double Precision Floating Point Number

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:

name (input)
A String object encapsulating the name that identifies the double precision floating point number.

Returns:
The double precision floating point number retrieved from the body of the map message.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getFloat – Get Floating Point Number

Interface:

```
xmsFLOAT getFloat(const String & name) const;
```

Get the floating point number identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the floating point number.

Returns:
The floating point number retrieved from the body of the map message.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getInt – Get Integer

Interface:

```
xmsINT getInt(const String & name) const;
```

Get the integer identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the integer.

Returns:
The integer retrieved from the body of the map message.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getLong – Get Long Integer

Interface:

```
xmsLONG getLong(const String & name) const;
```

Get the long integer identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the long integer.

Returns:
The long integer retrieved from the body of the map message.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getMap – Get Name-Value Pairs

Interface:

```
Iterator getMap() const;
```

Get a list of the name-value pairs in the body of the map message.

The method returns an iterator that encapsulates a list of Property objects, where each Property object encapsulates a name-value pair. The application can then use the iterator to access each name-value pair in turn.

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:

None

Returns:

The Iterator object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getObject – Get Object

Interface:

```
xmsOBJECT_TYPE getObject(const String & name,
                         xmsSBYTE *buffer,
                         const xmsINT bufferLength,
                         xmsINT *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, see "C++ methods that return a byte array" on page 67.

Parameters:

name (input)
A String object encapsulating the name of the name-value pair.

buffer (output)
The buffer to contain the value, which is returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value in bytes. If the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

Returns:

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_BYTEARRAY
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

Exceptions:

XMS_X_GENERAL_EXCEPTION

getShort – Get Short Integer

Interface:

```
xmsSHORT getShort(const String & name) const;
```

Get the short integer identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the short integer.

Returns:

The short integer retrieved from the body of the map message.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getString – Get String

Interface:

```
String getString(const String & name) const;
```

Get the string identified by name from the body of the map message.

Parameters:

name (input)
A String object encapsulating the name that identifies the string in the body of the map message.

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:

v XMS_X_GENERAL_EXCEPTION

itemExists – Check Name-Value Pair Exists

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:

name (input)
A String object encapsulating the name of the name-value pair.

Returns:

- xmsTRUE, if the body of the map message contains a name-value pair with the specified name.
- xmsFALSE, if the body of the map message does not contain a name-value pair with the specified name.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setBoolean – Set Boolean Value

Interface:

```
xmsVOID setBoolean(const String & name,
                   const xmsBOOL value);
```

Set a boolean value in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the boolean value in the body of the map message.

value (input)
The boolean value to be set.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setByte – Set Byte

Interface:

```
xmsVOID setByte(const String & name,
                const xmsSBYTE value);
```

Set a byte in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the byte in the body of the map message.

value (input)
The byte to be set.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setBytes – Set Bytes

Interface:

```
xmsVOID setBytes(const String & name,
                 const xmsSBYTE *value,
                 const xmsINT length);
```

Set an array of bytes in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the array of bytes in the body of the map message.

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

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

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setChar – Set Character

Interface:

```
xmsVOID setChar(const String & name,
                const xmsCHAR16 value);
```

Set a 2-byte character in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the character in the body of the map message.

value (input)
The character to be set.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setDouble – Set Double Precision Floating Point Number

Interface:

```
xmsVOID setDouble(const String & name,
                  const xmsDOUBLE value);
```

Set a double precision floating point number in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the double precision floating point number in the body of the map message.

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

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setFloat – Set Floating Point Number

Interface:

```
xmsVOID setFloat(const String & name,
                 const xmsFLOAT value);
```

Set a floating point number in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the floating point number in the body of the map message.

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

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setInt – Set Integer

Interface:

```
xmsVOID setInt(const String & name,
               const xmsINT value);
```

Set an integer in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the integer in the body of the map message.

value (input)
The integer to be set.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setLong – Set Long Integer

Interface:

```
xmsVOID setLong(const String & name,
                const xmsLONG value);
```

Set a long integer in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the long integer in the body of the map message.

value (input)
The long integer to be set.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setObject – Set Object

Interface:

```
xmsVOID setObject(const String & name,
                  const xmsOBJECT_TYPE objectType,
                  const xmsSBYTE *value,
                  const xmsINT length);
```

Set a value, with a specified data type, in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the value in the body of the map message.

objectType (input)
The data type of the value, which must be one of the following object types:

XMS_OBJECT_TYPE_BOOL
XMS_OBJECT_TYPE_BYTE
XMS_OBJECT_TYPE_BYTEARRAY
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

value (input)
An array of bytes containing the value to be set.

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

Returns:
Void

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setShort – Set Short Integer

Interface:

```
xmsVOID setShort(const String & name,
                 const xmsSHORT value);
```

Set a short integer in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the short integer in the body of the map message.

value (input)
The short integer to be set.

Returns:
Void

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setString – Set String

Interface:

```
xmsVOID setString(const String & name,
                  const String value);
```

Set a string in the body of the map message.

Parameters:

name (input)
A String object encapsulating the name to identify the string in the body of the map message.

value (input)
A String object encapsulating the string to be set.

Returns:
Void

Exceptions:
v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the Message class:

clearBody, clearProperties, getHandle, getJMSCorrelationID,getJMSDeliveryMode, getJMSDestination, getJMSExpiration, getJMSMessageID, getJMSPriority, getJMSRedelivered,

getJMSReplyTo, getJMSTimestamp, getJMSType, getProperties, isNull, propertyExists, setJMSCorrelationID, setJMSDeliveryMode, setJMSDestination, setJMSExpiration, setJMSMessageID, setJMSPriority, setJMSRedelivered, setJMSReplyTo, setJMSTimestamp, setJMSType

The following methods are inherited from the PropertyContext class:

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

---

Message

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

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
```

For a list of the JMS message header fields in a Message object, see "Header fields in an XMS message" on page 91. For a list of the JMS defined properties of a Message object, see "JMS-defined properties of a message" on page 93. For a list of the IBM defined properties of a Message object, see "IBM-defined properties of a message" on page 94.

Methods

Summary of methods:

Method	Description
~Message	Delete the message.
acknowledge	Acknowledge this message and all previously unacknowledged messages received by the session.
clearBody	Clear the body of the message.
clearProperties	Clear the properties of the message.
getHandle	Get the handle that a C application would use to access the message.
getJMSCorrelationID	Get the correlation identifier of the message.
getJMSDeliveryMode	Get the delivery mode of the message.
getJMSDestination	Get the destination of the message.
getJMSExpiration	Get the expiration time of the message.
getJMSMessageID	Get the message identifier of the message.
getJMSPriority	Get the priority of the message.
getJMSRedelivered	Get an indication of whether the message is being re-delivered.
getJMSReplyTo	Get the destination where a reply to the message is to be sent.
getJMSTimestamp	Get the time when the message was sent.
getJMSType	Get the type of the message.
getProperties	Get a list of the properties of the message.
isNull	Determine whether the Message object is a null object.
propertyExists	Check whether the message has a property with the specified name.
setJMSCorrelationID	Set the correlation identifier of the message.
setJMSDeliveryMode	Set the delivery mode of the message.
setJMSDestination	Set the destination of the message.
setJMSExpiration	Set the expiration time of the message.

Method	Description
setJMSMessageID	Set the message identifier of the message.
setJMSPriority	Set the priority of the message.
setJMSRedelivered	Indicate whether the message is being re-delivered.
setJMSReplyTo	Set the destination where a reply to the message is to be sent.
setJMSTimestamp	Set the time when the message is sent.
setJMSType	Set the type of the message.

~Message – Delete Message

Interface:

```
virtual ~Message();
```

Delete the message.

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

Parameters:

None

Exceptions:

- XMS_X_GENERAL_EXCEPTION

acknowledge – Acknowledge

Interface:

```
xmsVOID acknowledge();
```

Acknowledge this message and all previously unacknowledged messages received by the session.

An application can call this method if the acknowledgement mode of the session is XMSC_CLIENT_ACKNOWLEDGE. Calls to the method are ignored if the session has any other acknowledgement mode or is transacted.

Messages that have been received but not acknowledged might be re-delivered.

For more information about acknowledging messages, see "Message acknowledgement" on page 29.

Parameters:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

clearBody – Clear Body

Interface:

```
xmsVOID clearBody();
```

Clear the body of the message. The header fields and message properties are not cleared.

If an application clears a message body, the body is left in the same state as an empty body in a newly created message. The state of an empty body in a newly created message depends on the type of message body. For more information, see "The body of an XMS message" on page 95.

An application can clear a message body at any time, no matter what state the body is in. If a message body is read-only, the only way that an application can write to the body is for the application to clear the body first.

Parameters:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

clearProperties – Clear Properties

Interface:

```
xmsVOID clearProperties();
```

Clear the properties of the message. The header fields and the message body are not cleared.

If an application clears the properties of a message, the properties become readable and writable.

An application can clear the properties of a message at any time, no matter what state the properties are in. If the properties of a message are read-only, the only way that the properties can become writable is for the application to clear the properties first.

Parameters:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHMsg getHandle() const;
```

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

Parameters:
None

Returns:
The handle for the message.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getJMSCorrelationID – Get JMSCorrelationID

Interface:

```
String getJMSCorrelationID() const;
```

Get the correlation identifier of the message.

Parameters:
None

Returns:
A String object encapsulating the correlation identifier.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getJMSDeliveryMode – Get JMSDeliveryMode

Interface:

```
xmsINT getJMSDeliveryMode() const;
```

Get the delivery mode of the message. The delivery mode is set by the MessageProducer.send() call when the message is sent.

Parameters:
None

Returns:
The delivery mode of the message, which is one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

For a newly created message that has not been sent, the delivery mode is XMSC_DELIVERY_PERSISTENT. For a message that has been received, the method returns the delivery mode that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the delivery mode by calling setJMSDeliveryMode().

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getJMSDestination – Get JMSDestination

Interface:

```
Destination getJMSDestination() const;
```

Get the destination of the message. The destination is set by the MessageProducer.send() call when the message is sent.

Parameters:

None

Returns:

The Destination object.

For a newly created message that has not been sent, the method returns a null Destination object and throws an exception unless the sending application sets a destination by calling setJMSDestination(). For a message that has been received, the method returns a Destination object for the destination that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the destination by calling setJMSDestination().

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getJMSExpiration – Get JMSExpiration

Interface:

```
xmsLONG getJMSExpiration() const;
```

Get the expiration time of the message.

The expiration time is set by the MessageProducer.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 0, the MessageProducer.send() call sets the expiration time to 0 to indicate that the message does not expire.

XMS discards expired messages and does not deliver them to applications.

Parameters:

None

Returns:

The expiration time of the message.

For a newly created message that has not been sent, the expiration time is 0 unless the sending application sets a different expiration time by calling setJMSExpiration(). For a message that has been received, the method returns the expiration time that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the expiration time by calling setJMSExpiration().

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getJMSMessageID – Get JMSMessageID

Interface:

```
String getJMSMessageID() const;
```

Get the message identifier of the message. The message identifier is set by the MessageProducer.send() call when the message is sent.

Parameters:

None

Returns:

A String object encapsulating the message identifier.

For a message that has been received, the method returns the message identifier that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the message identifier by calling setJMSMessageID().

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Notes:

1. If a message has no message identifier, the method throws an exception.

getJMSPriority – Get JMSPriority

Interface:

```
xmsINT getJMSPriority() const;
```

Get the priority of the message. The priority is set by the MessageProducer.send() call when the message is sent.

Parameters:

None

Returns:

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 setJMSPriority(). For a message that has been received, the method returns the priority that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the priority by calling setJMSPriority().

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getJMSRedelivered – Get JMSRedelivered

Interface:

```
xmsBOOL getJMSRedelivered() const;
```

Get an indication of whether the message is being re-delivered. The indication is set by the MessageConsumer.receive() call when the message is received.

Parameters:

None

Returns:

- xmsTRUE, if the message is being re-delivered.
- xmsFALSE, if the message is not being re-delivered.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getJMSReplyTo – Get JMSReplyTo

Interface:

```
Destination getJMSReplyTo() const;
```

Get the destination where a reply to the message is to be sent.

Parameters:

None

Returns:

A Destination object for the destination where a reply to the message is to be sent. A null Destination object means that no reply is expected.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getJMSTimestamp – Get JMSTimestamp

Interface:

```
xmsLONG getJMSTimestamp() const;
```

Get the time when the message was sent. The time stamp is set by the MessageProducer.send() call when the message is sent and is expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

Parameters:

None

Returns:

The time when the message was sent.

For a newly created message that has not been sent, the time stamp is 0 unless the sending application sets a different time stamp by calling setJMSTimestamp(). For a message that has been received, the method returns the time stamp that was set by the MessageProducer.send() call when the message was sent unless the receiving application changes the time stamp by calling setJMSTimestamp().

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Notes:

1. If the time stamp is undefined, the method returns 0 but throws no exception.

getJMSType – Get JMSType

Interface:

```
String getJMSType() const;
```

Get the type of the message.

Parameters:

None

Returns:

A String encapsulating the type of the message. If data conversion is required, this is the type after conversion.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getProperties – Get Properties

Interface:

```
Iterator getProperties() const;
```

Get a list of the properties of the message.

The method returns an iterator that encapsulates a list of Property objects. The application can then use the iterator to access each property 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:

None

Returns:

The Iterator object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Message object is a null object.

Parameters:

None

Returns:

v xmsTRUE, if the Message object is a null object.

- xmsFALSE, if the Message object is not a null object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

propertyExists – Check Property Exists

Interface:

```
xmsBOOL propertyExists(const String & propertyName) const;
```

Check whether the message has a property with the specified name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

Returns:

- xmsTRUE, if the message has a property with the specified name.
- xmsFALSE, if the message does not have a property with the specified name.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setJMSCorrelationID – Set JMSCorrelationID

Interface:

```
xmsVOID setJMSCorrelationID(const String correlID);
```

Set the correlation identifier of the message.

Parameters:

correlID (input)
A String object encapsulating the correlation identifier.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setJMSDeliveryMode – Set JMSDeliveryMode

Interface:

```
xmsVOID setJMSDeliveryMode(const xmsINT 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 MessageProducer.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:

deliveryMode (input)
The delivery mode of the message, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setJMSDestination – Set JMSDestination

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 MessageProducer.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:

destination (input)
A Destination object representing the destination of the message.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

setJMSExpiration – Set JMSExpiration

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 MessageProducer.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:

expiration (input)
The expiration time of the message expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setJMSMessageID – Set JMSMessageID

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 MessageProducer.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:

msgID (input)
A String object encapsulating the message identifier.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setJMSPriority – Set JMSPriority

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 MessageProducer.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:

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.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setJMSRedelivered – Set JMSRedelivered

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 MessageProducer.send() call when the message is sent, and is ignored and replaced by the MessageConsumer.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:

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.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setJMSReplyTo – Set JMSReplyTo

Interface:

```
xmsVOID setJMSReplyTo(const Destination & destination);
```

Set the destination where a reply to the message is to be sent.

Parameters:

destination (input)
A Destination object representing the destination where a reply to the message is to be sent. A null Destination object means that no reply is expected.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setJMSTimestamp – Set JMSTimestamp

Interface:

```
xmsVOID setJMSTimestamp(const xmsLONG timeStamp);
```

Set the time when the message is sent.

A time stamp set by this method before the message is sent is ignored and replaced by the MessageProducer.send() call when the message is sent. However, you can use this method to change the time stamp of a message that has been received.

Parameters:

timeStamp (input)
The time when the message is sent expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

Returns:

Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setJMSType – Set JMSType

Interface:

```
xmsVOID setJMSType(const String & type);
```

Set the type of the message.

Parameters:

type (input)
A String object encapsulating the type of the message.

Returns:

Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

MessageConsumer

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

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::MessageConsumer
```

For a list of the XMS defined properties of a MessageConsumer object, see "Properties of MessageConsumer" on page 414.

Methods

Summary of methods:

Method	Description
close	Close the message consumer.
getHandle	Get the handle that a C application would use to access the message consumer.
getMessageListener	Get a pointer to the message listener that is registered with the message consumer.
getMessageSelector	Get the message selector for the message consumer.
isNull	Determine whether the MessageConsumer object is a null object.

Method	Description
receive	Receive the next message for the message consumer. The call waits indefinitely for a message, or until the message consumer is closed.
receive	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.
receiveNoWait	Receive the next message for the message consumer if one is available immediately.
setMessageListener	Register a message listener with the message consumer.

close – Close Message Consumer

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:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHMsgConsumer getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the message consumer.

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getMessageListener – Get Message Listener

Interface:

```
MessageListener * getMessageListener() const;
```

Get a pointer to the message listener that is registered with the message consumer.

For more information about using message listeners, see “Message listeners in C++” on page 72.

Parameters:

None

Returns:

A pointer to the message listener. If no message listener is registered with the message consumer, the method returns a null pointer.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getMessageSelector – Get Message Selector

Interface:

```
String getMessageSelector() const;
```

Get the message selector for the message consumer.

Parameters:

None

Returns:

A String object encapsulating the message selector expression. If data conversion is required, this is the message selector expression after conversion. If the message consumer does not have a message selector, the method returns a null String object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the MessageConsumer object is a null object.

Parameters:

None

Returns:

v `xmsTRUE`, if the MessageConsumer object is a null object.

v `xmsFALSE`, if the MessageConsumer object is not a null object.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

receive – Receive

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:

None

Returns:

A pointer to the Message object. If the message consumer is closed while the call is waiting for a message, the method returns a pointer to a null Message object.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

receive – Receive (with a wait interval)

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:

waitInterval (input)

The time, in milliseconds, that the call waits for a message. If you specify a wait interval of 0, the call waits indefinitely for a message.

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 method returns a pointer to a null Message object but throws no exception.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

receiveNoWait – Receive with No Wait

Interface:

```
Message * receiveNoWait() const;
```

Receive the next message for the message consumer if one is available immediately.

Parameters:

None

Returns:

A pointer to a Message object. If no message is available immediately, the method returns a pointer to a null Message object.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

setMessageListener – Set Message Listener

Interface:

```
xmsVOID setMessageListener(const MessageListener *lsr);
```

Register a message listener with the message consumer.

For more information about using message listeners, see "Message listeners in C++" on page 72.

Parameters:

lsr (input)

A pointer to the message listener. If a message listener is already registered with the message consumer, you can cancel the registration by specifying a null pointer instead.

Returns:

Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

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.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::MessageEOFException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

MessageFormatException

XMS throws this exception if XMS encounters a message with a format that is not valid.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::MessageFormatException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

MessageListener

An application uses a message listener to receive messages asynchronously.

Inheritance hierarchy:

None

Methods

Summary of methods:

Method	Description
onMessage	Deliver a message asynchronously to the message consumer.

onMessage – On Message

Interface:

```
virtual xmsVOID onMessage(Message *message);
```

Deliver a message asynchronously to the message consumer.

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 listeners, see “Message listeners in C++” on page 72.

Parameters:

message (input)
A pointer to the Message object.

Returns:

Void

MessageNotReadableException

XMS throws this exception if an application attempts to read the body of a message that is write-only.

Inheritance hierarchy:

```
std::exception
   |
   + --- xms::Exception
            |
            + --- xms::MessageNotReadableException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

MessageNotWritableException

XMS throws this exception if an application attempts to write to the body of a message that is read-only.

Inheritance hierarchy:

```
std::exception
   |
   + --- xms::Exception
            |
            + --- xms::MessageNotWritableException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

MessageProducer

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

Inheritance hierarchy:

```
xms::PropertyContext
   |
   + --- xms::MessageProducer
```

For a list of the XMS defined properties of a MessageProducer object, see "Properties of MessageProducer" on page 414.

Methods

Summary of methods:

Method	Description
close	Close the message producer.
getDeliveryMode	Get the default delivery mode for messages sent by the message producer.
getDestination	Get the destination for the message producer.

Method	Description
getDisableMsgID	Get an indication of whether a receiving application requires message identifiers to be included in messages sent by the message producer.
getDisableMsgTS	Get an indication of whether a receiving application requires time stamps to be included in messages sent by the message producer.
getHandle	Get the handle that a C application would use to access the message producer.
getPriority	Get the default priority for messages sent by the message producer.
getTimeToLive	Get the default length of time that a message exists before it expires.
isNull	Determine whether the MessageProducer object is a null object.
send	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.
send	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.
send	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.
send	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.
setDeliveryMode	Set the default delivery mode for messages sent by the message producer.
setDisableMsgID	Indicate whether a receiving application requires message identifiers to be included in messages sent by the message producer.
setDisableMsgTS	Indicate whether a receiving application requires time stamps to be included in messages sent by the message producer.
setPriority	Set the default priority for messages sent by the message producer.
setTimeToLive	Set the default length of time that a message exists before it expires.

close – Close Message Producer

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:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getDeliveryMode – Get Default Delivery Mode

Interface:

```
xmsINT getDeliveryMode() const;
```

Get the default delivery mode for messages sent by the message producer.

Parameters:

None

Returns:

The default delivery mode, which is one of the following values:

```
XMSC_DELIVERY_PERSISTENT
XMSC_DELIVERY_NOT_PERSISTENT
```

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getDestination – Get Destination

Interface:

```
Destination getDestination() const;
```

Get the destination for the message producer.

Parameters:

None

Returns:

The Destination object. If the message producer does not have a destination, the method returns a null Destination object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getDisableMsgID – Get Disable Message ID Flag

Interface:

```
xmsBOOL getDisableMsgID() const;
```

Get an indication of whether a receiving application requires message identifiers to be included in messages sent by the message producer.

Parameters:

None

Returns:

- xmsTRUE, if a receiving application does not require message identifiers to be included in messages sent by the message producer.
- xmsFALSE, if a receiving application does require message identifiers to be included in messages sent by the message producer.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getDisableMsgTS – Get Disable Time Stamp Flag

Interface:

```
xmsBOOL getDisableMsgTS() const;
```

Get an indication of whether a receiving application requires time stamps to be included in messages sent by the message producer.

Parameters:

None

Returns:

- xmsTRUE, if a receiving application does not require time stamps to be included in messages sent by the message producer.
- xmsFALSE, if a receiving application does require time stamps to be included in messages sent by the message producer.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHMsgProducer getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the message producer.

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getPriority – Get Default Priority

Interface:

```
xmsINT getPriority() const;
```

Get the default priority for messages sent by the message producer.

Parameters:

None

Returns:

The default message priority. The value is an integer in the range 0, the lowest priority, to 9, the highest priority.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getTimeToLive – Get Default Time to Live

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:

None

Returns:

The default time to live in milliseconds. A value of 0 means that a message never expires.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the MessageProducer object is a null object.

Parameters:

None

Returns:

v xmsTRUE, if the MessageProducer object is a null object.

v xmsFALSE, if the MessageProducer object is not a null object.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

send – Send

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:

message (input)

The Message object.

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

send – Send (specifying a delivery mode, priority, and time to live)

Interface:

```
xmsVOID send(const Message & message,
             const xmsINT 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:

message (input)
The Message object.

deliveryMode (input)
The delivery mode for the message, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_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.

timeToLive (input)
The time to live for the message in milliseconds. A value of 0 means that the message never expires.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

send – Send (to a specified destination)

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:

destination (input)
The Destination object.

message (input)
The Message object.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

send – Send (to a specified destination, specifying a delivery mode, priority, and time to live)

Interface:

```
xmsVOID send(const Destination & destination,
             const Message & message,
             const xmsINT 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:

destination (input)
The Destination object.

message (input)
The Message object.

deliveryMode (input)
The delivery mode for the message, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_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.

timeToLive (input)
The time to live for the message in milliseconds. A value of 0 means that the message never expires.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_FORMAT_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

setDeliveryMode – Set Default Delivery Mode

Interface:

```
xmsVOID setDeliveryMode(const xmsINT deliveryMode);
```

Set the default delivery mode for messages sent by the message producer.

Parameters:

deliveryMode (input)
The default delivery mode, which must be one of the following values:

XMSC_DELIVERY_PERSISTENT

XMSC_DELIVERY_NOT_PERSISTENT

The default value is XMSC_DELIVERY_PERSISTENT.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setDisableMsgID – Set Disable Message ID Flag

Interface:

```
xmsVOID setDisableMsgID(const xmsBOOL msgIDDisabled);
```

Indicate whether a receiving application requires message identifiers to be included in messages sent by the message producer.

On a connection to a queue manager, this flag is ignored. On a connection to a service integration bus, the flag is honoured.

Parameters:

msgIDDisabled (input)
The value xmsTRUE means that a receiving application does not require message identifiers to be included in messages sent by the message producer. The value xmsFALSE means that a receiving application does require message identifiers. The default value is xmsFALSE.

Returns:
Void

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setDisableMsgTS – Set Disable Time Stamp Flag

Interface:

```
xmsVOID setDisableMsgTS(const xmsBOOL timeStampDisabled);
```

Indicate whether a receiving application requires time stamps to be included in messages sent by the message producer.

On a connection to a queue manager, or on a connection to a service integration bus, the flag is honoured.

Parameters:

timeStampDisabled (input)
The value xmsTRUE means that a receiving application does not require time stamps to be included in messages sent by the message producer. The value xmsFALSE means that a receiving application does require time stamps. The default value is xmsFALSE.

Returns:
Void

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setPriority – Set Default Priority

Interface:

```
xmsVOID setPriority(const xmsINT priority);
```

Set the default priority for messages sent by the message producer.

Parameters:

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.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setTimeToLive – Set Default Time to Live

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:

timeToLive (input)
The default time to live in milliseconds. The default value is 0, which means that a message never expires.

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

ObjectMessage

An object message is a message whose body comprises a serialized Java object.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
           |
           +--- xms::ObjectMessage
```

Methods

Summary of methods:

Method	Description
getObject	Get the object that forms the body of the object message.
setObject	Set the string that forms the body of the object message.

getObject – Get Object as Bytes

Interface:

```
xmsINT getObject(xmsSBYTE *buffer,
                 xmsINT bufferLength,
                 xmsINT *actualLength);
```

Get the object that forms the body of the object message.

For more information about how to use this method, see "C++ methods that return a byte array" on page 67.

Parameters:

buffer (output)
The buffer to contain the object, which is returned as an array of bytes.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the object is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the object in bytes. If you specify a null pointer on input, the length is not returned.

Returns:
The length of the object in bytes.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

Notes:

1. If the buffer is not large enough to store the whole object, XMS returns the object truncated to the length of the buffer, sets the actualLength parameter to the actual length of the object, and returns an error.
2. If any other error occurs while attempting to get the object, XMS reports the error but does not set the actualLength parameter.

setObject – Set Object as Bytes

Interface:

```
xmsVOID setObject(xmsSBYTE *value,
                  xmsINT length);
```

Set the string that forms the body of the object message.

Parameters:

value (input)
An array of bytes representing the object to be set.

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods

The following methods are inherited from the Message class:

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

The following methods are inherited from the PropertyContext class:

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

Property

A Property object represents a property of an object.

Inheritance hierarchy:
None

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

If an application sets the property value attribute of a Property object, the property value replaces any previous value the attribute had.

This class is a helper class.

Constructors

Summary of constructors:

Constructor	Description
Property	Copy the Property object.
Property	Create a Property object with a property name, a property value, and a property type.
Property	Create a Property object with no property value or property type.

Property – Copy Property

Interface:

```
Property(const Property & property);

Property & duplicate(const Property & property);
```

Copy the Property object.

Parameters:

property (input)
The Property object.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Property – Create Property

Interface:

```
Property(const String & propertyName,
         const xmsBOOL propertyValue);

Property(const String & propertyName,
         const xmsSBYTE *propertyValue,
         xmsINT length);

Property(const String & propertyName,
         const xmsSBYTE 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:

v XMS_X_GENERAL_EXCEPTION

Property – Create Property (with no property value or property type)

Interface:

```
Property(const String & propertyName);

Property & create(const String & propertyName);
```

Create a Property object with no property value or property type.

Parameters:

propertyName (input)
A String object encapsulating the property name.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
~Property	Delete the Property object.
getBoolean	Get the boolean property value from the Property object.
getByte	Get the byte property value from the Property object.
getByteArray	Get the byte array property value from the Property object.
getChar	Get the 2-byte character property value from the Property object.
getDouble	Get the double precision floating point property value from the Property object.
getFloat	Get the floating point property value from the Property object.

Method	Description
getHandle	Get the handle that a C application would use to access the Property object.
getInt	Get the integer property value from the Property object.
getLong	Get the long integer property value from the Property object.
getShort	Get the short integer property value from the Property object.
getString	Get the string property value from the Property object.
getTypeId	Get the property type from the Property object.
isNull	Determine whether the Property object is a null object.
isTypeId	Check whether the Property object has the specified property type.
name	Get the property name from the Property object.
setBoolean	Set a boolean property value in the Property object and set the property type.
setByte	Set a byte property value in the Property object and set the property type.
setByteArray	Set a byte array property value in the Property object and set the property type.
setChar	Set a 2-byte character property value in the Property object and set the property type.
setDouble	Set a double precision floating point property value in the Property object and set the property type.
setFloat	Set a floating point property value in the Property object and set the property type.
setInt	Set an integer property value in the Property object and set the property type.
setLong	Set a long integer property value in the Property object and set the property type.
setShort	Set a short integer property value in the Property object and set the property type.
setString	Set a string property value in the Property object and set the property type.

~Property – Delete Property

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:

None

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getBoolean – Get Boolean Property Value

Interface:

```
xmsBOOL getBoolean() const;
```

Get the boolean property value from the Property object.

Parameters:
None

Returns:
The boolean property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getByte – Get Byte Property Value

Interface:

```
xmsSBYTE getByte() const;
```

Get the byte property value from the Property object.

Parameters:
None

Returns:
The byte property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getByteArray – Get Byte Array Property Value

Interface:

```
xmsINT getByteArray(xmsSBYTE *propertyValue,
                    const xmsINT length,
                    xmsINT *actualLength) const;
```

Get the byte array property value from the Property object.

For more information about how to use this method, see "C++ methods that return a byte array" on page 67.

Parameters:

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length is not returned.

Returns:
The length of the property value in bytes.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getChar – Get Character Property Value

Interface:

```
xmsCHAR16 getChar() const;
```

Get the 2-byte character property value from the Property object.

Parameters:
None

Returns:
The 2-byte character property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getDouble – Get Double Precision Floating Point Property Value

Interface:

```
xmsDOUBLE getDouble() const;
```

Get the double precision floating point property value from the Property object.

Parameters:
None

Returns:
The double precision floating point property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getFloat – Get Floating Point Property Value

Interface:

```
xmsFLOAT getFloat() const;
```

Get the floating point property value from the Property object.

Parameters:
None

Returns:
The floating point property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHProperty getHandle() const;
```

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

Parameters:
None

Returns:
The handle for the Property object.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getInt – Get Integer Property Value

Interface:

```
xmsINT getInt() const;
```

Get the integer property value from the Property object.

Parameters:
None

Returns:
The integer property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getLong – Get Long Integer Property Value

Interface:

```
xmsLONG getLong() const;
```

Get the long integer property value from the Property object.

Parameters:
None

Returns:
The long integer property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getShort – Get Short Integer Property Value

Interface:

```
xmsSHORT getShort() const;
```

Get the short integer property value from the Property object.

Parameters:
None

Returns:
The short integer property value.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getString – Get String Property Value

Interface:

```
String getString() const;
```

Get the string property value from the Property object.

Parameters:
None

Returns:
A String object encapsulating the string property value. If data conversion is required, this is the string after conversion.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getTypeId – Get Property Type

Interface:

```
xmsPROPERTY_TYPE getTypeId() const;
```

Get the property type from the Property object.

Parameters:
None

Returns:

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

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Property object is a null object.

Parameters:

None

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

isTypeId – Check Property Type

Interface:

```
xmsBOOL isTypeId(const xmsPROPERTY_TYPE propertyType) const;
```

Check whether the Property object has the specified property type.

Parameters:

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

Returns:

- v xmsTRUE, if the Property object has the specified property type.
- v xmsFALSE, if the Property object does not have the specified property type.

Thread context:
Any

Exceptions:
- v XMS_X_GENERAL_EXCEPTION

name – Get Property Name

Interface:

```
String name() const;
```

Get the property name from the Property object.

Parameters:
None

Returns:
A String object encapsulating the property name.

Thread context:
Any

Exceptions:
- v XMS_X_GENERAL_EXCEPTION

setBoolean – Set Boolean Property Value

Interface:

```
xmsVOID setBoolean(const xmsBOOL propertyValue);
```

Set a boolean property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The boolean property value.

Returns:
Void

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setByte – Set Byte Property Value

Interface:

```
xmsVOID setByte(const xmsSBYTE propertyValue);
```

Set a byte property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The byte property value.

Returns:
Void

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setByteArray – Set Byte Array Property Value

Interface:

```
xmsVOID setByteArray(const xmsBYTE *propertyValue,
                     const xmsINT length);
```

Set a byte array property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The property value, which is an array of bytes.

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

Returns:
Void

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

setChar – Set Character Property Value

Interface:

```
xmsVOID setChar(const xmsCHAR16 propertyValue);
```

Set a 2-byte character property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The 2-byte character property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setDouble – Set Double Precision Floating Point Property Value

Interface:

```
xmsVOID setDouble(const xmsDOUBLE propertyValue);
```

Set a double precision floating point property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The double precision floating point property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setFloat – Set Floating Point Property Value

Interface:

```
xmsVOID setFloat(const xmsFLOAT propertyValue);
```

Set a floating point property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The floating point property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setInt – Set Integer Property Value

Interface:

```
xmsVOID setInt(const xmsINT propertyValue);
```

Set an integer property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The integer property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setLong – Set Long Integer Property Value

Interface:

```
xmsVOID setLong(const xmsLONG propertyValue);
```

Set a long integer property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The long integer property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setShort – Set Short Integer Property Value

Interface:

```
xmsVOID setShort(const xmsSHORT propertyValue);
```

Set a short integer property value in the Property object and set the property type.

Parameters:

propertyValue (input)
The short integer property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

setString – Set String Property Value

Interface:

```
xmsVOID setString(const String & propertyValue);
```

Set a string property value in the Property object and set the property type.

Parameters:

propertyValue (input)
A String object encapsulating the string property value.

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

PropertyContext

PropertyContext is an abstract superclass that contains methods that get and set properties. These methods are inherited by other classes.

Inheritance hierarchy:
None

Methods

Summary of methods:

Method	Description
getBooleanProperty	Get the value of the boolean property identified by name.
getByteProperty	Get the value of the byte property identified by name.
getBytesProperty	Get the value of the byte array property identified by name.
getCharProperty	Get the value of the 2-byte character property identified by name.
getDoubleProperty	Get the value of the double precision floating point property identified by name.
getFloatProperty	Get the value of the floating point property identified by name.
getIntProperty	Get the value of the integer property identified by name.
getLongProperty	Get the value of the long integer property identified by name.
getObjectProperty	Get the value and data type of the property identified by name.
getProperty	Get a Property object for the property identified by name.
getShortProperty	Get the value of the short integer property identified by name.
getStringProperty	Get the value of the string property identified by name.
setBooleanProperty	Set the value of the boolean property identified by name.
setByteProperty	Set the value of the byte property identified by name.
setBytesProperty	Set the value of the byte array property identified by name.
setCharProperty	Set the value of the 2-byte character property identified by name.

Method	Description
setDoubleProperty	Set the value of the double precision floating point property identified by name.
setFloatProperty	Set the value of the floating point property identified by name.
setIntProperty	Set the value of the integer property identified by name.
setLongProperty	Set the value of the long integer property identified by name.
setObjectProperty	Set the value and data type of a property identified by name.
setProperty	Set the value of a property using a Property object.
setShortProperty	Set the value of the short integer property identified by name.
setStringProperty	Set the value of the string property identified by name.

getBooleanProperty – Get Boolean Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getByteProperty – Get Byte Property

Interface:

```
xmsSBYTE 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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getBytesProperty – Get Byte Array Property

Interface:

```
xmsINT getBytesProperty(const String & propertyName,
                        xmsSBYTE *propertyValue,
                        const xmsINT length,
                        xmsINT *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 67.

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 XMSC_QUERY_SIZE instead, 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. If you specify a null pointer on input, the length of the array is not returned.

Returns:
The number of bytes in the array.

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getCharProperty – Get Character Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getDoubleProperty – Get Double Precision Floating Point Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getFloatProperty – Get Floating Point Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getIntProperty – Get Integer Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getLongProperty – Get Long Integer Property

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.

Returns:
The value of the property.

Thread context:
Determined by the subclass

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getObjectProperty – Get Object Property

Interface:

```
xmsOBJECT_TYPE getObjectProperty(const String & propertyName,
                                 xmsSBYTE *propertyValue,
                                 const xmsINT length,
                                 xmsINT *actualLength);
```

Get the value and data type of the property identified by name.

For more information about how to use this method, see "C++ methods that return a byte array" on page 67.

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 returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, 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 the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

Returns:

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_BYTEARRAY

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

Thread context:

Determined by the subclass

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getProperty – Get Property

Interface:

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

Get a Property object for the property identified by name.

Parameters:

propertyName (input)

A String object encapsulating the name of the property.

Returns:

The Property object.

Thread context:

Determined by the subclass

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getShortProperty – Get Short Integer Property

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.

Returns:

The value of the property.

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getStringProperty – Get String Property

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.

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:
v XMS_X_GENERAL_EXCEPTION

setBooleanProperty – Set Boolean Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:
v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setByteProperty – Set Byte Property

Interface:

```
xmsVOID setByteProperty(const String & propertyName,
                        const xmsSBYTE 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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setBytesProperty – Set Byte Array Property

Interface:

```
xmsVOID setBytesProperty(const String & propertyName,
                         const xmsSBYTE *propertyValue,
                         const xmsINT 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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setCharProperty – Set Character Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setDoubleProperty – Set Double Precision Floating Point Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setFloatProperty – Set Floating Point Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setIntProperty – Set Integer Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setLongProperty – Set Long Integer Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setObjectProperty – Set Object Property

Interface:

```
xmsVOID setObjectProperty(const String & propertyName,
                          const xmsOBJECT_TYPE objectType,
                          const xmsSBYTE *propertyValue,
                          const xmsINT length);
```

Set the value and data type of a property identified by name.

Parameters:

propertyName (input)
A String object encapsulating the name of the property.

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_BYTEARRAY
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

propertyValue (input)
The value of the property as an array of bytes.

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

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setProperty – Set Property

Interface:

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

Set the value of a property using a Property object.

Parameters:

property (input)
The Property object.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setShortProperty – Set Short Integer Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

setStringProperty – Set String Property

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.

Returns:
Void

Thread context:
Determined by the subclass

Exceptions:
- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

QueueBrowser

An application uses a queue browser to browse messages on a queue without removing them.

Inheritance hierarchy:

```
xms::PropertyContext
    |
    +--- xms::QueueBrowser
```

Methods

Summary of methods:

Method	Description
close	Close the queue browser.
getEnumeration	Get a list of the messages on the queue.
getHandle	Get the handle that a C application would use to access the queue browser.
getMessageSelector	Get the message selector for the queue browser.
getQueue	Get the queue associated with the queue browser.
isNull	Determine whether the QueueBrowser object is a null object.

close – Close Queue Browser

Interface:

```
xmsVOID close();
```

Close the queue browser.

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

Parameters:
None

Returns:
Void

Exceptions:
- XMS_X_GENERAL_EXCEPTION

getEnumeration – Get Messages

Interface:

```
Iterator getEnumeration() const;
```

Get a list of the messages on the queue.

The method returns an iterator that encapsulates a list of Message objects. The order of the Message objects in the list is the same as the order in which the messages would be retrieved from the queue. The application can then use the iterator to browse each message in turn.

The iterator is updated dynamically as messages are put on the queue and removed from the queue. Each time the application calls Iterator.getNext() to browse the next message on the queue, the message returned reflects the current contents of the queue.

If an application calls this method more than once for a given queue browser, each call returns a new iterator. The application can therefore use more than one iterator to browse the messages on a queue and maintain multiple positions within the queue.

Parameters:

None

Returns:

The Iterator object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHQueueBrowser getHandle() const;
```

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

Parameters:

None

Returns:

The handle for the queue browser.

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getMessageSelector – Get Message Selector

Interface:

```
String getMessageSelector() const;
```

Get the message selector for the queue browser.

Parameters:

None

Returns:

A String object encapsulating the message selector expression. If data conversion is required, this is the message selector expression after conversion. If the queue browser does not have a message selector, the method returns a null String object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

getQueue – Get Queue

Interface:

```
Destination getQueue() const;
```

Get the queue associated with the queue browser.

Parameters:

None

Returns:

A Destination object representing the queue.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the QueueBrowser object is a null object.

Parameters:

None

Returns:

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

Thread context:

Any

Exceptions:

- XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

Requestor

An application uses a requestor to send a request message and then wait for, and receive, the reply.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Requestor
```

Constructors

Summary of constructors:

Constructor	Description
Requestor	Create a requestor.

Requestor – Create Requestor

Interface:

```
Requestor(const Session & session,
          const Destination & destination);
```

Create a requestor.

Parameters:

session (input)
A Session object. The session must not be transacted and must have one of the following acknowledgement modes:

XMSC_AUTO_ACKNOWLEDGE

XMSC_DUPS_OK_ACKNOWLEDGE

destination (input)
A Destination object representing the destination where the application can send request messages.

Thread context:
The session associated with the requestor

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
close	Close the requestor.
getHandle	Get the handle that a C application would use to access the requestor.
isNull	Determine whether the Requestor object is a null object.
request	Send a request message and then wait for, and receive, a reply from the application that receives the request message.

close – Close Requestor

Interface:

```
xmsVOID close();
```

Close the requestor.

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

Note: When an application closes a requestor, the associated session does not close as well. In this respect, XMS behaves differently compared to JMS.

Parameters:
None

Returns:
Void

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHRequestor getHandle() const;
```

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

Parameters:
None

Returns:
The handle for the requestor.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Requestor object is a null object.

Parameters:
None

Returns:
v xmsTRUE, if the Requestor object is a null object.
v xmsFALSE, if the Requestor object is not a null object.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

request – Request

Interface:

```
Message * request(const Message & requestMessage) const;
```

Send a request message and then wait for, and receive, a reply from the application that receives the request message.

A call to this method blocks until a reply is received or until the session ends, whichever is the sooner.

Parameters:

requestMessage (input)
The Message object encapsulating the request message.

Returns:
A pointer to the Message object encapsulating the reply message.

Note: Because the method returns a pointer to an object the application must release the object using the C++ delete operator.

Thread context:
The session associated with the requestor

Exceptions:

v XMS_X_GENERAL_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

ResourceAllocationException

XMS throws this exception if XMS cannot allocate the resources required by a method.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::ResourceAllocationException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

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.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::SecurityException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

Session

A session is a single threaded context for sending and receiving messages.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Session
```

For a list of the XMS defined properties of a Session object, see "Properties of Session" on page 414.

Methods

Summary of methods:

Method	Description
close	Close the session.
commit	Commit all messages processed in the current transaction.
createBrowser	Create a queue browser for the specified queue.
createBrowser	Create a queue browser for the specified queue using a message selector.
createBytesMessage	Create a bytes message.
createConsumer	Create a message consumer for the specified destination.
createConsumer	Create a message consumer for the specified destination using a message selector.
createConsumer	Create a message consumer for the specified destination using a message selector and, if the destination is a topic, specifying whether the message consumer receives the messages published by its own connection.
createDurableSubscriber	Create a durable subscriber for the specified topic.
createDurableSubscriber	Create a durable subscriber for the specified topic using a message selector and specifying whether the durable subscriber receives the messages published by its own connection.
createMapMessage	Create a map message.
createMessage	Create a message that has no body.
createObjectMessage	Create an object message.
createProducer	Create a message producer to send messages to the specified destination.

Method	Description
createQueue	Create a Destination object to represent a queue in the messaging server.
createStreamMessage	Create a stream message.
createTemporaryQueue	Create a temporary queue.
createTemporaryTopic	Create a temporary topic.
createTextMessage	Create a text message with an empty body.
createTextMessage	Create a text message whose body is initialized with the specified text.
createTopic	Create a Destination object to represent a topic.
getAcknowledgeMode	Get the acknowledgement mode for the session.
getHandle	Get the handle that a C application would use to access the session.
getTransacted	Determine whether the session is transacted.
isNull	Determine whether the Session object is a null object.
recover	Recover the session.
rollback	Rollback all messages processed in the current transaction.
unsubscribe	Delete a durable subscription.

close – Close Session

Interface:

```
xmsVOID close();
```

Close the session. If the session is transacted, any transaction in progress is rolled back.

All objects dependent on the session are deleted. For information about which objects are deleted, see "Object Deletion" on page 42.

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

Parameters:

None

Returns:

Void

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

commit – Commit

Interface:

```
xmsVOID commit();
```

Commit all messages processed in the current transaction.

The session must be a transacted session.

Parameters:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION
- XMS_X_TRANSACTION_ROLLED_BACK_EXCEPTION

createBrowser – Create Queue Browser

Interface:

```
QueueBrowser createBrowser(const Destination & queue) const;
```

Create a queue browser for the specified queue.

Parameters:

queue (input)
A Destination object representing the queue.

Returns:
The QueueBrowser object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

createBrowser – Create Queue Browser (with message selector)

Interface:

```
QueueBrowser createBrowser(const Destination & queue
                           const String & messageSelector) const;
```

Create a queue browser for the specified queue using a message selector.

Parameters:

queue (input)
A Destination object representing the queue.

messageSelector (input)
A String object encapsulating a message selector expression. Only those messages with properties that match the message selector expression are delivered to the queue browser.

A null String object means that there is no message selector for the queue browser.

Returns:
The QueueBrowser object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

createBytesMessage – Create Bytes Message

Interface:

```
BytesMessage createBytesMessage() const;
```

Create a bytes message.

Parameters:

None

Returns:

The BytesMessage object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createConsumer – Create Consumer

Interface:

```
MessageConsumer createConsumer(const Destination & destination) const;
```

Create a message consumer for the specified destination.

Parameters:

destination (input)
The Destination object.

Returns:

The MessageConsumer object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

createConsumer – Create Consumer (with message selector)

Interface:

```
MessageConsumer createConsumer(const Destination & destination,
                               const String & messageSelector) const;
```

Create a message consumer for the specified destination using a message selector.

Parameters:

destination (input)
The Destination object.

messageSelector (input)
A String object encapsulating a message selector expression. Only those messages with properties that match the message selector expression are delivered to the message consumer.

A null String object means that there is no message selector for the message consumer.

Returns:

The MessageConsumer object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

createConsumer – Create Consumer (with message selector and local message flag)

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, if the destination is a topic, specifying whether the message consumer receives the messages published by its own connection.

Parameters:

destination (input)
The Destination object.

messageSelector (input)
A String object encapsulating a message selector expression. Only those messages with properties that match the message selector expression are delivered to the message consumer.

A null String object means that there is no message selector for the message consumer.

noLocal (input)
The value `xmsTRUE` means that the message consumer does not receive the messages published by its own connection. The value `xmsFALSE` means that the message consumer does receive the messages published by its own connection. The default value is `xmsFALSE`.

Returns:
The MessageConsumer object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

createDurableSubscriber – Create Durable Subscriber

Interface:

```
MessageConsumer
createDurableSubscriber(const Destination & topic,
                        const String & subscriptionName) const;
```

Create a durable subscriber for the specified topic.

For more information about durable subscribers, see “Durable subscribers” on page 37.

Parameters:

topic (input)
A Destination object representing the topic. The topic must not be a temporary topic.

subscriptionName (input)
A String object encapsulating a name that identifies the durable subscription. The name must be unique within the client identifier for the connection.

Returns:
The MessageConsumer object representing the durable subscriber.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

createDurableSubscriber – Create Durable Subscriber (with message selector and local message flag)

Interface:

```
MessageConsumer createDurableSubscriber(const Destination & topic,
                                        const String & subscriptionName;
                                        const String & messageSelector,
                                        const xmsBOOL noLocal) const;
```

Create a durable subscriber for the specified topic using a message selector and specifying whether the durable subscriber receives the messages published by its own connection.

For more information about durable subscribers, see “Durable subscribers” on page 37.

Parameters:

topic (input)
A Destination object representing the topic. The topic must not be a temporary topic.

subscriptionName (input)
A String object encapsulating a name that identifies the durable subscription. The name must be unique within the client identifier for the connection.

messageSelector (input)
A String object encapsulating a message selector expression. Only those messages with properties that match the message selector expression are delivered to the durable subscriber.

A null String object means that there is no message selector for the durable subscriber.

noLocal (input)
The value `xmsTRUE` means that the durable subscriber does not

receive the messages published by its own connection. The value xmsFALSE means that the durable subscriber does receive the messages published by its own connection. The default value is xmsFALSE.

Returns:

The MessageConsumer object representing the durable subscriber.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_INVALID_SELECTOR_EXCEPTION

createMapMessage – Create Map Message

Interface:

```
MapMessage createMapMessage() const;
```

Create a map message.

Parameters:

None

Returns:

The MapMessage object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createMessage – Create Message

Interface:

```
Message createMessage() const;
```

Create a message that has no body.

Parameters:

None

Returns:

The Message object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createObjectMessage – Create Object Message

Interface:

```
ObjectMessage createObjectMessage() const;
```

Create an object message.

Parameters:

None

Returns:

The ObjectMessage object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createProducer – Create Producer

Interface:

```
MessageProducer createProducer(const Destination & destination) const;
```

Create a message producer to send messages to the specified destination.

Parameters:

destination (input)

The Destination object.

If you specify a null Destination object, the message producer is created without a destination. In this case, the application must specify a destination every time it uses the message producer to send a message.

Returns:

The MessageProducer object.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION

createQueue – Create Queue

Interface:

```
Destination createQueue(const String & queueName) const;
```

Create a Destination object to represent a queue in the messaging server.

This method does not create the queue in the messaging server. You must create the queue before an application can call this method.

Parameters:

queueName (input)

A String object encapsulating the name of the queue, or encapsulating a uniform resource identifier (URI) that identifies the queue.

Returns:

The Destination object representing the queue.

Exceptions:

- XMS_X_GENERAL_EXCEPTION

createStreamMessage – Create Stream Message

Interface:

```
StreamMessage createStreamMessage() const;
```

Create a stream message.

Parameters:

None

Returns:

The StreamMessage object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createTemporaryQueue – Create Temporary Queue

Interface:

```
Destination createTemporaryQueue() const;
```

Create a temporary queue.

The scope of the temporary queue is the connection. Only the sessions created by the connection can use the temporary queue.

The temporary queue remains until it is explicitly deleted, or the connection ends, whichever is the sooner.

For more information about temporary queues, see "Temporary destinations" on page 36.

Parameters:

None

Returns:

The Destination object representing the temporary queue.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createTemporaryTopic – Create Temporary Topic

Interface:

```
Destination createTemporaryTopic() const;
```

Create a temporary topic.

The scope of the temporary topic is the connection. Only the sessions created by the connection can use the temporary topic.

The temporary topic remains until it is explicitly deleted, or the connection ends, whichever is the sooner.

For more information about temporary topics, see "Temporary destinations" on page 36.

Parameters:

None

Returns:

The Destination object representing the temporary topic.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createTextMessage – Create Text Message

Interface:

```
TextMessage createTextMessage() const;
```

Create a text message with an empty body.

Parameters:

None

Returns:

The TextMessage object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createTextMessage – Create Text Message (initialized)

Interface:

```
TextMessage createTextMessage(const String & text) const;
```

Create a text message whose body is initialized with the specified text.

Parameters:

text (input)

A String object encapsulating the text to initialize the body of the text message.

None

Returns:

The TextMessage object.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

createTopic – Create Topic

Interface:

```
Destination createTopic(const String & topicName) const;
```

Create a Destination object to represent a topic.

Parameters:

topicName (input)
A String object encapsulating the name of the topic, or encapsulating a uniform resource identifier (URI) that identifies the topic.

Returns:
The Destination object representing the topic.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getAcknowledgeMode – Get Acknowledgement Mode

Interface:

```
xmsINT getAcknowledgeMode() const;
```

Get the acknowledgement mode for the session. The acknowledgement mode is specified when the session is created.

A session that is transacted has no acknowledgement mode.

For more information about acknowledgement modes, see "Message acknowledgement" on page 29.

Parameters:
None

Returns:
The acknowledgement mode. Provided the session is not transacted, the acknowledgement mode is one of the following values:

XMSC_AUTO_ACKNOWLEDGE

XMSC_CLIENT_ACKNOWLEDGE

XMSC_DUPS_OK_ACKNOWLEDGE

If the session is transacted, the method returns XMSC_SESSION_TRANSACTED instead.

Exceptions:
v XMS_X_GENERAL_EXCEPTION

getHandle – Get Handle

Interface:

```
xmsHSess getHandle() const;
```

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

Parameters:
None

Returns:
The handle for the session.

Thread context:
Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

getTransacted – Determine Whether Transacted

Interface:

```
xmsBOOL getTransacted() const;
```

Determine whether the session is transacted.

Parameters:

None

Returns:

v xmsTRUE, if the session is transacted.

v xmsFALSE, if the session is not transacted.

Exceptions:

v XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the Session object is a null object.

Parameters:

None

Returns:

v xmsTRUE, if the Session object is a null object.

v xmsFALSE, if the Session object is not a null object.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

recover – Recover

Interface:

```
xmsVOID recover() const;
```

Recover the session. Message delivery is stopped and then restarted with the oldest unacknowledged message.

The session must not be a transacted session.

For more information about recovering a session, see “Message acknowledgement” on page 29.

Parameters:
None

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

rollback – Rollback

Interface:

```
xmsVOID rollback() const;
```

Rollback all messages processed in the current transaction.

The session must be a transacted session.

Parameters:
None

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

unsubscribe – Unsubscribe

Interface:

```
xmsVOID unsubscribe(const String & subscriptionName) const;
```

Delete a durable subscription. The messaging server deletes the record of the durable subscription that it is maintaining and does not send any more messages to the durable subscriber.

An application cannot delete a durable subscription in any of the following circumstances:

- While there is an active message consumer for the durable subscription
- While a consumed message is part of a pending transaction
- While a consumed message has not been acknowledged

Parameters:

subscriptionName (input)
A String object encapsulating the name that identifies the durable subscription.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_INVALID_DESTINATION_EXCEPTION
- XMS_X_ILLEGAL_STATE_EXCEPTION

Inherited methods

The following methods are inherited from the PropertyContext class:

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

StreamMessage

A stream message is a message whose body comprises a stream of values, where each value has an associated data type.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
           |
           +--- xms::StreamMessage
```

The contents of the body are written and read sequentially.

When an application reads a value from the message stream, the value can be converted by XMS into another data type. For more information about this form of implicit conversion, see “Stream messages” on page 99.

Methods

Summary of methods:

Method	Description
readBoolean	Read a boolean value from the message stream.
readByte	Read a signed 8-bit integer from the message stream.
readBytes	Read an array of bytes from the message stream.
readChar	Read a 2-byte character from the message stream.
readDouble	Read an 8-byte double precision floating point number from the message stream.
readFloat	Read a 4-byte floating point number from the message stream.
readInt	Read a signed 32-bit integer from the message stream.
readLong	Read a signed 64-bit integer from the message stream.
readObject	Read a value from the message stream, and return its data type.
readShort	Read a signed 16-bit integer from the message stream.
readString	Read a string from the message stream.
reset	Put the body of the message into read-only mode and reposition the cursor at the beginning of the message stream.
writeBoolean	Write a boolean value to the message stream.
writeByte	Write a byte to the message stream.
writeBytes	Write an array of bytes to the message stream.
writeChar	Write a character to the message stream as 2 bytes, high order byte first.

Method	Description
writeDouble	Convert a double precision floating point number to a long integer and write the long integer to the message stream as 8 bytes, high order byte first.
writeFloat	Convert a floating point number to an integer and write the integer to the message stream as 4 bytes, high order byte first.
writeInt	Write an integer to the message stream as 4 bytes, high order byte first.
writeLong	Write a long integer to the message stream as 8 bytes, high order byte first.
writeObject	Write a value, with a specified data type, to the message stream.
writeShort	Write a short integer to the message stream as 2 bytes, high order byte first.
writeString	Write a string to the message stream.

readBoolean – Read Boolean Value

Interface:

```
xmsBOOL readBoolean() const;
```

Read a boolean value from the message stream.

Parameters:

None

Returns:

The boolean value that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readByte – Read Byte

Interface:

```
xmsSBYTE readByte() const;
```

Read a signed 8-bit integer from the message stream.

Parameters:

None

Returns:

The byte that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readBytes – Read Bytes

Interface:

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

Read an array of bytes from the message stream.

Parameters:

buffer (output)
The buffer to contain the array of bytes that is read.

If the number of bytes in the array is less than or equal to the length of the buffer, the whole array is read into the buffer. If the number of bytes in the array is greater than the length of the buffer, the buffer is filled with part of the array, and an internal cursor marks the position of the next byte to be read. A subsequent call to readBytes() reads bytes from the array starting from the current position of the cursor.

If you specify a null pointer on input, the call skips over the array of bytes without reading it.

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 in the array remaining to be read. If there are no bytes remaining to be read from the array before the call, the value is `XMSC_END_OF_BYTEARRAY`.

If you specify a null pointer on input, the method returns no value.

Returns:
See the description of the returnedLength parameter.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readChar – Read Character

Interface:

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

Read a 2-byte character from the message stream.

Parameters:
None

Returns:
The character that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readDouble – Read Double Precision Floating Point Number

Interface:

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

Read an 8-byte double precision floating point number from the message stream.

Parameters:

None

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

readFloat – Read Floating Point Number

Interface:

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

Read a 4-byte floating point number from the message stream.

Parameters:

None

Returns:

The floating point number that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readInt – Read Integer

Interface:

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

Read a signed 32-bit integer from the message stream.

Parameters:

None

Returns:

The integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readLong – Read Long Integer

Interface:

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

Read a signed 64-bit integer from the message stream.

Parameters:

None

Returns:

The long integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readObject – Read Object

Interface:

```
xmsOBJECT_TYPE readObject(xmsSBYTE *buffer,
                          const xmsINT bufferLength,
                          xmsINT *actualLength) const;
```

Read a value from the message stream, and return its data type.

For more information about how to use this method, see "C++ methods that return a byte array" on page 67.

Parameters:

buffer (output)
The buffer to contain the value, which is returned as an array of bytes. If the value is a string and data conversion is required, this is the value after conversion.

If you specify a null pointer on input, the call skips over the value without reading it.

bufferLength (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the value is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the value in bytes. If the value is a string and data conversion is required, this is the length after conversion. If you specify a null pointer on input, the length is not returned.

Returns:

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_BYTEARRAY
- 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

Exceptions:
XMS_X_GENERAL_EXCEPTION

readShort – Read Short Integer

Interface:

```
xmsSHORT readShort() const;
```

Read a signed 16-bit integer from the message stream.

Parameters:
None

Returns:
The short integer that is read.

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

readString – Read String

Interface:

```
String readString() const;
```

Read a string from the message stream. If required, XMS converts the characters in the string into the local code page.

Parameters:
None

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

reset – Reset

Interface:

```
xmsVOID reset() const;
```

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

Parameters:

None

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_READABLE_EXCEPTION
- XMS_X_MESSAGE_EOF_EXCEPTION

writeBoolean – Write Boolean Value

Interface:

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

Write a boolean value to the message stream.

Parameters:

value (input)

The boolean value to be written.

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeByte – Write Byte

Interface:

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

Write a byte to the message stream.

Parameters:

value (input)

The byte to be written.

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeBytes – Write Bytes

Interface:

```
xmsVOID writeBytes(const xmsSBYTE *value,
                   const xmsINT length);
```

Write an array of bytes to the message stream.

Parameters:

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

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeChar – Write Character

Interface:

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

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

Parameters:

value (input)
The character to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeDouble – Write Double Precision Floating Point Number

Interface:

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

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

Parameters:

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeFloat – Write Floating Point Number

Interface:

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

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

Parameters:

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

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeInt – Write Integer

Interface:

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

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

Parameters:

value (input)
The integer to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeLong – Write Long Integer

Interface:

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

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

Parameters:

value (input)
The long integer to be written.

Returns:
Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION

v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeObject – Write Object

Interface:

```
xmsVOID writeObject(const xmsOBJECT_TYPE objectType,
                    const xmsSBYTE *value,
                    const xmsINT length);
```

Write a value, with a specified data type, to the message stream.

Parameters:

objectType (input)
The data type of the value, which must be one of the following object types:

XMS_OBJECT_TYPE_BOOL
XMS_OBJECT_TYPE_BYTE
XMS_OBJECT_TYPE_BYTEARRAY
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

value (input)
An array of bytes containing the value to be written.

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

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION

writeShort – Write Short Integer

Interface:

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

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

Parameters:

value (input)
The short integer to be written.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

writeString – Write String

Interface:

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

Write a string to the message stream.

Parameters:

value (input)
A String object encapsulating the string to be written.

Returns:
Void

Exceptions:

v XMS_X_GENERAL_EXCEPTION
v XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods

The following methods are inherited from the Message class:

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

The following methods are inherited from the PropertyContext class:

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

String

A String object encapsulates a string. This class is a helper class.

Inheritance hierarchy:
None

Constructors

Summary of constructors:

Constructor	Description
String	Create a String object that encapsulates a null string.
String	Create a String object from an array of bytes.
String	Create a String object from an array of characters.

String – Create String

Interface:

```
String();
```

Create a String object that encapsulates a null string.

Parameters:

None

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

String – Create String (from a byte array)

Interface:

```
String(const xmsSBYTE *value,
       const xmsINT 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:

v XMS_X_GENERAL_EXCEPTION

String – Create String (from a character array)

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:

v XMS_X_GENERAL_EXCEPTION

Methods

Summary of methods:

Method	Description
~String	Delete the String object.
c_str	Get a pointer to the string encapsulated by the String object.
concatenate	Concatenate the string encapsulated by the String object with the string encapsulated by a second String object.
equalTo	Determine whether the string encapsulated by the String object is equal to the string encapsulated by a second String object.
get	Get the string encapsulated by the String object.
isNull	Determine whether the String object is a null object.

~String – Delete String

Interface:

```
virtual ~String();
```

Delete the String object.

Parameters:

None

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

c_str – Get Pointer to String

Interface:

```
xmsCHAR * c_str() const;
```

Get a pointer to the string encapsulated by the String object.

Parameters:

None

Returns:

A pointer to the string encapsulated by the String object.

Thread context:

Any

Exceptions:

v XMS_X_GENERAL_EXCEPTION

concatenate – Concatenate Strings

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.

Returns:
The original String object encapsulating the concatenated strings.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

equalTo – Compare Strings

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.

Returns:
v xmsTRUE, if the two strings are equal.
v xmsFALSE, if the two strings are not equal.

Thread context:
Any

Exceptions:
v XMS_X_GENERAL_EXCEPTION

get – Get String

Interface:

```
xmsVOID get(xmsSBYTE *value,
            const xmsINT length,
            xmsINT *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 67.

Parameters:

value (output)
The buffer to contain the string.

length (input)
The length of the buffer in bytes. If you specify XMSC_QUERY_SIZE instead, the string is not returned, but its length is returned in the actualLength parameter.

actualLength (output)
The length of the string in bytes. If you specify a null pointer on input, the length is not returned.

Returns:
Void

Thread context:
Any

Exceptions:
- XMS_X_GENERAL_EXCEPTION

isNull – Check Whether Null

Interface:

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

Determine whether the String object is a null object.

Parameters:
None

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

TextMessage

A text message is a message whose body comprises a string.

Inheritance hierarchy:

```
xms::PropertyContext
   |
   +--- xms::Message
           |
           +--- xms::TextMessage
```

Methods

Summary of methods:

Method	Description
getText	Get the string that forms the body of the text message.
setText	Set the string that forms the body of the text message.

getText – get Text

Interface:

```
String getText() const;
```

Get the string that forms the body of the text message. If required, XMS converts the characters in the string into the local code page.

Parameters:

None

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

setText – Set Text

Interface:

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

Set the string that forms the body of the text message.

Parameters:

value (input)

A String object encapsulating the string to be set.

Returns:

Void

Exceptions:

- XMS_X_GENERAL_EXCEPTION
- XMS_X_MESSAGE_NOT_WRITABLE_EXCEPTION

Inherited methods

The following methods are inherited from the Message class:

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

The following methods are inherited from the PropertyContext class:

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

TransactionInProgressException

XMS throws this exception if an application requests an operation that is not valid because a transaction is in progress.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::TransactionInProgressException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

TransactionRolledBackException

XMS throws this exception if an application calls Session.commit() to commit the current transaction, but the transaction is subsequently rolled back.

Inheritance hierarchy:

```
std::exception
   |
   +--- xms::Exception
           |
           +--- xms::TransactionRolledBackException
```

Inherited methods

The following methods are inherited from the Exception class:

dump, getErrorCode, getErrorData, getErrorString, getHandle, getJMSException, getLinkedException, isNull

Chapter 15. Properties of XMS objects

This chapter documents the object properties defined by XMS.

The chapter contains the following sections:

- "Properties of Connection"
- "Properties of ConnectionFactory" on page 402
- "Properties of ConnectionMetaData" on page 406
- "Properties of Destination" on page 407
- "Properties of InitialContext" on page 408
- "Properties of Message" on page 409
- "Properties of MessageConsumer" on page 414
- "Properties of MessageProducer" on page 414
- "Properties of Session" on page 414

Each section lists the properties of an object of the specified type and provides a short description of each property.

This section also contains the following subsections:

- "Property definitions" on page 414

which provides a definition of each property.

If an application defines its own properties of the objects discussed in this section, it does not cause an error, but it might cause unpredictable results.

Properties of Connection

An overview of the properties of the Connection object, with links to more detailed reference information.

Table 36. Properties of Connection

Name of property	Description
XMSC_CLIENT_CCSID	The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer.
XMSC_WPM_CONNECTION_PROTOCOL	The communications protocol used for the connection to the messaging engine. This property is read-only.
XMSC_WPM_HOST_NAME	The host name or IP address of the system that contains the messaging engine to which the application is connected. This property is read-only.
XMSC_WPM_ME_NAME	The name of the messaging engine to which the application is connected. This property is read-only.
XMSC_WPM_PORT	The number of the port listened on by the messaging engine to which the application is connected. This property is read-only.

A Connection object also has read-only properties that are derived from the properties of the connection factory that was used to create the connection. These

properties are derived not only from the connection factory properties that were set at the time the connection was created, but also from the default values of the properties that were not set. The properties include only those that are relevant for the type of messaging server that the application is connected to. The names of the properties are the same as the names of the connection factory properties.

Properties of ConnectionFactory

An overview of the properties of the ConnectionFactory object, with links to more detailed reference information.

Table 37. Properties of ConnectionFactory

Name of property	Description
XMSC_ASYNC_EXCEPTIONS	This property determines whether XMS informs an ExceptionListener only when a connection is broken, or when any exception occurs asynchronously to a XMS API call. This applies to all Connections created from this ConnectionFactory that have an ExceptionListener registered.
XMSC_CLIENT_CCSID	The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer.
XMSC_CLIENT_ID	The client identifier for a connection.
XMSC_CONNECTION_TYPE	The type of messaging server to which an application connects.
XMSC_PASSWORD	A password that can be used to authenticate the application when it attempts to connect to a messaging server.
XMSC_USERID	A user identifier that can be used to authenticate the application when it attempts to connect to a messaging server.

Table 37. Properties of ConnectionFactory (continued)

Name of property	Description
XMSC_WMQ_CHANNEL	The name of the channel to be used for a connection.
XMSC_WMQ_CONNECTION_MODE	The mode by which an application connects to a queue manager.
XMSC_WMQ_FAIL_IF_QUIESCE	Whether calls to certain methods fail if the queue manager to which the application is connected is in a quiescing state.
XMSC_WMQ_HOST_NAME	The host name or IP address of the system on which a queue manager resides.
XMSC_WMQ_LOCAL_ADDRESS	For a connection to a queue manager, this property specifies the local network interface to be used, or the local port or range of local ports to be used, or both.
XMSC_WMQ_MSG_BATCH_SIZE	The maximum number of messages to be retrieved from a queue in one batch when using asynchronous message delivery. **Note:** This property has no effect for an application connected to a IBM MQ V7.0 and above queue manager unless the XMSC_WMQ_PROVIDER_VERSION property of the connection factory is set to a version number less than 7.

Table 37. Properties of ConnectionFactory (continued)

Name of property	Description
XMSC_WMQ_POLLING_INTERVAL	If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. **Note:** This property has no effect for an application connected to a IBM MQ V7.0 and above queue manager unless the XMSC_WMQ_PROVIDER_VERSION property of the connection factory is set to a version number less than 7.
XMSC_WMQ_PORT	The number of the port on which a queue manager listens for incoming requests.
XMSC_WMQ_PROVIDER_VERSION	The version, release, modification level and fix pack of the queue manager to which the application intends to connect.
XMSC_WMQ_QMGR_CCSID	The identifier (CCSID) of the coded character set, or code page, in which fields of character data defined in the Message Queue Interface (MQI) are exchanged between the XMS client and the IBM MQ client.
XMSC_WMQ_QUEUE_MANAGER	The name of the queue manager to connect to.
XMSC_WMQ_RECEIVE_EXIT	Identifies a channel receive exit, or a sequence of channel receive exits, to be run in succession.
XMSC_WMQ_RECEIVE_EXIT_INIT	The user data that is passed to channel receive exits when they are called.
XMSC_WMQ_SECURITY_EXIT	Identifies a channel security exit.
XMSC_WMQ_SECURITY_EXIT_INIT	The user data that is passed to a channel security exit when it is called.
XMSC_WMQ_SEND_EXIT	Identifies a channel send exit, or a sequence of channel send exits, to be run in succession.
XMSC_WMQ_SEND_EXIT_INIT	The user data that is passed to channel send exits when they are called.
XMSC_WMQ_SEND_CHECK_COUNT	The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted XMS session.
XMSC_WMQ_SHARE_CONV_ALLOWED	Whether a client connection can share its socket with other top-level XMS connections from the same process to the same queue manager, if the channel definitions match. This property is provided to allow complete isolation of Connections in separate sockets if required for application development, maintenance or operational reasons.
XMSC_WMQ_SSL_CERT_STORES	The locations of the servers that hold the certificate revocation lists (CRLs) to be used on an SSL connection to a queue manager.
XMSC_WMQ_SSL_CIPHER_SPEC	The name of the cipher spec to be used on a secure connection to a queue manager.

Table 37. Properties of ConnectionFactory (continued)

Name of property	Description
XMSC_WMQ_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on an SSL or TLS connection to a queue manager. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WMQ_SSL_CRYPTO_HW	Configuration details for the cryptographic hardware connected to the client system.
XMSC_WMQ_SSL_FIPS_REQUIRED	The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.
XMSC_WMQ_SSL_KEY_REPOSITORY	The location of the key database file in which keys and certificates are stored.
XMSC_WMQ_SSL_KEY_RESETCOUNT	The KeyResetCount represents the total number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated.
XMSC_WMQ_SSL_PEER_NAME	The peer name to be used on an SSL connection to a queue manager.
XMSC_WMQ_SYNCPOINT_ALL_GETS	Whether all messages must be retrieved from queues within syncpoint control.
XMSC_WMQ_TEMP_Q_PREFIX	The prefix used to form the name of the IBM MQ dynamic queue that is created when the application creates an XMS temporary queue.
XMSC_WMQ_TEMP_TOPIC_PREFIX	When creating temporary topics, XMS will generate a topic string of the form "TEMP/TEMPTOPICPREFIX/unique_id", or if this property is left with the default value, just "TEMP/unique_id". Specifying a non-empty value allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.
XMSC_WMQ_TEMPORARY_MODEL	The name of the IBM MQ model queue from which a dynamic queue is created when the application creates an XMS temporary queue.
XMSC_WMQ_WILDCARD_FORMAT	This property determines which version of wildcard syntax is to be used.
XMSC_WPM_BUS_NAME	For a connection factory, the name of the service integration bus that the application connects to or, for a destination, the name of the service integration bus in which the destination exists.
XMSC_WPM_CONNECTION_PROXIMITY	The connection proximity setting for the connection.
XMSC_WPM_DUR_SUB_HOME	The name of the messaging engine where all durable subscriptions for a connection or a destination are managed.
XMSC_WPM_LOCAL_ADDRESS	For a connection to a service integration bus, this property specifies the local network interface to be used, or the local port or range of local ports to be used, or both.
XMSC_WPM_NON_PERSISTENT_MAP	The reliability level of nonpersistent messages that are sent using the connection.
XMSC_WPM_PERSISTENT_MAP	The reliability level of persistent messages that are sent using the connection.
XMSC_WPM_PROVIDER_ENDPOINTS	A sequence of one or more endpoint addresses of bootstrap servers.

Table 37. Properties of ConnectionFactory (continued)

Name of property	Description
XMSC_WPM_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on an SSL or TLS connection to a WebSphere service integration bus messaging engine. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WPM_SSL_KEY_REPOSITORY	A path to the file that is the keyring file containing the public or private keys to be used in the secure connection.
XMSC_WPM_SSL_KEYRING_LABEL	The certificate to be used when authenticating with the server.
XMSC_WPM_SSL_KEYRING_PW	The password for the keyring file.
XMSC_WPM_SSL_KEYRING_STASH_FILE	The name of a binary file containing the password of the key repository file.
XMSC_WPM_SSL_FIPS_REQUIRED	The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.
XMSC_WPM_TARGET_GROUP	The name of a target group of messaging engines.
XMSC_WPM_TARGET_SIGNIFICANCE	The significance of the target group of messaging engines.
XMSC_WPM_TARGET_TRANSPORT_CHAIN	The name of the inbound transport chain that the application must use to connect to a messaging engine.
XMSC_WPM_TARGET_TYPE	The type of the target group of messaging engines.
XMSC_WPM_TEMP_Q_PREFIX	The prefix used to form the name of the temporary queue that is created in the service integration bus when the application creates an XMS temporary queue.
XMSC_WPM_TEMP_TOPIC_PREFIX	The prefix used to form the name of a temporary topic that is created by the application.

Properties of ConnectionMetaData

An overview of the properties of the ConnectionMetaData object, with links to more detailed reference information.

Table 38. Properties of ConnectionMetaData

Name of property	Description
XMSC_JMS_MAJOR_VERSION	The major version number of the JMS specification upon which XMS is based. This property is read-only.
XMSC_JMS_MINOR_VERSION	The minor version number of the JMS specification upon which XMS is based. This property is read-only.
XMSC_JMS_VERSION	The version identifier of the JMS specification upon which XMS is based. This property is read-only.
XMSC_MAJOR_VERSION	The version number of the XMS client. This property is read-only.
XMSC_MINOR_VERSION	The release number of the XMS client. This property is read-only.
XMSC_PROVIDER_NAME	The provider of the XMS client. This property is read-only.
XMSC_VERSION	The version identifier of the XMS client. This property is read-only.

Properties of Destination

An overview of the properties of the Destination object, with links to more detailed reference information.

Table 39. Properties of Destination

Name of property	Description
XMSC_DELIVERY_MODE	The delivery mode of messages sent to the destination.
XMSC_PRIORITY	The priority of messages sent to the destination.
XMSC_TIME_TO_LIVE	The time to live for messages sent to the destination.
XMSC_WMQ_CCSID	The identifier (CCSID) of the coded character set, or code page, that the strings of character data in the body of a message will be in when the XMS client forwards the message to the destination.
XMSC_WMQ_DUR_SUBQ	The name of the subscriber queue for a durable subscriber that is receiving messages from the destination. **Note:** This property has no effect for an application connected to a IBM MQ V7.0 (and above) queue manager unless the XMSC_WMQ_PROVIDER_VERSION property of the connection factory is set to a version number less than 7.
XMSC_WMQ_ENCODING	How numerical data in the body of a message will be represented when the XMS client forwards the message to the destination.
XMSC_WMQ_FAIL_IF_QUIESCE	Whether calls to certain methods fail if the queue manager to which the application is connected is in a quiescing state.
XMSC_WMQ_MESSAGE_BODY	This property determines whether a XMS application processes the MQRFH2 of a IBM MQ message as part of the message payload (that is, as part of the message body).
XMSC_WMQ_MQMD_MESSAGE_CONTEXT	Determines what level of message context is to be set by the XMS application. The application must be running with appropriate context authority for this property to take effect.
XMSC_WMQ_MQMD_READ_ENABLED	This property determines whether a XMS application can extract the values of MQMD fields or not.
XMSC_WMQ_MQMD_WRITE_ENABLED	This property determines whether a XMS application can set the values of MQMD fields or not.
XMSC_WMQ_READ_AHEAD_CLOSE_POLICY	This property determines, for messages being delivered to an asynchronous message listener, what happens to messages in the internal read ahead buffer when the message consumer is closed.
XMSC_WMQ_READ_AHEAD_ALLOWED	This property determines whether message consumers and queue browsers are allowed to use read ahead to get non-persistent, non-transactional messages from this destination into an internal buffer before receiving them.
XMSC_WMQ_PUT_ASYNC_ALLOWED	This property determines whether message producers are allowed to use asynchronous puts to send messages to this destination.
XMSC_WMQ_WILDCAD_FORMAT	This property determines which version of wildcard syntax is to be used.

Table 39. Properties of Destination (continued)

Name of property	Description
XMSC_WMQ_TARGET_CLIENT	Whether messages sent to the destination contain an MQRFH2 header.
XMSC_WMQ_TEMP_TOPIC_PREFIX	When creating temporary topics, XMS will generate a topic string of the form "TEMP/TEMPTOPICPREFIX/unique_id", or if this property is left with the default value, just "TEMP/unique_id". Specifying a non-empty value allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.
XMSC_WPM_BUS_NAME	For a connection factory, the name of the service integration bus that the application connects to or, for a destination, the name of the service integration bus in which the destination exists.
XMSC_WPM_TOPIC_SPACE	The name of the topic space that contains the topic.

Properties of InitialContext

An overview of the properties of the InitialContext object, with links to more detailed reference information.

Table 40. Properties of InitialContext

Name of property	Description
XMSC_IC_URL	For LDAP and FileSystem contexts, the address of the repository containing administered objects.

Table 41. Properties of InitialContext

Name of property	Description
XMSC_IC_PROVIDER_URL	Used to locate the JNDI naming directory so that the COS naming service does not need to be on the same machine as the web service.
XMSC_IC_SECURITY_AUTHENTICATION	Based on the Java Context interface SECURITY_AUTHENTICATION. This property is only applicable to the COS naming context.
XMSC_IC_SECURITY_CREDENTIALS	Based on the Java Context interface SECURITY_CREDENTIALS. This property is only applicable to the COS naming context.
XMSC_IC_SECURITY_PRINCIPAL	Based on the Java Context interface SECURITY_PRINCIPAL. This property is only applicable to the COS naming context.
XMSC_IC_SECURITY_PROTOCOL	Based on the Java Context interface SECURITY_PROTOCOL This property is only applicable to the COS naming context.
XMSC_IC_URL	For LDAP and FileSystem contexts, the address of the repository containing administered objects. For COS naming contexts, the address of the web service that looks up the objects in the directory.

Properties of Message

An overview of the properties of the Message object, with links to more detailed reference information.

Table 42. Properties of Message

Name of property	Description
"JMS_IBM_ArmCorrelator" on page 417	The Open Group Application Response Measurement Correlator property, set on a message. This IBM-defined property associates a unique Id with the application data in the message. Use JMS_TOG_ARM_Correlator in preference to this property.
JMS_IBM_CHARACTER_SET	The identifier (CCSID) of the coded character set, or code page, that the strings of character data in the body of the message will be in when the XMS client forwards the message to its intended destination. In XMS this property has a numeric value and maps to CCSID. However, this property is based on a JMS property so has a string type value and maps to the Java character set that represents this numeric CCSID.
JMS_IBM_ENCODING	How numerical data in the body of the message will be represented when the XMS client forwards the message to its intended destination.
JMS_IBM_EXCEPTIONMESSAGE	Text that describes why the message was sent to the exception destination. This property is read-only.
JMS_IBM_EXCEPTIONPROBLEMDESTINATION	The name of the destination that the message was at before the message was sent to the exception destination.
JMS_IBM_EXCEPTIONREASON	A reason code indicating the reason why the message was sent to the exception destination.
JMS_IBM_EXCEPTIONTIMESTAMP	The time when the message was sent to the exception destination.
JMS_IBM_FEEDBACK	A code that indicates the nature of a report message.
JMS_IBM_FORMAT	The nature of the application data in the message.
JMS_IBM_LAST_MSG_IN_GROUP	Indicate whether the message is the last message in a message group.
JMS_IBM_MSGTYPE	The type of the message.
JMS_IBM_PUTAPPLTYPE	The type of application that sent the message.
JMS_IBM_PUTDATE	The date when the message was sent.
JMS_IBM_PUTTIME	The time when the message was sent.
JMS_IBM_REPORT_COA	Request confirm on arrival report messages, specifying how much application data from the original message must be included in a report message.
JMS_IBM_REPORT_COD	Request confirm on delivery report messages, specifying how much application data from the original message must be included in a report message.
JMS_IBM_REPORT_DISCARD_MSG	Request that the message is discarded if it cannot be delivered to its intended destination.
JMS_IBM_REPORT_EXCEPTION	Request exception report messages, specifying how much application data from the original message must be included in a report message.

Table 42. Properties of Message (continued)

Name of property	Description
JMS_IBM_REPORT_EXPIRATION	Request expiration report messages, specifying how much application data from the original message must be included in a report message.
JMS_IBM_REPORT_NAN	Request negative action notification report messages.
JMS_IBM_REPORT_PAN	Request positive action notification report messages.
JMS_IBM_REPORT_PASS_CORREL_ID	Request that the correlation identifier of any report or reply message is the same as that of the original message.
JMS_IBM_REPORT_PASS_MSG_ID	Request that the message identifier of any report or reply message is the same as that of the original message.
JMS_IBM_RETAIN	Setting this property indicates to the queue manager to treat a message as Retained Publication.
JMS_IBM_SYSTEM_MESSAGEID	An identifier that identifies the message uniquely within the service integration bus. This property is read-only.
"JMS_TOG_ARM_Correlator" on page 427	The Open Group Application Response Measurement Correlator property, set on a message. Associates a unique Id with the application data in the message.
JMSX_APPID	The name of the application that sent the message.
JMSX_DELIVERY_COUNT	The number of attempts to deliver the message.
JMSX_GROUPID	The identifier of the message group to which the message belongs.
JMSX_GROUPSEQ	The sequence number of the message within a message group.
JMSX_USERID	The user identifier associated with the application that sent the message.

JMS_IBM_MQMD* properties

IBM Message Service Client for C/C++ enables client applications to read/write MQMD fields using APIs. It also allows access to MQ message data. By default access to MQMD is disabled and must be enabled explicitly by the application using Destination properties XMSC_WMQ_MQMD_WRITE_ENABLED and XMSC_WMQ_MQMD_READ_ENABLED. These two properties are independent of each other.

All MQMD fields except StrucId and Version are exposed as additional Message object properties and are prefixed JMS_IBM_MQMD.

JMS_IBM_MQMD* properties take higher precedence over other properties like JMS_IBM* described in the above table.

Sending messages

All MQMD fields except StrucId and Version are represented. These properties refer only to the MQMD fields; where a property occurs both in the MQMD and in the MQRFH2 header, the version in the MQRFH2 is not set or extracted. Any of these properties can be set, except JMS_IBM_MQMD_BackoutCount. Any value set for JMS_IBM_MQMD_BackoutCount is ignored.

If a property has a maximum length and you supply a value that is too long, the value is truncated.

For certain properties, you must also set the WMQ_MQMD_MESSAGE_CONTEXT property on the Destination object. The application must be running with appropriate context authority for this property to take effect. If you do not set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value, the property value is ignored. If you set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value but you do not have sufficient context authority for the queue manager, an exception is issued. Properties requiring specific values of WMQ_MQMD_MESSAGE_CONTEXT are as follows.

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_IDENTITY_CONTEXT or WMQ_MDCTX_SET_ALL_CONTEXT:

- JMS_IBM_MQMD_UserIdentifier
- JMS_IBM_MQMD_AccountingToken
- JMS_IBM_MQMD_ApplIdentityData

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_ALL_CONTEXT:

- JMS_IBM_MQMD_PutApplType
- JMS_IBM_MQMD_PutApplName
- JMS_IBM_MQMD_PutDate
- JMS_IBM_MQMD_PutTime
- JMS_IBM_MQMD_ApplOriginData

Receiving messages

All these properties are available on a received message if WMQ_MQMD_READ_ENABLED property is set to true, irrespective of the actual properties the producing application has set. An application cannot modify the properties of a received message unless all properties are cleared first, according to the JMS specification. The received message can be forwarded without modifying the properties.

Note: If your application receives a message from a destination with WMQ_MQMD_READ_ENABLED property set to true, and forwards it to a destination with WMQ_MQMD_WRITE_ENABLED set to true, this results in all the MQMD field values of the received message being copied into the forwarded message. Table of properties

Table 43. Properties of the Message object representing the MQMD fields

Property	Description	Type
JMS_IBM_MQMD_REPORT	Options for report messages	xmsINT
JMS_IBM_MQMD_MSGTYPE	Message type	xmsINT
JMS_IBM_MQMD_EXPIRY	message lifetime	xmsINT
JMS_IBM_MQMD_FEEDBACK	Feedback or reason code	xmsINT
JMS_IBM_MQMD_ENCODING	Numeric encoding of message data	xmsINT
JMS_IBM_MQMD_CODEDCHARSETID	Character set identifier of message data	xmsINT

Table 43. Properties of the Message object representing the MQMD fields (continued)

Property	Description	Type
JMS_IBM_MQMD_FORMAT	Format name of message data	String
JMS_IBM_MQMD_PRIORITY **Note:** If you assign a value to JMS_IBM_MQMD_PRIORITY that is not within the range 0-9, this violates the JMS specification.	Message priority	xmsINT
JMS_IBM_MQMD_PERSISTENCE	Message persistence	xmsINT
JMS_IBM_MQMD_MSGID **Note:** The JMS specification states that the message ID must be set by the JMS provider and that it must either be unique or null. If you assign a value to JMS_IBM_MQMD_MSGID, this value is copied to the JMSMessageID. Thus it is not set by the JMS provider and might not be unique: this violates the JMS specification.	Message identifier	Byte Array **Note:** The use of byte array properties on a message violates the JMS specification.
JMS_IBM_MQMD_CORRELID **Note:** If you assign a value to JMS_IBM_MQMD_CORRELID that starts with the string 'ID:', this violates the JMS specification.	Correlation identifier	Byte Array **Note:** The use of byte array properties on a message violates the JMS specification.
JMS_IBM_MQMD_BACKOUTCOUNT	Backout counter	xmsINT
JMS_IBM_MQMD_REPLYTOQ	Name of reply queue	String
JMS_IBM_MQMD_REPLYTOQMGR	Name of reply queue manager	String
JMS_IBM_MQMD_USERIDENTIFIER	User identifier	String
JMS_IBM_MQMD_ACCOUNTINGTOKEN	Accounting token	Byte Array **Note:** The use of byte array properties on a message violates the JMS specification.
JMS_IBM_MQMD_APPLIDENTITYDATA	Application data relating to identity	String
JMS_IBM_MQMD_PUTAPPLTYPE	Type of application that put the message	xmsINT
JMS_IBM_MQMD_PUTAPPLNAME	Name of the application that put the message	String
JMS_IBM_MQMD_PUTDATE	Date when message was put	String
JMS_IBM_MQMD_PUTTIME	Time when message was put	String
JMS_IBM_MQMD_APPLORIGINDATA	Application data relating to origin	String
JMS_IBM_MQMD_GROUPID	Group identifier	Byte Array **Note:** The use of byte array properties on a message violates the JMS specification.
JMS_IBM_MQMD_MSGSEQNUMBER	Sequence number of local message within group	xmsINT
JMS_IBM_MQMD_OFFSET	Offset of data in physical message from start of logical message	xmsINT
JMS_IBM_MQMD_MSGFLAGS	Message flags	xmsINT

Table 43. Properties of the Message object representing the MQMD fields (continued)

Property	Description	Type
JMS_IBM_MQMD_ORIGINALLENGTH	Length of original message	xmsINT

For further details on MQMD please refer IBM MQ v7.0 Application Programming Reference.

Examples

This example results in a message being put to a queue or topic with MQMD.UserIdentifier set to "JoeBloggs".

```
// Create a ConnectionFactory, connection, session, producer, message
// ...

// Create a destination
// ...

// Enable MQMD write
dest.setBooleanProperty(XMSC_WMQ_MQMD_WRITE_ENABLED, XMSC_WMQ_MQMD_WRITE_ENABLED_YES);

// Optionally, set a message context if applicable for this MD field
dest.setIntProperty(XMSC_WMQ_MQMD_MESSAGE_CONTEXT,
  XMSC_WMQ_MDCTX_SET_IDENTITY_CONTEXT);

// On the message, set property to provide custom UserId
msg.setStringProperty(JMS_IBM_MQMD_USERIDENTIFIER, "JoeBloggs");

// Send the message
// ...
```

It is necessary to set XMSC_WMQ_MQMD_MESSAGE_CONTEXT before setting JMS_IBM_MQMD_USERIDENTIFIER. For more information about the use of XMSC_WMQ_MQMD_MESSAGE_CONTEXT, see Message object properties.

Similarly, you can extract the contents of the MQMD fields by setting XMSC_WMQ_MQMD_READ_ENABLED to true before receiving a message and then using the get methods of the message, such as getStringProperty. Any properties received are read-only.

This example results in the value field holding the value of the MQMD.ApplIdentityData field of a message got from a queue or a topic.

```
// Create a ConnectionFactory, connection, session, consumer
// ...

// Create a destination
// ...

// Enable MQMD read
dest.setBooleanProperty(XMSC_WMQ_MQMD_READ_ENABLED, XMSC_WMQ_MQMD_READ_ENABLED_YES);

// Receive a message
// ...

// Get desired MQMD field value using a property
String value = rcvMsg.getStringProperty(JMS_IBM_MQMD_APPLIDENTITYDATA);
```

Properties of MessageConsumer

An overview of the properties of the MessageConsumer object, with links to more detailed reference information.

Table 44. Properties of MessageConsumer

Name of property	Description
XMSC_CLIENT_CCSID	The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer.

Properties of MessageProducer

An overview of the properties of the MessageProducer object, with links to more detailed reference information.

Table 45. Properties of MessageProducer

Name of property	Description
XMSC_CLIENT_CCSID	The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer.

Properties of Session

An overview of the properties of the Session object, with links to more detailed reference information.

Table 46. Properties of Session

Name of property	Description
XMSC_CLIENT_CCSID	The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer.

Property definitions

This topic provides a definition of each object property.

Each property definition includes the following information:

- The data type of the property
- The types of object that have the property
- For a property of Destination, the name that can be used in a uniform resource identifier (URI)
- A more detailed description of the property
- The valid values of the property
- The default value of the property

Properties whose names commence with one of the following prefixes are relevant only for the specified type of connection:

XMSC_WMQ
The properties are relevant only when an application connects to a IBM MQ queue manager. The names of the properties are defined as named constants in the header file `xmsc_wmq.h`.

XMSC_WPM
The properties are relevant only when an application connects to a WebSphere service integration bus. The names of the properties are defined as named constants in the header file `xmsc_wpm.h`.

Unless stated otherwise in their definitions, the remaining properties are relevant for all types of connection. The names of the properties are defined as named constants in the header file `xmsc.h`. Properties whose names commence with the prefix JMSX are JMS defined properties of a message, and properties whose names commence with the prefix JMS_IBM are IBM defined properties of a message. For more information about the properties of messages, see "Properties of an XMS message" on page 92.

Unless stated otherwise in its definition, each property is relevant in both the point-to-point and publish/subscribe domains.

An application can get and set the value of any property, unless the property is designated as read-only.

The following properties are defined:
"JMS_IBM_CHARACTER_SET" on page 417
"JMS_IBM_ENCODING" on page 418
"JMS_IBM_EXCEPTIONMESSAGE" on page 419
"JMS_IBM_EXCEPTIONPROBLEMDESTINATION" on page 419
"JMS_IBM_EXCEPTIONREASON" on page 419
"JMS_IBM_EXCEPTIONTIMESTAMP" on page 419
"JMS_IBM_FEEDBACK" on page 420
"JMS_IBM_FORMAT" on page 420
"JMS_IBM_LAST_MSG_IN_GROUP" on page 420
"JMS_IBM_MSGTYPE" on page 421
"JMS_IBM_PUTAPPLTYPE" on page 421
"JMS_IBM_PUTDATE" on page 421
"JMS_IBM_PUTTIME" on page 422
"JMS_IBM_REPORT_COA" on page 422
"JMS_IBM_REPORT_COD" on page 423
"JMS_IBM_REPORT_DISCARD_MSG" on page 423
"JMS_IBM_REPORT_EXCEPTION" on page 423
"JMS_IBM_REPORT_EXPIRATION" on page 424
"JMS_IBM_REPORT_NAN" on page 425
"JMS_IBM_REPORT_PAN" on page 425
"JMS_IBM_REPORT_PASS_CORREL_ID" on page 425
"JMS_IBM_REPORT_PASS_MSG_ID" on page 426
"JMS_IBM_SYSTEM_MESSAGEID" on page 427
"JMSX_APPID" on page 427
"JMSX_DELIVERY_COUNT" on page 428
"JMSX_GROUPID" on page 428
"JMSX_GROUPSEQ" on page 428

"JMSX_USERID" on page 429
"XMSC_CLIENT_CCSID" on page 429
"XMSC_CLIENT_ID" on page 430
"XMSC_CONNECTION_TYPE" on page 430
"XMSC_DELIVERY_MODE" on page 431
"XMSC_IC_PROVIDER_URL" on page 432
"XMSC_IC_SECURITY_AUTHENTICATION" on page 432
"XMSC_IC_SECURITY_CREDENTIALS" on page 432
"XMSC_IC_SECURITY_PRINCIPAL" on page 432
"XMSC_IC_SECURITY_PROTOCOL" on page 433
"XMSC_IC_URL" on page 433
"XMSC_JMS_MAJOR_VERSION" on page 433
"XMSC_JMS_MINOR_VERSION" on page 433
"XMSC_JMS_VERSION" on page 433
"XMSC_MAJOR_VERSION" on page 434
"XMSC_MINOR_VERSION" on page 434
"XMSC_PASSWORD" on page 434
"XMSC_PRIORITY" on page 434
"XMSC_PROVIDER_NAME" on page 435
"XMSC_TIME_TO_LIVE" on page 437
"XMSC_USERID" on page 437
"XMSC_VERSION" on page 437
"XMSC_WMQ_CCSID" on page 439
"XMSC_WMQ_CHANNEL" on page 440
"XMSC_WMQ_CONNECTION_MODE" on page 441
"XMSC_WMQ_DUR_SUBQ" on page 442
"XMSC_WMQ_ENCODING" on page 443
"XMSC_WMQ_FAIL_IF_QUIESCE" on page 444
"XMSC_WMQ_HOST_NAME" on page 450
"XMSC_WMQ_LOCAL_ADDRESS" on page 450
"XMSC_WMQ_MESSAGE_SELECTION" on page 451
"XMSC_WMQ_MSG_BATCH_SIZE" on page 451
"XMSC_WMQ_POLLING_INTERVAL" on page 452
"XMSC_WMQ_PORT" on page 452
"XMSC_WMQ_PUB_ACK_INTERVAL" on page 454
"XMSC_WMQ_QMGR_CCSID" on page 454
"XMSC_WMQ_QUEUE_MANAGER" on page 454
"XMSC_WMQ_RECEIVE_EXIT" on page 455
"XMSC_WMQ_RECEIVE_EXIT_INIT" on page 455
"XMSC_WMQ_SECURITY_EXIT" on page 456
"XMSC_WMQ_SECURITY_EXIT_INIT" on page 456
"XMSC_WMQ_SEND_EXIT" on page 456
"XMSC_WMQ_SEND_EXIT_INIT" on page 457
"XMSC_WMQ_SYNCPOINT_ALL_GETS" on page 463
"XMSC_WMQ_TARGET_CLIENT" on page 464
"XMSC_WMQ_TEMP_Q_PREFIX" on page 464
"XMSC_WMQ_TEMPORARY_MODEL" on page 465

"XMSC_WPM_BUS_NAME" on page 466
"XMSC_WPM_CONNECTION_PROTOCOL" on page 466
"XMSC_WPM_CONNECTION_PROXIMITY" on page 467
"XMSC_WPM_DUR_SUB_HOME" on page 467
"XMSC_WPM_HOST_NAME" on page 467
"XMSC_WPM_LOCAL_ADDRESS" on page 468
"XMSC_WPM_ME_NAME" on page 469
"XMSC_WPM_NON_PERSISTENT_MAP" on page 469
"XMSC_WPM_PERSISTENT_MAP" on page 469
"XMSC_WPM_PORT" on page 470
"XMSC_WPM_PROVIDER_ENDPOINTS" on page 470
"XMSC_WPM_TARGET_GROUP" on page 474
"XMSC_WPM_TARGET_SIGNIFICANCE" on page 474
"XMSC_WPM_TARGET_TRANSPORT_CHAIN" on page 475
"XMSC_WPM_TARGET_TYPE" on page 475
"XMSC_WPM_TEMP_Q_PREFIX" on page 476
"XMSC_WPM_TEMP_TOPIC_PREFIX" on page 476
"XMSC_WPM_TOPIC_SPACE" on page 476

JMS_IBM_ArmCorrelator

Data type:
String

Property of:
Message

The Open Group Application Response Measurement Correlator property, set on a message. This IBM-defined property associates a unique Id with the application data in the message. Use JMS_TOG_ARM_Correlator in preference to this property.

JMS_IBM_ArmCorrelator is a synonym of JMS_TOG_ARM_Correlator. This property is available for compatibility with some existing JMS programs.

This property can be set by using the xmsSetStringProperty method:

```
xmsSetStringProperty(xmsHMsg, JMS_IBM_ArmCorrelator, "ARM_Correlator",
sizeof("ARM_Correlator"), xmsHError);
```

By default, the property is not set.

The value for this property can be obtained using the GetStringProperty method.

This property is not valid for Real Time Transport.

JMS_IBM_CHARACTER_SET

Data type:
xmsINT

Property of:
Message

The identifier (CCSID) of the coded character set, or code page, that the strings of character data in the body of the message will be in when the XMS client forwards the message to its intended destination. In XMS this property has a numeric value and maps to CCSID. However, this property is based on a JMS property so has a

string type value and maps to the Java character set that represents this numeric CCSID. This property overrides any CCSID specified for the destination by the XMSC_WMQ_CCSID property.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_ENCODING

Data type:
xmsINT

Property of:
Message

How numerical data in the body of the message will be represented when the XMS client forwards the message to its intended destination. This property overrides any encoding specified for the destination by the XMSC_WMQ_ENCODING property. The property specifies the representation of binary integers, packed decimal integers, and floating point numbers.

The valid values of the property are the same as the values that can be specified in the *Encoding* field of a message descriptor. For more information about the *Encoding* field, see the *IBM MQ Application Programming Reference*.

An application can use the following named constants to set the property:

Named constant	Meaning
MQENC_INTEGER_NORMAL	Normal integer encoding
MQENC_INTEGER_REVERSED	Reversed integer encoding
MQENC_DECIMAL_NORMAL	Normal packed decimal encoding
MQENC_DECIMAL_REVERSED	Reversed packed decimal encoding
MQENC_FLOAT_IEEE_NORMAL	Normal IEEE floating point encoding
MQENC_FLOAT_IEEE_REVERSED	Reversed IEEE floating point encoding
MQENC_FLOAT_S390	zSeries (System/390®) architecture floating point encoding
MQENC_NATIVE	Native machine encoding

To form a value for the property, the application can add together three of these constants as follows:

- A constant whose name commences with MQENC_INTEGER, to specify the representation of binary integers
- A constant whose name commences with MQENC_DECIMAL, to specify the representation of packed decimal integers
- A constant whose name commences with MQENC_FLOAT, to specify the representation of floating point numbers

Alternatively, the application can set the property to MQENC_NATIVE, whose value is environment dependent.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_EXCEPTIONMESSAGE

Data type:
String

Property of:
Message

Text that describes why the message was sent to the exception destination. This property is read-only.

This property is relevant only when an application connects to a service integration bus and receives a message from an exception destination.

JMS_IBM_EXCEPTIONPROBLEMDESTINATION

Data type:
String

Property of:
Message

The name of the destination that the message was at before the message was sent to the exception destination.

This property is relevant only when an application connects to a service integration bus and receives a message from an exception destination.

JMS_IBM_EXCEPTIONREASON

Data type:
xmsINT

Property of:
Message

A reason code indicating the reason why the message was sent to the exception destination.

For a list of all possible reason codes, see the definition of the com.ibm.websphere.sib.SIRCConstants class in the documentation generated by the Javadoc tool, as supplied with WebSphere Application Server.

This property is relevant only when an application connects to a service integration bus and receives a message from an exception destination.

JMS_IBM_EXCEPTIONTIMESTAMP

Data type:
xmsLONG

Property of:
Message

The time when the message was sent to the exception destination.

The time is expressed in milliseconds since 00:00:00 GMT on the 1 January 1970.

This property is relevant only when an application connects to a service integration bus and receives a message from an exception destination.

JMS_IBM_FEEDBACK

Data type:
xmsINT

Property of:
Message

A code that indicates the nature of a report message.

The valid values of the property are the feedback codes and reason codes that can be specified in the *Feedback* field of a message descriptor. For more information about the *Feedback* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

JMS_IBM_FORMAT

Data type:
String

Property of:
Message

The nature of the application data in the message.

The valid values of the property are the same as the values that can be specified in the *Format* field of a message descriptor. For more information about the *Format* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_LAST_MSG_IN_GROUP

Data type:
xmsBOOL

Property of:
Message

Indicate whether the message is the last message in a message group.

Set the property to xmsTRUE if the message is the last message in a message group. Otherwise, set the property to xmsFALSE, or do not set the property. By default, the property is not set.

The value xmsTRUE corresponds to the status flag MQMF_LAST_MSG_IN_GROUP, which can be specified in the *MsgFlags* field of a message descriptor. For more information about this flag, see the *IBM MQ Application Programming Reference*.

This property is ignored in the publish/subscribe domain and is not relevant when an application connects to a service integration bus.

JMS_IBM_MSGTYPE

Data type:
xmsINT

Property of:
Message

The type of the message.

The valid values of the property are as follows:

Valid value	Meaning
MQMT_DATAGRAM	The message is one that does not require a reply.
MQMT_REQUEST	The message is one that requires a reply.
MQMT_REPLY	The message is a reply message.
MQMT_REPORT	The message is a report message.

These values correspond to the message types that can be specified in the *MsgType* field of a message descriptor. For more information about the *MsgType* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_PUTAPPLTYPE

Data type:
xmsINT

Property of:
Message

The type of application that sent the message.

The valid values of the property are the application types that can be specified in the *PutApplType* field of a message descriptor. For more information about the *PutApplType* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_PUTDATE

Data type:
String

Property of:
Message

The date when the message was sent.

The valid values of the property are the same as the values that can be specified in the *PutDate* field of a message descriptor. For more information about the *PutDate* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_PUTTIME

Data type:
String

Property of:
Message

The time when the message was sent.

The valid values of the property are the same as the values that can be specified in the *PutTime* field of a message descriptor. For more information about the *PutTime* field, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

This property is not relevant when an application connects to a service integration bus.

JMS_IBM_REPORT_COA

Data type:
xmsINT

Property of:
Message

Request confirm on arrival report messages, specifying how much application data from the original message must be included in a report message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_COA	Request confirm on arrival report messages, with no application data from the original message included in a report message.
MQRO_COA_WITH_DATA	Request confirm on arrival report messages, with the first 100 bytes of application data from the original message included in a report message.
MQRO_COA_WITH_FULL_DATA	Request confirm on arrival report messages, with all the application data from the original message included in a report message.

These values correspond to report options that can be specified in the *Report* field of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

JMS_IBM_REPORT_COD

Data type:
xmsINT

Property of:
Message

Request confirm on delivery report messages, specifying how much application data from the original message must be included in a report message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_COD	Request confirm on delivery report messages, with no application data from the original message included in a report message.
MQRO_COD_WITH_DATA	Request confirm on delivery report messages, with the first 100 bytes of application data from the original message included in a report message.
MQRO_COD_WITH_FULL_DATA	Request confirm on delivery report messages, with all the application data from the original message included in a report message.

These values correspond to report options that can be specified in the *Report* field of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

JMS_IBM_REPORT_DISCARD_MSG

Data type:
xmsINT

Property of:
Message

Request that the message is discarded if it cannot be delivered to its intended destination.

Set the property to MQRO_DISCARD_MSG to request that the message is discarded if it cannot be delivered to its intended destination. If you require the message to be put on a dead letter queue instead, or sent to an exception destination, do not set the property. By default, the property is not set.

The value MQRO_DISCARD_MSG corresponds to a report option that can be specified in the *Report* field of a message descriptor. For more information about this option, see the *IBM MQ Application Programming Reference*.

JMS_IBM_REPORT_EXCEPTION

Data type:
xmsINT

Property of:
Message

Request exception report messages, specifying how much application data from the original message must be included in a report message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_EXCEPTION	Request exception report messages, with no application data from the original message included in a report message.
MQRO_EXCEPTION_WITH_DATA	Request exception report messages, with the first 100 bytes of application data from the original message included in a report message.
MQRO_EXCEPTION_WITH_FULL_DATA	Request exception report messages, with all the application data from the original message included in a report message.

These values correspond to report options that can be specified in the *Report* field of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*.

By default, the property is not set.

JMS_IBM_REPORT_EXPIRATION

Data type:
xmsINT

Property of:
Message

Request expiration report messages, specifying how much application data from the original message must be included in a report message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_EXPIRATION	Request expiration report messages, with no application data from the original message included in a report message.
MQRO_EXPIRATION_WITH_DATA	Request expiration report messages, with the first 100 bytes of application data from the original message included in a report message.
MQRO_EXPIRATION_WITH_FULL_DATA	Request expiration report messages, with all the application data from the original message included in a report message.

These values correspond to report options that can be specified in the *Report* field

of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*. By

default, the property is not set.

JMS_IBM_REPORT_NAN

Data type:
xmsINT

Property of:
Message

Request negative action notification report messages.

Set the property to MQRO_NAN to request negative action notification report messages. If you do not require negative action notification report messages, do not set the property. By default, the property is not set.

The value MQRO_NAN corresponds to a report option that can be specified in the *Report* field of a message descriptor. For more information about this option, see the *IBM MQ Application Programming Reference*.

JMS_IBM_REPORT_PAN

Data type:
xmsINT

Property of:
Message

Request positive action notification report messages.

Set the property to MQRO_PAN to request positive action notification report messages. If you do not require positive action notification report messages, do not set the property. By default, the property is not set.

The value MQRO_PAN corresponds to a report option that can be specified in the *Report* field of a message descriptor. For more information about this option, see the *IBM MQ Application Programming Reference*.

JMS_IBM_REPORT_PASS_CORREL_ID

Data type:
xmsINT

Property of:
Message

Request that the correlation identifier of any report or reply message is the same as that of the original message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_PASS_CORREL_ID	Request that the correlation identifier of any report or reply message is the same as that of the original message.

Valid value	Meaning
MQRO_COPY_MSG_ID_TO_CORREL_ID	Request that the correlation identifier of any report or reply message is the same as the message identifier of the original message.

These values correspond to report options that can be specified in the *Report* field of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*.

The default value of the property is MQRO_COPY_MSG_ID_TO_CORREL_ID.

JMS_IBM_REPORT_PASS_MSG_ID

Data type:
xmsINT

Property of:
Message

Request that the message identifier of any report or reply message is the same as that of the original message.

The valid values of the property are as follows:

Valid value	Meaning
MQRO_PASS_MSG_ID	Request that the message identifier of any report or reply message is the same as that of the original message.
MQRO_NEW_MSG_ID	Request that a new message identifier is generated for each report or reply message.

These values correspond to report options that can be specified in the *Report* field of a message descriptor. For more information about these options, see the *IBM MQ Application Programming Reference*.

The default value of the property is MQRO_NEW_MSG_ID.

JMS_IBM_RETAIN

Data type:
xmsINT

Property of:
Message

Setting this property indicates to the queue manager to treat a message as Retained Publication. When a subscriber receives messages from topics, it may receive additional messages immediately after subscribing, beyond those that would have been received in previous releases. These are the optional retained publication(s) for the topic(s) subscribed. For each topic matching the subscription, if there is a retained publication it will be made available for delivery to the subscribing message consumer.

RETAIN_PUBLICATION is the only valid value for this property. By default this property is not set.

Note: This property is relevant only in publish/subscribe domain only

JMS_IBM_SYSTEM_MESSAGEID

Data type:
String

Property of:
Message

An identifier that identifies the message uniquely within the service integration bus. This property is read-only.

This property is relevant only when an application connects to a service integration bus.

JMS_TOG_ARM_Correlator

Data type:
String

Property of:
Message

The Open Group Application Response Measurement Correlator property, set on a message. Associates a unique Id with the application data in the message.

This is a JMS property and is a synonym of JMS_IBM_ArmCorrelator.

Use JMS_TOG_ARM_Correlator in preference to JMS_IBM_ArmCorrelator. JMS_IBM_ArmCorrelator is available for compatibility with some existing JMS programs.

This property can be set by using the xmsSetStringProperty method:

```
xmsSetStringProperty(xmsHMsg, JMS_TOG_ARM_Correlator, “ARM_Correlator”,
sizeof(“ARM_Correlator”), xmsHError);
```

By default, the property is not set.

The value for this property can be obtained using the GetStringProperty method.

This property is not valid for Real Time Transport.

JMSX_APPID

Data type:
String

Property of:
Message

The name of the application that sent the message.

This property is the JMS defined property with the JMS name JMSXAppID. For more information about the property, see the *Java Message Service Specification, Version 1.1*.

By default, the property is not set.

JMSX_DELIVERY_COUNT

Data type:
xmsINT

Property of:
Message

The number of attempts to deliver the message.

This property is the JMS defined property with the JMS name JMSXDeliveryCount. For more information about the property, see the *Java Message Service Specification, Version 1.1*.

By default, the property is not set.

JMSX_GROUPID

Data type:
String

Property of:
Message

The identifier of the message group to which the message belongs.

This property is the JMS defined property with the JMS name JMSXGroupID. For more information about the property, see the *Java Message Service Specification, Version 1.1*.

By default, the property is not set.

JMSX_GROUPSEQ

Data type:
xmsINT

Property of:
Message

The sequence number of the message within a message group.

This property is the JMS defined property with the JMS name JMSXGroupSeq. For more information about the property, see the *Java Message Service Specification, Version 1.1*.

By default, the property is not set.

JMSX_USERID

Data type:
String

Property of:
Message

The user identifier associated with the application that sent the message.

This property is the JMS defined property with the JMS name JMSXUserID. For more information about the property, see the *Java Message Service Specification, Version 1.1*.

By default, the property is not set.

XMSC_ASYNC_EXCEPTIONS

Data type:
xmsINT

Property of:
ConnectionFactory

This property determines whether XMS informs an ExceptionListener only when a connection is broken, or when any exception occurs asynchronously to a XMS API call. This applies to all Connections created from this ConnectionFactory that have an ExceptionListener registered.

Valid values for this property are:

XMSC_ASYNC_EXCEPTIONS_ALL
Any exception detected asynchronously, outside the scope of a synchronous API call, and all connection broken exceptions are sent to the ExceptionListener.

XMSC_ASYNC_EXCEPTIONS_CONNECTIONBROKEN
Only exceptions indicating a broken connection are sent to the ExceptionListener. Any other exceptions occurring during asynchronous processing are not reported to the ExceptionListener, and hence the application is not informed of these exceptions.

By default this property is set to XMSC_ASYNC_EXCEPTIONS_ALL.

XMSC_CLIENT_CCSID

Data type:
xmsINT

Property of:
Connection, ConnectionFactory, Session, MessageProducer, and MessageConsumer

The identifier (CCSID) of the coded character set, or code page, used by a connection, session, message producer, or message consumer. This property is used in C and C++ only. For further information, see "Coded character set identifiers" on page 46.

The following named constants are defined for certain Unicode CCSIDs and can be used when setting the property:

Named constant	CCSID
XMSC_CCSID_UTF8	The UTF-8 representation of Unicode data
XMSC_CCSID_UTF16	The UTF-16 representation of Unicode data
XMSC_CCSID_UTF32	The UTF-32 representation of Unicode data

Instead of a CCSID, the property can have one of the following special values:

XMSC_CCSID_PROCESS
The object is using the code page identified by the process CCSID.

XMSC_CCSID_HOST
The object is using the code page identified by the CCSID that is derived from the environment in which the application is running.

XMSC_CCSID_NO_CONVERSION
The character data in messages received by the object is not converted.

For more information about the property, including how it is set, see "Coded character set identifiers" on page 46.

XMSC_CLIENT_ID

Data type:
String

Property of:
ConnectionFactory

The client identifier for a connection.

A client identifier is used only to support durable subscriptions in the publish/subscribe domain, and is ignored in the point-to-point domain. For further information about setting client identifiers, see "ConnectionFactories and Connection objects" on page 26.

XMSC_CONNECTION_TYPE

Data type:
xmsINT

Property of:
ConnectionFactory

The type of messaging server to which an application connects.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_CT_WMQ	A connection to a IBM MQ queue manager.

Valid value	Meaning
XMSC_CT_WPM	A connection to a WebSphere service integration bus.

By default, the property is not set.

XMSC_DELIVERY_MODE

Data type:
xmsINT

Property of:
Destination

Name used in a URI:
persistence (for a IBM MQ destination)
deliveryMode (for a WebSphere default messaging provider destination)

The delivery mode of messages sent to the destination.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_DELIVERY_NOT_PERSISTENT	A message sent to the destination is nonpersistent. The default delivery mode of the message producer, or any delivery mode specified on the Send call, is ignored. If the destination is a IBM MQ queue, the value of the queue attribute *DefPersistence* is also ignored.
XMSC_DELIVERY_PERSISTENT	A message sent to the destination is persistent. The default delivery mode of the message producer, or any delivery mode specified on the Send call, is ignored. If the destination is a IBM MQ queue, the value of the queue attribute *DefPersistence* is also ignored.
XMSC_DELIVERY_AS_APP	A message sent to the destination has the delivery mode specified on the Send call. If the Send call specifies no delivery mode, the default delivery mode of the message producer is used instead. If the destination is a IBM MQ queue, the value of the queue attribute *DefPersistence* is ignored.

Valid value	Meaning
XMSC_DELIVERY_AS_DEST	If the destination is a IBM MQ queue, a message put on the queue has the delivery mode specified by the value of the queue attribute *DefPersistence*. The default delivery mode of the message producer, or any delivery mode specified on the Send call, is ignored. If the destination is not a IBM MQ queue, the meaning is the same as that of XMSC_DELIVERY_AS_APP.

The default value is XMSC_DELIVERY_AS_APP.

XMSC_IC_PROVIDER_URL

Data type:
String

Property of:
InitialContext

Used to locate the JNDI naming directory so that the COS naming service does not need to be on the same machine as the web service.

XMSC_IC_SECURITY_AUTHENTICATION

Data type:
String

Property of:
InitialContext

Based on the Java Context interface SECURITY_AUTHENTICATION. This property is only applicable to the COS naming context.

XMSC_IC_SECURITY_CREDENTIALS

Data type:
String

Property of:
InitialContext

Based on the Java Context interface SECURITY_CREDENTIALS. This property is only applicable to the COS naming context.

XMSC_IC_SECURITY_PRINCIPAL

Data type:
String

Property of:
InitialContext

Based on the Java Context interface SECURITY_PRINCIPAL. This property is only applicable to the COS naming context.

XMSC_IC_SECURITY_PROTOCOL

Data type:
String

Property of:
InitialContext

Based on the Java Context interface SECURITY_PROTOCOL This property is only applicable to the COS naming context.

XMSC_IC_URL

Data type:
String

Property of:
InitialContext

For LDAP and FileSystem contexts, the address of the repository containing administered objects.

For COS naming contexts, the address of the web service that looks up the objects in the directory.

XMSC_JMS_MAJOR_VERSION

Data type:
xmsINT

Property of:
ConnectionMetaData

The major version number of the JMS specification upon which XMS is based. This property is read-only.

XMSC_JMS_MINOR_VERSION

Data type:
xmsINT

Property of:
ConnectionMetaData

The minor version number of the JMS specification upon which XMS is based. This property is read-only.

XMSC_JMS_VERSION

Data type:
String

Property of:
ConnectionMetaData

The version identifier of the JMS specification upon which XMS is based. This property is read-only.

XMSC_MAJOR_VERSION

Data type:
xmsINT

Property of:
ConnectionMetaData

The version number of the XMS client. This property is read-only.

XMSC_MINOR_VERSION

Data type:
xmsINT

Property of:
ConnectionMetaData

The release number of the XMS client. This property is read-only.

XMSC_PASSWORD

Data type:
Byte array

Property of:
ConnectionFactory

A password that can be used to authenticate the application when it attempts to connect to a messaging server. The password is used in conjunction with the XMSC_USERID property.

By default, the property is not set.

If you are connecting to IBM MQ, and you set the XMSC_USERID property of the connection factory, it must match the userid of the logged on user. If you do not set these properties, the queue manager will use the userid of the logged on user by default. If you require further connection-level authentication of individual users you can write a client authentication exit which is configured in IBM MQ. You can learn more about creating a client authentication exit in the Authentication topic in the IBM MQ documentation manual.

XMSC_PRIORITY

Data type:
xmsINT

Property of:
Destination

Name used in a URI:
priority

The priority of messages sent to the destination.

The valid values of the property are as follows:

Valid value	Meaning
An integer in the range 0, the lowest priority, to 9, the highest priority	A message sent to the destination has the specified priority. The default priority of the message producer, or any priority specified on the Send call, is ignored. If the destination is a IBM MQ queue, the value of the queue attribute *DefPriority* is also ignored.

XMSC_PRIORITY_AS_APP	A message sent to the destination has the priority specified on the Send call. If the Send call specifies no priority, the default priority of the message producer is used instead. If the destination is a IBM MQ queue, the value of the queue attribute *DefPriority* is ignored.
XMSC_PRIORITY_AS_DEST	If the destination is a IBM MQ queue, a message put on the queue has the priority specified by the value of the queue attribute *DefPriority*. The default priority of the message producer, or any priority specified on the Send call, is ignored. If the destination is not a IBM MQ queue, the meaning is the same as that of XMSC_PRIORITY_AS_APP.

The default value is XMSC_PRIORITY_AS_APP.

IBM MQ Real-Time Transport and IBM MQ Multicast Transport take no action based upon the priority of a message.

XMSC_PROVIDER_NAME

Data type:
String

Property of:
ConnectionMetaData

The provider of the XMS client. This property is read-only.

XMSC_TIME_TO_LIVE

Data type:
xmsINT

Property of:
Destination

Name used in a URI:
expiry (for a WebSphere MQ destination)
timeToLive (for a WebSphere default messaging provider destination)

The time to live for messages sent to the destination.

The valid values of the property are as follows:

Valid value	Meaning
0	A message sent to the destination never expires.
A positive integer	A message sent to the destination has the specified time to live in milliseconds. The default time to live of the message producer, or any time to live specified on the Send call, is ignored.
XMSC_TIME_TO_LIVE_AS_APP	A message sent to the destination has the time to live specified on the Send call. If the Send call specifies no time to live, the default time to live of the message producer is used instead.

The default value is XMSC_TIME_TO_LIVE_AS_APP.

XMSC_USERID

Data type:
String

Property of:
ConnectionFactory

A user identifier that can be used to authenticate the application when it attempts to connect to a messaging server. The user identifier is used in conjunction with the XMSC_PASSWORD property.

By default, the property is not set.

If you are connecting to IBM MQ, and you set the XMSC_USERID property of the connection factory, it must match the userid of the logged on user. If you do not set these properties, the queue manager will use the userid of the logged on user by default. If you require further connection-level authentication of individual users you can write a client authentication exit which is configured in IBM MQ. You can learn more about creating a client authentication exit in the Authentication topic in the IBM MQ documentation manual.

XMSC_VERSION

Data type:
String

Property of:
ConnectionMetaData

The version identifier of the XMS client. This property is read-only.

XMSC_WMQ_CCSID

Data type:
xmsINT

Property of:
Destination

Name used in a URI:
CCSID

The identifier (CCSID) of the coded character set, or code page, that the strings of character data in the body of a message will be in when the XMS client forwards the message to the destination. If set for an individual message, the JMS_IBM_CHARACTER_SET property overrides the CCSID specified for the destination by this property.

The default value of the property is 1208.

This property is relevant only to messages sent to the destination, not to messages received from the destination.

XMSC_WMQ_CHANNEL

Data type:
String

Property of:
ConnectionFactory

The name of the channel to be used for a connection.

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_CLIENT_RECONNECT_OPTIONS

Data type:
xmsINT

Property of:
ConnectionFactory

This property determines if a connection is reconnectable. The valid values of the property and meaning are as follows:

Table 47. Values for client reconnection

Valid value	Meaning
XMSC_WMQ_CLIENT_RECONNECT_Q_MGR	This option requests that in case of failure a reconnect to exactly the same Queue Manager is required.
XMSC_WMQ_CLIENT_RECONNECT	This option requests that in case of failure a reconnect is made by the system. Application can reconnect to any of the queue managers specified in the connection name list.
XMSC_WMQ_CLIENT_RECONNECT_DISABLE	DBy specifying this option, the application cannot be reconnected. This is the default value.

Table 47. Values for client reconnection (continued)

Valid value	Meaning
XMSC_WMQ_CLIENT_RECONNECT_AS_DEF	The reconnection option is resolved to its default value. The default value is set in CHANNELS stanza of mqclient.ini file. DefRecon=NO|YES|QMGR|DISABLED The DefRecon attribute enable client programs to automatically reconnect, or to disable the automatic reconnection of a client program that has been written to reconnect automatically. The interpretation of the DefRecon options depends on whether an XMSC_WMQ_CLIENT_RECONNECT_OPTIONS is also set in the client program, and what value is set. If the client program sets the XMSC_WMQ_CLIENT_RECONNECT_AS_DEF option on ConnectionFactory, the reconnect value set by DefRecon takes effect. If no reconnect value is set in the program, or by the DefRecon option, the client program is not reconnected automatically. **NO** : Unless overridden by ConnectionFactory Client Reconnect Options, the client is not reconnected automatically. **YES** : Unless overridden by ConnectionFactory Client Reconnect Option, the client reconnects automatically. **QMGR** : Unless overridden by ConnectionFactory Client Reconnect Options, the client reconnects automatically, but only to the same queue manager. This has the same effect as XMSC_WMQ_CLIENT_RECONNECT_Q_MGR option. **DISABLED** : Reconnection is disabled, even if requested by the client program using the ConnectionFactory Client Reconnect Options.

XMSC_WMQ_CONNECTION_MODE

Data type:
xmsINT

Property of:
ConnectionFactory

The mode by which an application connects to a queue manager.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WMQ_CM_BINDINGS	A connection to a queue manager in bindings mode, for optimal performance. This is the default value for C/C++.
XMSC_WMQ_CM_CLIENT	A connection to a queue manager in client mode, to ensure a fully managed stack.
XMSC_WMQ_CM_CLIENT_UNMANAGED	A connection to a queue manager which forces an unmanaged client stack.

XMSC_WMQ_CONNECTION_NAME_LIST

Data type:
String

Property of:
ConnectionFactory

The connection name list is a comma separated list of host/IP port pairs. If port part is omitted, it will assume the default port value as 1414.

Note: If XMSC_WMQ_CONNECTION_NAME_LIST is provided along with the XMSC_WMQ_HOST_NAME and XMSC_WMQ_PORT, in this case XMSC_WMQ_HOST_NAME and XMSC_WMQ_PORT property is ignored and the XMSC_WMQ_CONNECTION_NAME_LIST will be used.

For example, 127.0.0.1(1414), MACH1.ABC.COM(1400)

XMSC_WMQ_DUR_SUBQ

Data type:
String

Property of:
Destination

The name of the subscriber queue for a durable subscriber that is receiving messages from the destination. Only a destination that is a topic can have this property.

The name of the subscriber queue must start with the following characters:
SYSTEM.JMS.D.

If you want all durable subscribers to share the same subscriber queue, specify the complete name of the shared queue. A queue with the specified name must exist before an application can create a durable subscriber.

If you want each durable subscriber to retrieve messages from its own exclusive subscriber queue, specify a queue name that ends with an asterisk (*). Subsequently, when an application creates a durable subscriber, the XMS client creates a dynamic queue for exclusive use by the durable subscriber. The XMS client uses the value of the property to set the contents of the *DynamicQName* field in the object descriptor that is used to create the dynamic queue.

The default value of the property is SYSTEM.JMS.D.SUBSCRIBER.QUEUE, which means that XMS uses the shared queue approach by default.

This property is relevant only in the publish/subscribe domain.

XMSC_WMQ_ENCODING

Data type:
xmsINT

Property of:
Destination

How numerical data in the body of a message will be represented when the XMS client forwards the message to the destination. If set for an individual message, the JMS_IBM_ENCODING property overrides the encoding specified for the destination by this property. The property specifies the representation of binary integers, packed decimal integers, and floating point numbers.

The valid values of the property are the same as the values that can be specified in the *Encoding* field of a message descriptor. For more information about the *Encoding* field, see the *IBM MQ Application Programming Reference*.

An application can use the following named constants to set the property:

Named constant	Meaning
MQENC_INTEGER_NORMAL	Normal integer encoding
MQENC_INTEGER_REVERSED	Reversed integer encoding
MQENC_DECIMAL_NORMAL	Normal packed decimal encoding
MQENC_DECIMAL_REVERSED	Reversed packed decimal encoding
MQENC_FLOAT_IEEE_NORMAL	Normal IEEE floating point encoding
MQENC_FLOAT_IEEE_REVERSED	Reversed IEEE floating point encoding
MQENC_FLOAT_S390	zSeries (System/390) architecture floating point encoding
MQENC_NATIVE	Native machine encoding

To form a value for the property, the application can add together three of these constants as follows:

- A constant whose name commences with MQENC_INTEGER, to specify the representation of binary integers
- A constant whose name commences with MQENC_DECIMAL, to specify the representation of packed decimal integers
- A constant whose name commences with MQENC_FLOAT, to specify the representation of floating point numbers

Alternatively, the application can set the property to MQENC_NATIVE, whose value is environment dependent.

The default value of the property is MQENC_NATIVE.

This property is relevant only to messages sent to the destination, not to messages received from the destination.

XMSC_WMQ_FAIL_IF_QUIESCE

Data type:
xmsINT

Property of:
ConnectionFactory and Destination

Name used in a URI:
failIfQuiesce

Whether calls to certain methods fail if the queue manager to which the application is connected is in a quiescing state.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WMQ_FIQ_YES	Calls to certain methods fail if the queue manager is in a quiescing state. When the application detects that the queue manager is quiescing, the application can complete its immediate task and close the connection, allowing the queue manager to stop.
XMSC_WMQ_FIQ_NO	No method calls fail because the queue manager is in a quiescing state. If you specify this value, the application cannot detect that the queue manager is quiescing. The application might continue to perform operations against the queue manager and therefore prevent the queue manager from stopping.

The default value for a connection factory is XMSC_WMQ_FIQ_YES but, by default, the property is not set for a destination. Setting the property for a destination overrides any value specified by the connection factory property.

For information about the different ways in which a queue manager can be stopped, see the *IBM MQ Documentation*.

XMSC_WMQ_MESSAGE_BODY

Data type:
xmsINT

Property of:
Destination

This property determines whether a XMS application processes the MQRFH2 of a IBM MQ message as part of the message payload (that is, as part of the message body).

Note: When sending messages to a destination, XMSC_WMQ_MESSAGE_BODY property supersedes existing XMS Destination property XMSC_WMQ_TARGET_CLIENT.

Valid values for this property are:

XMSC_WMQ_MESSAGE_BODY_JMS
Receive: The inbound XMS message type and body are determined by the contents of the MQRFH2 (if present) or the MQMD (if there is no MQRFH2) in the received MQ message.

Send: The outbound XMS message body contains a pre-pended and auto-generated MQRFH2 header based on XMS Message properties and header fields.

XMSC_WMQ_MESSAGE_BODY_MQ
Receive: The inbound XMS message type is always ByteMessage, irrespective of the contents of received IBM MQ message or the format field of the received MQMD. The XMS message body is the unaltered message data returned by the underlying messaging provider API call. The character set and encoding of the data in the message body is determined by the CodedCharSetId and Encoding fields of the MQMD. The format of the data in the message body is determined by the Format field of the MQMD.

Send: The outbound XMS message body contains the application payload as-is; and no auto-generated WMQ header is added to the body.

XMSC_WMQ_MESSAGE_BODY_UNSPECIFIED
Receive: The XMS client determines a suitable value for this property. On receive path, this is simply WMQ_MESSAGE_BODY_JMS property value.

Send: The XMS client determines a suitable value for this property. On send path, this is the value of XMSC_WMQ_TARGET_CLIENT property.

By default this property is set to XMSC_WMQ_MESSAGE_BODY_UNSPECIFIED.

Note: This property is not relevant only when an application connects to Service integration bus

XMSC_WMQ_MQMD_MESSAGE_CONTEXT

Data type:
xmsINT

Property of:
Destination

Determines what level of message context is to be set by the XMS application. The application must be running with appropriate context authority for this property to take effect.

The valid values for this property are:

XMSC_WMQ_MDCTX_DEFAULT
For outbound messages, the MQOPEN API call and the MQPMO structure will specify no explicit message context options.

XMSC_WMQ_MDCTX_SET_IDENTITY_CONTEXT
The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_CONTEXT and the MQPMO structure specifies MQPMO_SET_IDENTITY_CONTEXT.

XMSC_WMQ_MDCTX_SET_ALL_CONTEXT
The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT and the MQPMO structure specifies MQPMO_SET_ALL_CONTEXT.

By default this property will be set to XMSC_WMQ_MDCTX_DEFAULT.

Note: This property is not relevant when an application connects to System Integration Bus.

Following properties require XMSC_WMQ_MQMD_MESSAGE_CONTEXT property to be set to XMSC_WMQ_MDCTX_SET_IDENTITY_CONTEXT property value or XMSC_WMQ_MDCTX_SET_ALL_CONTEXT property value when sending a message for in order to have desired effect:

- JMS_IBM_MQMD_USERIDENTIFIER
- JMS_IBM_MQMD_ACCOUNTINGTOKEN
- JMS_IBM_MQMD_APPLIDENTITYDATA

Following properties require XMSC_WMQ_MQMD_MESSAGE_CONTEXT property to be set to XMSC_WMQ_MDCTX_SET_ALL_CONTEXT property value when sending a message for in order to have desired effect:

- JMS_IBM_MQMD_PUTAPPLTYPE
- JMS_IBM_MQMD_PUTAPPLNAME
- JMS_IBM_MQMD_PUTDATE
- JMS_IBM_MQMD_PUTTIME
- JMS_IBM_MQMD_APPLORIGINDATA

For further information about the Message Context, see IBM MQ documentation and IBM MQ Application Programming Reference book.

XMSC_WMQ_MQMD_READ_ENABLED

Data type:
xmsINT

Property of:
Destination

This property determines whether a XMS application can extract the values of MQMD fields or not.

The valid values for this property are:

XMSC_WMQ_READ_ENABLED_NO
When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD.

When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them.

XMSC_WMQ_READ_ENABLED_YES
When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including those that the sender did not set explicitly.

When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including those that the sender did not set explicitly.

By default this property is set to XMSC_WMQ_READ_ENABLED_NO.

XMSC_WMQ_MQMD_WRITE_ENABLED

Data type:
xmsINT

Property of:
Destination

This property determines whether a XMS application can set the values of MQMD fields or not.

The valid values for this property are:

XMSC_WMQ_WRITE_ENABLED_NO
All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure.

XMSC_WMQ_WRITE_ENABLED_YES
JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure.

By default this property is set to XMSC_WMQ_WRITE_ENABLED_NO.

XMSC_WMQ_PUT_ASYNC_ALLOWED

Data type:
xmsINT

Property of:
Destination

This property determines whether message producers are allowed to use asynchronous puts to send messages to this destination.

The valid values for this property are:

XMSC_WMQ _PUT_ASYNC_ALLOWED_AS_DEST
Determine whether asynchronous puts are allowed by referring to the queue or topic definition.

XMSC_WMQ _PUT_ASYNC_ALLOWED_AS_Q_DEF
Determine whether asynchronous puts are allowed by referring to the queue definition.

XMSC_WMQ _PUT_ASYNC_ALLOWED_AS_TOPIC_DEF
Determine whether asynchronous puts are allowed by referring to the topic definition.

XMSC_WMQ _PUT_ASYNC_ALLOWED_DISABLED
Asynchronous puts are not allowed.

XMSC_WMQ _PUT_ASYNC_ALLOWED_ENABLED
Asynchronous puts are allowed.

By default this property is set to XMSC_WMQ _PUT_ASYNC_ALLOWED_AS_DEST.

Note: This property is not relevant when an application is connecting to System Integration Bus.

XMSC_WMQ_READ_AHEAD_ALLOWED

Data type:
xmsINT

Property of:
Destination

This property determines whether message consumers and queue browsers are allowed to use read ahead to get non-persistent, non-transactional messages from this destination into an internal buffer before receiving them.

The valid values for this property are:

XMSC_WMQ_READ_AHEAD_ALLOWED_AS_Q_DEF
Determine whether read ahead is allowed by referring to the queue definition.

XMSC_WMQ_READ_AHEAD_ALLOWED_AS_ TOPIC _DEF
Determine whether read ahead is allowed by referring to the topic definition.

XMSC_WMQ_READ_AHEAD_ALLOWED_AS_DEST
Determine whether read ahead is allowed by referring to the queue or topic definition.

XMSC_WMQ_READ_AHEAD_ALLOWED_DISABLED
Read ahead is not allowed while consuming or browsing messages

XMSC_WMQ_READ_AHEAD_ALLOWED_ENABLED
Read ahead is allowed.

By default this property is set to XMSC_WMQ _READ_AHEAD_ALLOWED_AS_DEST.

XMSC_WMQ_READ_AHEAD_CLOSE_POLICY

Data type:
xmsINT

Property of:
Destination

This property determines, for messages being delivered to an asynchronous message listener, what happens to messages in the internal read ahead buffer when the message consumer is closed.

This property is applicable in specifying closing queue options when consuming messages from a destination and not applicable when sending messages to a destination.

This property will be ignored for Queue Browsers since during browse the messages will still be available in the queues.

The valid values for this property are:

XMSC_WMQ_READ_AHEAD_CLOSE_POLICY_DELIVER_CURRENT
Only the current message listener invocation completes before returning, potentially leaving messages in the internal read ahead buffer, which are then discarded.

XMSC_WMQ_READ_AHEAD_CLOSE_POLICY_DELIVER_ALL
All messages in the internal read ahead buffer are delivered to the application's message listener before returning. Please see **Notes** below.

By default this property is set to XMSC_WMQ
_READ_AHEAD_CLOSE_POLICY_DELIVER_CURRENT.

Notes:

- **Abnormal application termination**

  All the messages in the read ahead buffer will be lost when a XMS application terminates abruptly.

- **Implications on Transactions**

  The read ahead will be disabled when the applications use transaction. So, the application will not be seeing any difference in the behavior when they use transacted sessions.

- **Implications of Session Acknowledgement modes**

  The read ahead will be enabled when the on a non transacted session when the acknowledgement modes are either XMSC_AUTO_ACKNOWLEDGE or XMSC_DUPS_OK_ACKNOWLEDGE. The read ahead will be disabled if the session acknowledgement mode is XMSC_CLIENT_ACKNOWLEDGE irrespective of transacted or non transacted sessions.

- **Implications on Queue Browsers and Queue Browser Selectors**

  The Queue Browsers and Queue Browser Selectors, used in XMS applications, will get the performance advantage from read ahead. Closing the Queue Browser won't impact, since the message is still available in the queue for ay further operations. There will not be any other implication on queue browsers and queue browser selectors apart from performance benefits of read ahead.

- **Consumer close**

  Closing a consumer that has been created with XMSC_WMQ_READ_AHEAD_CLOSE_POLICY_DELIVER_ALL option after stopping the connection might result in loss of messages which have already been streamed.

- **Connection close**

  Closing a connection without explicitly closing a consumer which has been created with XMSC_WMQ_READ_AHEAD_CLOSE_POLICY_DELIVER_ALL option might result in loss of messages which have already been streamed.

For further information about the Read Ahead, see IBM MQ documentation book and IBM MQ Application Programming Reference book.

XMSC_WMQ_RESOLVED_QUEUE_MANAGER_ID

Data type:
String

Property of:
Connection

This property is used to obtain the unique queue manager ID to which it is connected. This property is read-only.

XMSC_WMQ_HOST_NAME

Data type:
String

Property of:
ConnectionFactory

The host name or IP address of the system on which a queue manager resides.

This property is used only when an application connects to a queue manager in client mode. The property is used in conjunction with the XMSC_WMQ_PORT property to identify the queue manager.

The default value of the property is `localhost`.

Related reference:

"Network stack selection mechanism" on page 49
This section describes the network stack selection mechanism when both IPv4 and IPv6 network stacks are enabled on a machine.

XMSC_WMQ_LOCAL_ADDRESS

Data type:
String

Property of:
ConnectionFactory

For a connection to a queue manager, this property specifies the local network interface to be used, or the local port or range of local ports to be used, or both.

The value of the property is a string with the following format:

[*host_name*][(*low_port*)[,*high_port*])]

The meanings of the variables are as follows:

host_name
The host name or IP address of the local network interface to be used for the connection.

Providing this information is necessary only if the system on which the application is running has two or more network interfaces and you need to be able to specify which interface must be used for the connection. If the system has only one network interface, only that interface can be used. If the system has two or more network interfaces and you do not specify which interface must be used, the interface is selected at random.

low_port
The number of the local port to be used for the connection.

If *high_port* is also specified, *low_port* is interpreted the lowest port number in a range of port numbers.

high_port
The highest port number in a range of port numbers. One of the ports in the specified range must be used for the connection.

The maximum length of the string is 48 characters.

Here are some examples of valid values of the property:

JUPITER
9.20.4.98
JUPITER(1000)
9.20.4.98(1000,2000)
(1000)
(1000,2000)
fecc:0:0:a2::2
fecc:0:0:a2::2(1000,2000)

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

Related reference:

"Network stack selection mechanism" on page 49
This section describes the network stack selection mechanism when both IPv4 and IPv6 network stacks are enabled on a machine.

XMSC_WMQ_MESSAGE_SELECTION

Data type:
xmsINT

Property of:
ConnectionFactory

Determines whether message selection is done by the XMS client or by the broker.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WMQ_MSEL_CLIENT	Message selection is done by the XMS client.

The default value is XMSC_WMQ_MSEL_CLIENT.

This property is relevant only in the publish/subscribe domain..

XMSC_WMQ_MSG_BATCH_SIZE

Data type:
xmsINT

Property of:
ConnectionFactory

The maximum number of messages to be retrieved from a queue in one batch when using asynchronous message delivery.

When an application is using asynchronous message delivery, under certain conditions, the XMS client retrieves a batch of messages from a queue before forwarding each message individually to the application. This property specifies the maximum number of messages that can be in the batch.

The value of the property is a positive integer, and the default value is 10. Only consider setting the property to a different value if you have a specific performance problem that you need to address.

If an application is connected to a queue manager over a network, raising the value of this property can reduce network overheads and response times, but increase the amount of memory required to store the messages on the client system. Conversely, lowering the value of this property might increase network overheads and response times, but reduce the amount of memory required to store the messages.

XMSC_WMQ_POLLING_INTERVAL

Data type:
xmsINT

Property of:
ConnectionFactory

If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue.

If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property.

The value of the property is a positive integer. The default value is 5000.

XMSC_WMQ_PORT

Data type:
xmsINT

Property of:
ConnectionFactory

The number of the port on which a queue manager listens for incoming requests.

This property is used only when an application connects to a queue manager in client mode. The property is used in conjunction with the XMSC_WMQ_HOST_NAME property to identify the queue manager.

The default value of the property is XMSC_WMQ_DEFAULT_CLIENT_PORT, or 1414.

XMSC_WMQ_PROVIDER_VERSION

Data type:
String

Property of:
ConnectionFactory

The version, release, modification level and fix pack of the queue manager to which the application intends to connect. Valid values for this property are:

v Unspecified

Or a string in one of the following formats

v V.R.M.F

v V.R.M

v V.R

v V

Where V, R, M and F are integer values greater than or equal to zero.

A value of 7 or greater indicates that this is intended as a IBM MQ Version 7.0 ConnectionFactory for connections to a IBM MQ Version 7.0 queue manager. A value lower than 7 (for example "6.0.2.0"), indicates that it is intended for use with queue managers earlier than Version 7.0. The default value, unspecified, allows connections to any level of queue manager, determining the applicable properties and functionality available based on the queue manager's capabilities.

By default this property is set to “unspecified”.

Note:

v No socket sharing happens if XMSC_WMQ_PROVIDER_VERSION is set to 6. 2.

v Connection will fail if XMSC_WMQ_PROVIDER_VERSION is set to 7 and on the server SHARECNV for the channel has been set 0.

v MQ v7 specific features will be disabled if XMSC_WMQ_PROVIDER_VERSION is set to UNSPECIFIED and SHARECNV is set to 0.

The version of IBM MQ Client also plays major role in whether a XMS client application can use IBM MQ version 7 specific features. The following table describes the behavior.

Note: A system property XMSC_WMQ_OVERRIDEPROVIDERVERSION has been provided to override XMSC_WMQ_PROVIDER_VERSION property. This can be used if you are unable to change connection factory setting.

Table 48. XMS client - Ability to use IBM MQ v7 specific features.

#	XMSC_WMQ_PROVIDER_VERSION	IBM MQ Client Version	IBM MQ v7 features
1	unspecified	7	ON
2	unspecified	6	OFF
3	7	7	ON
4	7	6	Exception
5	6	6	OFF
6	6	7	OFF

XMSC_WMQ_PUB_ACK_INTERVAL

Data type:
xmsINT

Property of:
ConnectionFactory

If you lower the value of this property, the client requests acknowledgements more often, and therefore the performance of the publisher decreases..

The value of the property is a positive integer. The default value is 25.

XMSC_WMQ_QMGR_CCSID

Data type:
xmsINT

Property of:
ConnectionFactory

The identifier (CCSID) of the coded character set, or code page, in which fields of character data defined in the Message Queue Interface (MQI) are exchanged between the XMS client and the IBM MQ client. This property does not apply to the strings of character data in the bodies of messages.

When an XMS application connects to a queue manager in client mode, the XMS client links to the IBM MQ client. The information exchanged between the two clients contains fields of character data that are defined in the MQI. Under normal circumstances, the IBM MQ client assumes that these fields are in the code page of the system on which the clients are running. If the XMS client provides and expects to receive these fields in a different code page, you must set this property to inform the IBM MQ client.

When the IBM MQ client forwards these fields of character data to the queue manager, the data in them must be converted if necessary into the code page used by the queue manager. Similarly, when the WebSphere MQ client receives these fields from the queue manager, the data in them must be converted if necessary into the code page in which the XMS client expects to receive the data. The IBM MQ client uses this property to perform these data conversions.

By default, the property is not set.

Setting this property is equivalent to setting the MQCCSID environment variable for a IBM MQ client that is supporting native IBM MQ client applications. For more information about this environment variable, see *IBM MQ documentation*.

XMSC_WMQ_QUEUE_MANAGER

Data type:
String

Property of:
ConnectionFactory

The name of the queue manager to connect to.

By default, the property is not set.

XMSC_WMQ_RECEIVE_EXIT

Data type:
String

Property of:
ConnectionFactory

Identifies a channel receive exit, or a sequence of channel receive exits, to be run in succession.

The value of the property is a string of one or more items separated by commas, where each item identifies a channel receive exit and has the following format:

libraryName(*entryPointName*)

For more information about the format of the string that identifies an individual channel receive exit, see *IBM MQ Intercommunication*.

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_RECEIVE_EXIT_INIT

Data type:
String

Property of:
ConnectionFactory

The user data that is passed to channel receive exits when they are called.

The value of the property is a string of one or more items of user data separated by commas. By default, the property is not set.

Note the following rules when specifying user data that is passed to a sequence of channel receive exits:

- If the number of items of user data in the string is more than the number of channel receive exits in the sequence, the excess items of user data are ignored.
- If the number of items of user data in the string is less than the number of channel receive exits in the sequence, each unspecified item of user data is set to the empty string.
- Two commas is succession within the string, or a comma at the beginning of the string, also denotes an unspecified item of user data.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SECURITY_EXIT

Data type:
String

Property of:
ConnectionFactory

Identifies a channel security exit.

The value of the property is a string that identifies a channel security exit and has the following format:

libraryName(*entryPointName*)

For more information about the format of the string that identifies a channel security exit, see *IBM MQ Intercommunication*. The maximum length of the string is 128 characters.

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SECURITY_EXIT_INIT

Data type:
String

Property of:
ConnectionFactory

The user data that is passed to a channel security exit when it is called.

The maximum length of the string of user data is 32 characters.

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SEND_EXIT

Data type:
String

Property of:
ConnectionFactory

Identifies a channel send exit, or a sequence of channel send exits, to be run in succession.

The value of the property is a string of one or more items separated by commas, where each item identifies a channel send exit and has the following format:

libraryName(*entryPointName*)

For more information about the format of the string that identifies an individual channel send exit, see *IBM MQ Intercommunication*.

By default, the property is not set.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SEND_EXIT_INIT

Data type:
String

Property of:
ConnectionFactory

The user data that is passed to channel send exits when they are called.

The value of the property is a string of one or more items of user data separated by commas. By default, the property is not set.

The rules for specifying user data that is passed to a sequence of channel send exits are the same as those for specifying user data that is passed to a sequence of channel receive exits. For the rules therefore, see "XMSC_WMQ_RECEIVE_EXIT_INIT" on page 455.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SEND_CHECK_COUNT

Data type:
xmsINT

Property of:
ConnectionFactory

The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted XMS session.

By default this property is set to 0.

XMSC_WMQ_SHARE_CONV_ALLOWED

Data type:
xmsINT

Property of:
ConnectionFactory

Whether a client connection can share its socket with other top-level XMS connections from the same process to the same queue manager, if the channel definitions match. This property is provided to allow complete isolation of Connections in separate sockets if required for application development, maintenance or operational reasons. Setting this property merely indicates to XMS to make the underlying socket shared. It does not indicate how many connections will share a single socket. The number of connections sharing a socket is determined by SHARECONV value which is negotiated between MQI Client and WMQ Server.

An application can set the following named constants to set the property:

- XMSC_WMQ_SHARE_CONV_ALLOWED_DISABLED - Connections will not share a socket.
- XMSC_WMQ_SHARE_CONV_ALLOWED_ENABLED - Connections share a socket.

By default the property is set to XMSC_WMQ_SHARE_CONV_ALLOWED_ENABLED.

This property is relevant only when an application connects to a queue manager in client mode.

XMSC_WMQ_SSL_CERT_STORES

Data type:
String

Property of:
ConnectionFactory

The locations of the servers that hold the certificate revocation lists (CRLs) to be used on an SSL connection to a queue manager.

The value of the property is a list of one or more URLs separated by commas. Each URL has the following format:

```
[user[/password]@]ldap://[serveraddress][:portnum][,...]
```

This format is compatible with, but extended from, the basic MQJMS format.

It is valid to have an empty 'serveraddress'. In this case, XMS assumes that the value is the string "localhost".

An example list is:

```
myuser/mypassword@ldap://server1.mycom.com:389
ldap://server1.mycom.com
ldap://
ldap://:389
```

By default, the property is not set.

XMSC_WMQ_SSL_CIPHER_SPEC

Data type:
String

Property of:
ConnectionFactory

The name of the cipher spec to be used on a secure connection to a queue manager.

The canonical values of this property that apply to XMS are:

- DES_SHA_EXPORT
- DES_SHA_EXPORT1024
- FIPS_WITH_3DES_EDE_CBC_SHA
- FIPS_WITH_DES_CBC_SHA
- NULL_MD5

v NULL_SHA
v RC2_MD5_EXPORT
v RC4_MD5_EXPORT
v RC4_MD5_US
v RC4_SHA_US
v TLS_RSA_WITH_3DES_EDE_CBC_SHA
v TLS_RSA_WITH_AES_128_CBC_SHA
v TLS_RSA_WITH_AES_256_CBC_SHA
v TLS_RSA_WITH_DES_CBC_SHA
v TRIPLE_DES_SHA_US
v ECDHE_ECDSA_3DES_EDE_CBC_SHA256
v ECDHE_ECDSA_AES_128_CBC_SHA256
v ECDHE_ECDSA_AES_128_GCM_SHA256
v ECDHE_ECDSA_AES_256_CBC_SHA384
v ECDHE_ECDSA_AES_256_GCM_SHA384
v ECDHE_ECDSA_NULL_SHA256
v ECDHE_ECDSA_RC4_128_SHA256
v ECDHE_RSA_3DES_EDE_CBC_SHA256
v ECDHE_RSA_AES_128_CBC_SHA256
v ECDHE_RSA_AES_128_GCM_SHA256
v ECDHE_RSA_AES_256_CBC_SHA384
v ECDHE_RSA_AES_256_GCM_SHA384
v ECDHE_RSA_NULL_SHA256
v ECDHE_RSA_RC4_128_SHA256
v TLS_RSA_WITH_AES_128_CBC_SHA256
v TLS_RSA_WITH_AES_128_GCM_SHA256
v TLS_RSA_WITH_AES_256_CBC_SHA256
v TLS_RSA_WITH_AES_256_GCM_SHA384
v TLS_RSA_WITH_NULL_SHA256
v TLS_RSA_WITH_RC4_128_SHA256

For additional information about these values, see *IBM MQ documentation*.

The following example shows how this value is supplied at the MQI:

```
strncpy(pChDef->SSLCipherSpec, "TRIPLE_DES_SHA_US", sizeof(pChDef->SSLCipherSpec));
```

XMS takes a copy of the first 32 bytes of the string in the correct single-byte code page into the SSLCipherSpec field of the channel definition structure, MQCD before calling MQCONNX.

If a value is specified for the XMSC_WMQ_SSL_CIPHER_SPEC property, this value overrides any value that is specified for the XMSC_WMQ_SSL_CIPHER_SUITE property. If neither of these properties has a specified value, the MQCD.SSLCipherSpec field is filled with space characters.

The XMSC_WMQ_SSL_CIPHER_SPEC property is relevant only if the application connects to a queue manager in client mode.

By default, the property is not set.

XMSC_WMQ_SSL_CIPHER_SUITE

Data type:
String

Property of:
ConnectionFactory

The name of the CipherSuite to be used on an SSL or TLS connection to a queue manager. The protocol used in negotiating the secure connection depends on the specified CipherSuite.

This property has the following canonical values:

- SSL_RSA_WITH_DES_CBC_SHA
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_FIPS_WITH_DES_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_AES_128_CBC_SHA
- SSL_RSA_WITH_AES_256_CBC_SHA
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- SSL_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- SSL_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- SSL_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- SSL_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- SSL_ECDHE_ECDSA_WITH_NULL_SHA
- SSL_ECDHE_ECDSA_WITH_RC4_128_SHA
- SSL_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- SSL_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- SSL_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- SSL_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- SSL_ECDHE_RSA_WITH_NULL_SHA
- SSL_ECDHE_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_AES_128_CBC_SHA256
- SSL_RSA_WITH_AES_128_GCM_SHA256
- SSL_RSA_WITH_AES_256_CBC_SHA256
- SSL_RSA_WITH_AES_256_GCM_SHA384
- SSL_RSA_WITH_NULL_SHA256
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
- SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA

This value can be supplied as an alternative to XMSC_WMQ_SSL_CIPHER_SPEC.

If a non-empty value is specified for XMSC_WMQ_SSL_CIPHER_SPEC, this value overrides the setting for XMSC_WMQ_SSL_CIPHER_SUITE. If XMSC_WMQ_SSL_CIPHER_SPEC does not have a value, the value of XMSC_WMQ_SSL_CIPHER_SUITE is used as the cipher suite to be given to GSKit. In this case, the value is mapped on to the equivalent CipherSpec value, as described in "CipherSuite and CipherSpec name mappings for connections to a IBM MQ queue manager" on page 86.

If both XMSC_WMQ_SSL_CIPHER_SPEC and XMSC_WMQ_SSL_CIPHER_SUITE are empty, the field pChDef->SSLCipherSpec is filled with spaces.

By default, the property is not set.

XMSC_WMQ_SSL_ENCRYPTION_POLICY_SUITE_B

Data type:
xmsINT

Property of:
ConnectionFactory

The value of this property determines whether an application can use the Suite B compliant cipher suites. You can enable the Suite B compliance cipher suits by setting this property to one or more of the following values:

- XMSC_WMQ_SUITE_B_NONE
- XMSC_ WMQ _SUITE_B_128_BIT
- XMSC_ WMQ _SUITE_B_192_BIT

Setting the **XMSC_WMQ_SUITE_B_NONE** property to any other value is invalid.

Related reference:

"XMSC_WPM_SSL_ENCRYPTION_POLICY_SUITE_B" on page 472

XMSC_WMQ_SSL_CRYPTO_HW

Data type:
String

Property of:
ConnectionFactory

Configuration details for the cryptographic hardware connected to the client system.

This property has the following canonical values:

- GSK_ACCELERATOR_RAINBOW_CS_OFF
- GSK_ACCELERATOR_RAINBOW_CS_ON
- GSK_ACCELERATOR_NCIPHER_NF_OFF
- GSK_ACCELERATOR_NCIPHER_NF_ON

There is a special format for PKCS11 cryptogragraphic hardware (where DriverPath, TokenLabel and TokenPassword are user-specified strings):

```
GSK_PKCS11=PKCS#11 DriverPath; PKCS#11 TokenLabel;PKCS#11 TokenPassword
```

For additional information about the format of this property, see *IBM MQ Application Programming Reference*.

XMS does not interpret or alter the contents of the string. It simply copies the value supplied, up to a limit of 256 single-byte characters, into the MQSCO.CryptoHardware field.

By default, the property is not set.

XMSC_WMQ_SSL_FIPS_REQUIRED

Data type:
Boolean

Property of:
ConnectionFactory

The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.

This property can have the following values, which translate to the two canonical values for MQSCO.FipsRequired:

Table 49. Table of values for MQSCO.FlipsRequired property

Value	Description	Corresponding value of MQSCO.FipsRequired
xmsFALSE	Any CipherSpec can be used.	MQSSL_FIPS_NO (the default)
xmsTRUE	Only FIPS-certified cryptographic algorithms can be used in the CipherSpec applying to this client connection.	MQSSL_FIPS_YES

XMS copies the relevant value into MQSCO.FipsRequired before calling MQCONNX.

The parameter MQSCO.FipsRequired is only available from IBM MQ version 6. In the case of IBM MQ version 5.3, if this property is set, XMS does not attempt to make the connection to the queue manager, and throws an appropriate exception instead.

XMSC_WMQ_SSL_KEY_REPOSITORY

Data type:
String

Property of:
ConnectionFactory

The location of the key database file in which keys and certificates are stored.

XMS copies the string, up to a limit of 256 single-byte characters, into the MQSCO.KeyRepository field. IBM MQ interprets this string as a filename, including the full path.

By default, the property is not set.

XMSC_WMQ_SSL_KEY_RESETCOUNT

Data type:
xmsINT

Property of:
ConnectionFactory

The KeyResetCount represents the total number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated. The number of bytes includes control information sent by the MCA.

XMS copies the value that you supply for this property into MQSCO.KeyResetCount before calling MQCONNX.

The parameter MQSCO.KeyRestCount is only available from IBM MQ version 6. In the case of IBM MQ version 5.3, if this property is set, XMS does not attempt to make the connection to the queue manager, and throws an appropriate exception instead.

The default value of this property is zero, which means that secret keys are never renegotiated. For further information, see the *IBM MQ Application Programming Reference*.

XMSC_WMQ_SSL_PEER_NAME

Data type:
String

Property of:
ConnectionFactory

The peer name to be used on an SSL connection to a queue manager.

There is no list of canonical values for this property. Instead, you must build this string according to the rules for SSLPEER described in *IBM MQ Using Java* and *IBM MQ documentation*.

An example of a peer name is:

```
"CN=John Smith, O=IBM ,OU=Test , C=GB"
```

XMS copies the string into the correct single-byte code page, and places the correct values into MQCD.SSLPeerNamePtr and MQCD.SSLPeerNameLength before calling MQCONNX.

This property is relevant only if the application connects to a queue manager in client mode.

By default, the property is not set.

XMSC_WMQ_SYNCPOINT_ALL_GETS

Data type:
xmsBOOL

Property of:
ConnectionFactory

Whether all messages must be retrieved from queues within syncpoint control.

The valid values of the property are as follows:

Valid value	Meaning
xmsFALSE	When the circumstances are appropriate, the XMS client can retrieve messages from queues outside of syncpoint control.
xmsTRUE	The XMS client must retrieve all messages from queues within syncpoint control.

The default value is xmsFALSE.

XMSC_WMQ_TARGET_CLIENT

Data type:
xmsINT

Property of:
Destination

Name used in a URI:
targetClient

Whether messages sent to the destination contain an MQRFH2 header.

If an application sends a message containing an MQRFH2 header, the receiving application must be able to handle the header.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WMQ_TARGET_DEST_JMS	Messages sent to the destination contain an MQRFH2 header. Specify this value if the application is sending the messages to another XMS application, a WebSphere JMS application, or a native IBM MQ application that has been designed to handle an MQRFH2 header.
XMSC_WMQ_TARGET_DEST_MQ	Messages sent to the destination do not contain an MQRFH2 header. Specify this value if the application is sending the messages to a native IBM MQ application that has not been designed to handle an MQRFH2 header.

The default value is XMSC_WMQ_TARGET_DEST_JMS.

XMSC_WMQ_TEMP_Q_PREFIX

Data type:
String

Property of:
ConnectionFactory

The prefix used to form the name of the IBM MQ dynamic queue that is created when the application creates an XMS temporary queue.

The rules for forming the prefix are the same as those for forming the contents of the *DynamicQName* field in an object descriptor, but the last non blank character

must be an asterisk(*). If the property is not set, the value used is CSQ.* on z/OS and AMQ.* on the other platforms. By default, the property is not set.

This property is relevant only in the point-to-point domain.

XMSC_WMQ_TEMP_TOPIC_PREFIX

Data type:
String

Property of:
ConnectionFactory, Destination

When creating temporary topics, XMS will generate a topic string of the form "TEMP/TEMPTOPICPREFIX/unique_id", or if this property is left with the default value, just "TEMP/unique_id". Specifying a non-empty value allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.

Any non-null string consisting only of valid characters for a IBM MQ topic string is a valid value for this property.

By default this property is set to "" (empty string).

Note: This property is relevant only in the publish/subscribe domain.

XMSC_WMQ_TEMPORARY_MODEL

Data type:
String

Property of:
ConnectionFactory

The name of the IBM MQ model queue from which a dynamic queue is created when the application creates an XMS temporary queue.

The default value of the property is SYSTEM.DEFAULT.MODEL.QUEUE.

This property is relevant only in the point-to-point domain.

XMSC_WMQ_WILDCARD_FORMAT

Data type:
xmsINT

Property of:
ConnectionFactory, Destination

This property determines which version of wildcard syntax is to be used.

When using Publish/Subscribe with IBM MQ '*' and '?' are treated as wildcards..

The valid values for this property are:

XMSC_WMQ_WILDCARD_TOPIC_ONLY
Recognizes the topic level wildcards only i.e. only '#' and '+' are treated as wildcards.

XMSC_WMQ_WILDCARD_CHAR_ONLY
Recognizes the character wildcards only i.e. '*' and '?' are treated as wildcards.

By default this property is set to XMSC_WMQ_WILDCARD_TOPIC_ONLY.

XMSC_WPM_BUS_NAME

Data type:
String

Property of:
ConnectionFactory and Destination

Name used in a URI:
busName

For a connection factory, the name of the service integration bus that the application connects to or, for a destination, the name of the service integration bus in which the destination exists.

For a destination that is a topic, this property is the name of the service integration bus in which the associated topic space exists. This topic space is specified by the XMSC_WPM_TOPIC_SPACE property.

If the property is not set for a destination, the queue or associated topic space is assumed to exist in the service integration bus to which the application connects.

By default, the property is not set.

XMSC_WPM_CONNECTION_PROTOCOL

Data type:
xmsINT

Property of:
Connection

The communications protocol used for the connection to the messaging engine. This property is read-only.

The possible values of the property are as follows:

Value	Meaning
XMSC_WPM_CP_HTTP	The connection uses HTTP over TCP/IP.
XMSC_WPM_CP_TCP	The connection uses TCP/IP.

XMSC_WPM_CONNECTION_PROXIMITY

Data type:
xmsINT

Property of:
ConnectionFactory

The connection proximity setting for the connection. This property determines how close the messaging engine that the application connects to must be to the bootstrap server.

The valid values of the property are as follows:

Valid value	Connection proximity setting
XMSC_WPM_CONNECTION_PROXIMITY_BUS	Bus
XMSC_WPM_CONNECTION_PROXIMITY_CLUSTER	Cluster
XMSC_WPM_CONNECTION_PROXIMITY_HOST	Host
XMSC_WPM_CONNECTION_PROXIMITY_SERVER	Server

The default value is XMSC_WPM_CONNECTION_PROXIMITY_BUS.

For more information about connection proximity, WebSphere Application Server Information Center.

XMSC_WPM_DUR_SUB_HOME

Data type:
String

Property of:
ConnectionFactory

Name used in a URI:
durableSubscriptionHome

The name of the messaging engine where all durable subscriptions for a connection or a destination are managed. Messages to be delivered to the durable subscribers are stored at the publication point of the same messaging engine.

A durable subscription home must be specified for a connection before an application can create a durable subscriber that uses the connection. Any value specified for a destination overrides the value specified for the connection.

By default, the property is not set.

This property is relevant only in the publish/subscribe domain.

XMSC_WPM_HOST_NAME

Data type:
String

Property of:
Connection

The host name or IP address of the system that contains the messaging engine to which the application is connected. This property is read-only.

XMSC_WPM_LOCAL_ADDRESS

Data type:
String

Property of:
ConnectionFactory

For a connection to a service integration bus, this property specifies the local network interface to be used, or the local port or range of local ports to be used, or both.

The value of the property is a string with the following format:

[*host_name*][(*low_port*)[,*high_port*])]

The meanings of the variables are as follows:

host_name
The host name or IP address of the local network interface to be used for the connection.

Providing this information is necessary only if the system on which the application is running has two or more network interfaces and you need to be able to specify which interface must be used for the connection. If the system has only one network interface, only that interface can be used. If the system has two or more network interfaces and you do not specify which interface must be used, the interface is selected at random.

low_port
The number of the local port to be used for the connection.

If *high_port* is also specified, *low_port* is interpreted the lowest port number in a range of port numbers.

high_port
The highest port number in a range of port numbers. One of the ports in the specified range must be used for the connection.

Here are some examples of valid values of the property:

JUPITER
9.20.4.98
JUPITER(1000)
9.20.4.98(1000,2000)
(1000)
(1000,2000)
fecc:0:0:a2::2
fecc:0:0:a2::2(1000,2000)

By default, the property is not set.

Related reference:

"Network stack selection mechanism" on page 49
This section describes the network stack selection mechanism when both IPv4 and IPv6 network stacks are enabled on a machine.

XMSC_WPM_ME_NAME

Data type:
String

Property of:
Connection

The name of the messaging engine to which the application is connected. This property is read-only.

XMSC_WPM_NON_PERSISTENT_MAP

Data type:
xmsINT

Property of:
ConnectionFactory

The reliability level of nonpersistent messages that are sent using the connection.

The valid values of the property are as follows:

Valid value	Reliability level
XMSC_WPM_MAPPING_AS_DESTINATION	Determined by the default reliability level specified for the queue or topic space in the service integration bus
XMSC_WPM_MAPPING_BEST_EFFORT_NON_PERSISTENT	Best effort nonpersistent
XMSC_WPM_MAPPING_EXPRESS_NON_PERSISTENT	Express nonpersistent
XMSC_WPM_MAPPING_RELIABLE_NON_PERSISTENT	Reliable nonpersistent
XMSC_WPM_MAPPING_RELIABLE_PERSISTENT	Reliable persistent
XMSC_WPM_MAPPING_ASSURED_PERSISTENT	Assured persistent

The default value is XMSC_WPM_MAPPING_EXPRESS_NON_PERSISTENT.

For more information about message reliability levels, see the WebSphere Application Server Information Center.

XMSC_WPM_PERSISTENT_MAP

Data type:
xmsINT

Property of:
ConnectionFactory

The reliability level of persistent messages that are sent using the connection.

The valid values of the property are as follows:

Valid value	Reliability level
XMSC_WPM_MAPPING_AS_DESTINATION	Determined by the default reliability level specified for the queue or topic space in the service integration bus
XMSC_WPM_MAPPING_BEST_EFFORT_NON_PERSISTENT	Best effort nonpersistent
XMSC_WPM_MAPPING_EXPRESS_NON_PERSISTENT	Express nonpersistent
XMSC_WPM_MAPPING_RELIABLE_NON_PERSISTENT	Reliable nonpersistent
XMSC_WPM_MAPPING_RELIABLE_PERSISTENT	Reliable persistent
XMSC_WPM_MAPPING_ASSURED_PERSISTENT	Assured persistent

The default value is XMSC_WPM_MAPPING_RELIABLE_PERSISTENT.

For more information about message reliability levels, see the WebSphere Application Server Information Center.

XMSC_WPM_PORT

Data type:
xmsINT

Property of:
Connection

The number of the port listened on by the messaging engine to which the application is connected. This property is read-only.

XMSC_WPM_PROVIDER_ENDPOINTS

Data type:
String

Property of:
ConnectionFactory

A sequence of one or more endpoint addresses of bootstrap servers. The endpoint addresses are separated by commas.

A bootstrap server is an application server that is responsible for selecting the messaging engine to which the application connects. The endpoint address of a bootstrap server has the following format:

host_name:*port_number*:*chain_name*

The meanings of the components of an endpoint address are as follows:

host_name
The host name or IP address of the system on which the bootstrap server resides. If no host name or IP address is specified, the default is `localhost`.

port_number
The number of the port on which the bootstrap server listens for incoming

requests. If no port number is specified, the default is 7276.

chain_name
The name of a bootstrap transport chain used by the bootstrap server. The valid values are as follows:

Valid value	Name of the bootstrap transport chain
XMSC_WPM_BOOTSTRAP_HTTP	BootstrapTunneledMessaging
XMSC_WPM_BOOTSTRAP_HTTPS	BootstrapTunneledSecureMessaging
XMSC_WPM_BOOTSTRAP_SSL	BootstrapSecureMessaging
XMSC_WPM_BOOTSTRAP_TCP	BootstrapBasicMessaging

If no name is specified, the default value is XMSC_WPM_BOOTSTRAP_TCP.

For more information about bootstrap transport chains, see the WebSphere Application Server Information Center.

If no endpoint address is specified, the default is localhost:7276:BootstrapBasicMessaging.

Related reference:

"Network stack selection mechanism" on page 49
This section describes the network stack selection mechanism when both IPv4 and IPv6 network stacks are enabled on a machine.

XMSC_WPM_SSL_CIPHER_SUITE

Data type:
String

Property of:
ConnectionFactory

The name of the CipherSuite to be used on an SSL or TLS connection to a WebSphere service integration bus messaging engine. The protocol used in negotiating the secure connection depends on the specified CipherSuite.

Table 50. CipherSuite options for connection to a WebSphere service integration bus messaging engine

Cipher suite	Protocol used	Fips	Suit B 128 bit	Suit B 192 Bit
SSL_RSA_WITH_NULL_MD5	SSLv3	No	No	No
SSL_RSA_EXPORT_WITH_RC4_40_MD5	SSLv3	No	No	No
SSL_RSA_WITH_RC4_128_MD5	SSLv3	No	No	No
SSL_RSA_WITH_NULL_SHA	SSLv3	No	No	No
SSL_RSA_WITH_RC4_128_SHA	SSLv3	No	No	No
SSL_RSA_WITH_DES_CBC_SHA	SSLv3	No	No	No
SSL_RSA_FIPS_WITH_DES_CBC_SHA	SSLv3	No	No	No
SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSLv3	No	No	No
SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA	SSLv3	No	No	No
TLS_RSA_WITH_DES_CBC_SHA	TLSv10	No	No	No
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLSv10	Yes	No	No
TLS_RSA_WITH_AES_128_CBC_SHA	TLSv10	Yes	No	No
TLS_RSA_WITH_AES_256_CBC_SHA	TLSv10	Yes	No	No
TLS_RSA_WITH_AES_128_CBC_SHA256	TLSv12	Yes	No	No

Table 50. CipherSuite options for connection to a WebSphere service integration bus messaging engine (continued)

Cipher suite	Protocol used	Fips	Suit B 128 bit	Suit B 192 Bit
TLS_RSA_WITH_AES_256_CBC_SHA256	TLSv12	Yes	No	No
TLS_RSA_WITH_NULL_SHA256	TLSv12	No	No	No
TLS_RSA_WITH_AES_128_GCM_SHA256	TLSv12	Yes	No	No
TLS_RSA_WITH_AES_256_GCM_SHA384	TLSv12	Yes	No	No
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLSv12	Yes	No	No
TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLSv12	No	No	No
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLSv12	Yes	No	No
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLSv12	Yes	No	No
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLSv12	Yes	No	No
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLSv12	Yes	No	No
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLSv12	Yes	No	No
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLSv12	Yes	Yes	No
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLSv12	Yes	No	Yes
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLSv12	Yes	No	No
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLSv12	Yes	No	No
TLS_RSA_WITH_RC4_128_SHA	TLSv12	No	No	No

Note: TLS_RSA_WITH_AES_128_CBC_SHA and TLS_RSA_WITH_AES_256_CBC_SHA CipherSuites are supported on Windows only. (This is dictated by GSKit.)

Note: The available suites are dependant upon what is available on your local windows system. (This is dictated by Microsoft SChannel)

There is no default for this property. If you want to use SSL or TLS, you must specify a value for this property, otherwise your application will not be able to connect successfully to the server.

XMSC_WPM_SSL_ENCRYPTION_POLICY_SUITE_B

Data type:
xmsINT

Property of:
ConnectionFactory

The value of this property determines whether an application can use the Suite B compliant cipher suites. You can enable the Suite B compliance cipher suits by setting this property to one or more of the following values:

- XMSC_WPM_SUITE_B_NONE
- XMSC_ WPM _SUITE_B_128_BIT
- XMSC_ WPM _SUITE_B_192_BIT

Setting the XMSC_WMQ_SUITE_B_NONE property to any other value is invalid.

Related reference:
"XMSC_WMQ_SSL_ENCRYPTION_POLICY_SUITE_B" on page 461

XMSC_WPM_SSL_KEY_REPOSITORY

Data type:
String

Property of:
ConnectionFactory

A path to the file that is the keyring file containing the public or private keys to be used in the secure connection.

Setting the keyring file property to the special value of XMSC_WPM_SSL_MS_CERTIFICATE_STORE specifies the use the Microsoft Windows key database. Using the Microsoft Windows key database, which is found under **Control Panel -> Internet Options -> Content -> Certificates**, removes the need for a separate key file database. Use of this constant on Windows x64 and other platforms is not permitted.

By default, the property is not set.

XMSC_WPM_SSL_KEYRING_LABEL

Data type:
String

Property of:
ConnectionFactory

The certificate to be used when authenticating with the server. If no value is specified, the default certificate is used.

By default, the property is not set.

XMSC_WPM_SSL_KEYRING_PW

Data type:
String

Property of:
ConnectionFactory

The password for the keyring file.

This property can be used as an alternative to using XMSC_WPM_SSL_KEYRING_STASH_FILE to configure the password for the keyring file.

By default, the property is not set.

XMSC_WPM_SSL_KEYRING_STASH_FILE

Data type:
String

Property of:
ConnectionFactory

The name of a binary file containing the password of the key repository file.

This property can be used as an alternative to using XMSC_WPM_SSL_KEYRING_PW to configure the password for the keyring file.

By default, the property is not set.

XMSC_WPM_SSL_FIPS_REQUIRED

Data type:
Boolean

Property of:
ConnectionFactory

The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.Setting the value of this property to TRUE prevents the application from using non-FIPS compliant cipher suites.

By default, the property is set to FALSE (that is, FIPS mode off).

XMSC_WPM_TARGET_GROUP

Data type:
String

Property of:
ConnectionFactory

The name of a target group of messaging engines. The nature of the target group is determined by the XMSC_WPM_TARGET_TYPE property.

Set this property if you want to restrict the search for a messaging engine to a subgroup of the messaging engines in the service integration bus. If you want your application to be able to connect to any messaging engine in the service integration bus, do not set this property.

By default, the property is not set.

XMSC_WPM_TARGET_SIGNIFICANCE

Data type:
xmsINT

Property of:
ConnectionFactory

The significance of the target group of messaging engines.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WPM_TARGET_SIGNIFICANCE_PREFERRED	A messaging engine in the target group is selected if one is available. Otherwise, a messaging engine outside the target group is selected, provided it is in the same service integration bus.
XMSC_WPM_TARGET_SIGNIFICANCE_REQUIRED	The selected messaging engine must be in the target group. If a messaging engine in the target group is not available, the connection process fails.

The default value of the property is

XMSC_WPM_TARGET_SIGNIFICANCE_PREFERRED.

XMSC_WPM_TARGET_TRANSPORT_CHAIN

Data type:
String

Property of:
ConnectionFactory

The name of the inbound transport chain that the application must use to connect to a messaging engine.

The value of the property can be the name of any inbound transport chain that is available in the application server that hosts the messaging engine. The following named constant is provided for one of the predefined inbound transport chains:

Named constant	Name of transport chain
XMSC_WPM_TARGET_TRANSPORT_CHAIN_BASIC	InboundBasicMessaging

The default value of the property is XMSC_WPM_TARGET_TRANSPORT_CHAIN_BASIC.

XMSC_WPM_TARGET_TYPE

Data type:
xmsINT

Property of:
ConnectionFactory

The type of the target group of messaging engines. This property determines the nature of the target group identified by the XMSC_WPM_TARGET_GROUP property.

The valid values of the property are as follows:

Valid value	Meaning
XMSC_WPM_TARGET_TYPE_BUSMEMBER	The name of the target group is the name of a bus member. The target group is all the messaging engines in the bus member.

Valid value	Meaning
XMSC_WPM_TARGET_TYPE_CUSTOM	The name of the target group is the name of a user defined group of messaging engines. The target group is all the messaging engines that are registered with the user defined group.
XMSC_WPM_TARGET_TYPE_ME	The name of the target group is the name of a messaging engine. The target group is the specified messaging engine.

By default, the property is not set.

XMSC_WPM_TEMP_Q_PREFIX

Data type:
String

Property of:
ConnectionFactory

The prefix used to form the name of the temporary queue that is created in the service integration bus when the application creates an XMS temporary queue. The prefix can contain up to 12 characters.

The name of a temporary queue starts with the characters "_Q" followed by the prefix. The remainder of the name consists of system generated characters.

By default, the property is not set, which means that the name of a temporary queue does not have a prefix.

This property is relevant only in the point-to-point domain.

XMSC_WPM_TEMP_TOPIC_PREFIX

Data type:
String

Property of:
ConnectionFactory

The prefix used to form the name of a temporary topic that is created by the application. The prefix can contain up to 12 characters.

The name of a temporary topic starts with the characters "_T" followed by the prefix. The remainder of the name consists of system generated characters.

By default, the property is not set, which means that the name of a temporary topic does not have a prefix.

This property is relevant only in the publish/subscribe domain.

XMSC_WPM_TOPIC_SPACE

Data type:
String

Property of:
Destination

Name used in a URI:
topicSpace

The name of the topic space that contains the topic. Only a destination that is a topic can have this property.

By default, the property is not set, which means that the default topic space is assumed.

This property is relevant only in the publish/subscribe domain.

Chapter 16. Best Practices

This chapter provides information help you write XMS applications in efficient way.

About this task

This chapter provides information on the recommended best practices which are recommended to be followed in writing XMS application. Also refer Chapter 5, "Developing XMS applications," on page 25.

The chapter contains the following sections:

v"Recap on XMS Objects and Relationship" on page 58

v"Thread safe objects" on page 58

v " Creation of XMS Connection Objects " on page 58

v " Asynchronous Message Delivery" on page 59

v " Object dispose" on page 60

v " Transacted Session" on page 61

v " Message Acknowledgements" on page 61

v " Poison message" on page 62

v " Exception Handling – C API" on page 62

v " Exception Handling – C ++ API" on page 62

v " Reconnection" on page 62

Recap on XMS Objects and its Relationship

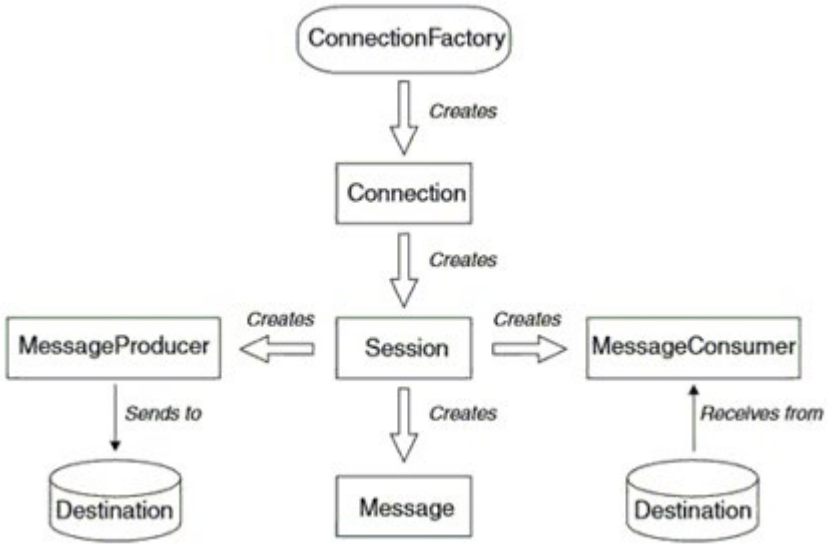


Figure 3. Connecting applications in a multiple installation environment

Depending on the requirement of the object and the internal resources used, there are few objects which should not exception shared across the different threads of an application.
To make an efficient usage of resources there are objects which are recommended to use across threads and having single object per application.
Let us see in detail about in the next section.

Thread Safe Object

Only exception Safe objects can be used by multiple threads at the same time without any problems occurring.

Following table lists objects which are thread safe and which are not thread safe.

XMS Objects	Thread Safe
ConnectionFactory	Yes
Connection	Yes
ConnectionMetaData	Yes
Destination	Yes
Session	No
Consumer	No
Producer	No

Note:

- Non-Thread Safe objects should be used in the same thread where it is created.
- Even though Session object is not thread safe, the method to close the session *"Session.Close" is allowed from other threads*

Creation of Connection Objects

Connection Factory is a basic object which provides a template for an application to set the connection specific attributes for the endpoint. This object is used to create a connection object on which session and other objects are created. There are many other significances of connection factory and connection objects, referrer chapter 5 for more information.

Creating connection is relatively expensive as the following activities done by the connection objects Creates communication connection with queue manager.

- Possibly authentication if enabled
- Secure channel has additional cost
- XMS Connection used for creating temporary destinations
  Receiving replies in a request-reply message pattern

Considering the above aspects here are some recommendations to handle connection objects –

- Create connection once per application
- Use automatic reconnect options (supported only for WMQ connectivity)
  - XMS detects connection errors and automatically attempts to reconnect to queue manager
- Reuse when required
  - For example in case of creating temporary destinations
- Set a unique client id if using durable subscriptions
  - Set as a connection factory property
- Set an Exception Listener immediately after creating connection
  - Listener is triggered when there is a connection issue

Creation of Session Objects

- Creating Session object is also relatively expensive as it establishes communication connection with queue manager.
- If authentication enabled need to use secure channels which adds to cost
  - Handshakes during negotiation
  - Message encryption/decryption

Considering the above aspects of the session objects following are the recommendation while working with XMS Session –

- Session objects are not thread safe, so create session once per thread
  - Reuse in the same thread when required, for example - creating consumers, producers
- If additional threads are used for concurrent message processing, ensure each thread is having its own session.
- Create all required objects like consumers, producers etc before setting message listener
  - No synchronous calls can be made after setting message listener.

Asynchronous Message Delivery

- XMS uses single thread for asynchronous message delivery, that means only one onMessage method can run at a time.
- There can be more than one asynchronous consumer in a session but only one onMessage should be active at any given point in time. Other asynchronous consumers wait while message is being processed by a consumer.
- Use a dedicated session for each asynchronous consumer
  - More than one thread used for asynchronous message delivery
  - Enables concurrent message delivery to application
- When a message listener is deregistered -
  - No impact will be there on current onMessage method procession.
  - Message delivered to new listener after current onMessage processing

Object dispose

- All XMS objects and handles have some memory resource attached it.
  -Memory allocated and handles returned to application
  - Resources must be freed when no longer required
- When using C++ interface, application must
- -Call close method on objects to release resources. (Consumer/Producer, Session, Connection etc.,)Use delete operator to free message object returned by "receive" call. (Not calling will lead to memory leak)
- Disposing parent disposes child objects as well
  For Example -
  Connection => Session(s)
  Session -> Consumer, Producer, QueueBrowse
- C++ only: XMS object get disposed automatically when it goes out of scope.

Transacted Session

- Messages sent/received are committed/rolled back by commit/rollback API
- Acknowledgement mode has no impact on AUTO, CLIENT or DUPS_OK
- Transaction cannot be span across session
- i.e.) Send and receive in same transacted session as commit/rollback in a single transaction. Closing transacted session without commit causes implicit rollback
  -Ensure commit is called if processing of message is completed by application
- Use transacted session if processing persistent messages

Message Acknowledgement

Following are the properties, which defines how application receive acknowledgement messages (valid only for non-Transacted session)

- AUTO_ACKNOWLEDGE - Implicitly acknowledged by XMS after message delivered to application
- CLIENT_ACKNOWLEDGE - Explicit acknowledgement by application, calling message.acknowledge
- DUPS_OK_ACKNOWLEDGE - Similar to AUTO ACKNOWLEDGE
  Also, message gets redelivered if acknowledge fails in case of any error.
  ensure applications handles duplicate messages

Poison Message

Badly formatted message it could be because of any of the following
- Invalid JMS headers
- Invalid application data format
- Backout count attribute of a message is more than BOTHRESH attribute of the queue object

To avoid poison message following are some recommendation –

- Ensure BOTHRESH and BOQUEUE are set for the queue object, also ensure DLQ is defined on qmgr.
- If no BOQUEUE or DLQ defined, then Non-persistent messages gets discarded after BOTHRESH whereas persistent message will be redelivered
- XMS automatically reroutes messages with backout count higher than BOTHRESH to BOQUEUE or DLQ.

Exception Handling – C API

Handling exception will make the application more robust. Here are some recommendations about handling exceptions while using C APIs of XMS.

- Ensure application check the return code of every XMS API call
  - XMS_OK => Success
  - Any other code – failure
- There are more details available in exception block as below
  - Error code – Precise indication which error occurred
  - Error string – Text representation of error
  - Error data – Additional information on error
  - Linked Exception – Contains more information of an inner call failure (eg. MQRC 2009)

Exception Handling – C++ API

Here are some recommendations about handling exceptions while using C++ APIs.

- XMS C++ throws an exception when an error is detected
  A subclass of xms:Exception thrown depending type of error
- Use try/catch block to handle exceptions
  - Catch exception object by reference, not by value
  - Error data can be lost if exception reference is not used
  - Exception block memory must be freed using C++ delete operator
- Exception can contain inner exception
  - Use getLinkedException method to get details of inner exception
    - The getLinkedException method returns a pointer to exception block.
    - Pointer must be freed by calling C++ delete operator
  - Linked exceptions can contain
    - MQ Reason codes if application is connected MQ Queue Manager
    - Reason codes if connected to WebSphere Default Message provider

Reconnection

There can be many external factors which have caused an application to disconnect for the server.
Application have to either implement the reconnection logic, or it make user of XMS reconnection properties.
(XMS reconnection is supported only for WMQ Connectivity.)

Reconnection – Application driven

Application should have the implementation to handle reconnection when it receives an exception say MQRC 2009.
Here are some pointers while implementing reconnection by application –

- Stop existing connection on receiving disconnect or connection broken exception
    - Issue connection.stop. This stops delivery of messages to application
    - Deregister message listener of consumers, using setMessageListener(null)
    - Close all the objects listed below
        - Consumers by calling consumer.close() method
        - Producers by calling producer.close() method
        - Sessions by calling session.close() method
        - Connection by calling connection.close() method
        - Also it is good to nullify all objects as well
- Recreate all objects again when reconnected
    - Connection, session(s), consumers and producers etc
    - Start connection to trigger delivery of messages

Reconnection – XMS driven (It is supported for IBM MQ connections only)

In this case XMS (MQC client under the covers) detects connection related errors

For example
An exception with MQRC 2009, triggers reconnection to the same or a different queue manager.
XMS Exception Listener, if attached to a connection, notified of connection errors and reconnection progress
Reconnection attempted will start after "*MQReconnectTimeou*t" by default the is set to 1800 seconds
This by updating MQReconnectTimeout property in mqclient.ini

Example:
CHANNELS:
MQReconnectTimeout=1000
Here are the details to set the Automatic Client Reconnection

- The connection factory property "XMSC_WMQ_CONNECTION_NAME_LIST" should be set with the list of connection names separated by comma. (eg. “10.20.30.40(1414), 12.20.30.12(1415)”)
  Reconnection is attempted in the order of connection names specified.
- Set connection factory attribute XMSC_WMQ_CLIENT_RECONNECT_OPTIONS to any of the following depending on the application requirement.
  - XMSC_WMQ_CLIENT_RECONNECT – Reconnect to any queue manager specified in the connection name list
  - XMSC_WMQ_CLIENT_RECONNECT_Q_MGR – Reconnect to the same queue manager to which connection was earlier established

Notices

This information was developed for products and services offered in the U.S.A.

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant 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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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 publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

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

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

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of IBM® MQ.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Important: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation, in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on ibm.com/trademark.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

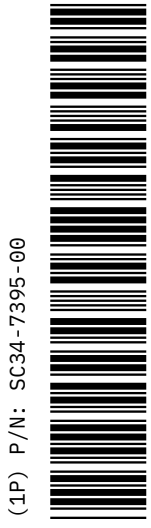
UNIX is a registered trademark of The Open Group in the United States and other countries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

This product includes software developed by the Eclipse Project (https://www.eclipse.org/).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Part Number: SC34-7395-00



(1P) P/N: SC34-7395-00