MQSeries®

LotusScript® Extension

SC34-5404-00

MQSeries®

LotusScript® Extension

SC34-5404-00

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix D, “Notices” on page 187.

First edition (March 1999)

This edition applies to MQSeries for Windows NT® V5.1 and to any subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM® representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page titled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories,
Information Development,
Mail Point 095,
Hursley Park,
Winchester,
Hampshire,
England,
SO21 2JN

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

Tables

About this book

This book describes the IBM MQSeries link LotusScript Extension (MQLSX), and shows how you can use it in your Lotus Notes® applications.

Information in this book includes:

- Where to find more information about MQSeries, Lotus Notes, and LotusScript
- Guidance on how to design and program your applications using the MQLSX
- How to use trace
- Some of the common pitfalls
- Reason codes
- A full reference section on the MQLSX classes and their use
- Code samples and how you can use them in your own applications

Who this book is for

This book is for designers and programmers who develop Lotus Notes applications that interoperate with other, non-Notes applications, using LotusScript.

This book is for you if:

- You are an experienced developer who may or may not be experienced in using LotusScript.
- You have some experience or knowledge of MQSeries.

MQSeries publications

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX® V5.1
- MQSeries for AS/400® V4R2M1
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Digital OpenVMS V2.2
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp® V5.1
- MQSeries for OS/390® V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2
- MQSeries for VSE/ESA V2.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated. (Publications that support the MQSeries Level 1 products are listed in "MQSeries Level 1 product publications" on page x. For a functional comparison of the Level 1 and Level 2 MQSeries products, see the *MQSeries Planning Guide*.)

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
MQSeries: An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Command Reference
The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management
The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, Programmable Command Format (PCF) messages, and installable services.

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes “AMQ” messages issued by MQSeries, applies to these MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

MQSeries Application Programming Guide
The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference
The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary
The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++
MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V4R2M1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java
MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries Java is supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Administration Interface Programming Guide and Reference
The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:

MQSeries for AIX V5.1
MQSeries for HP-UX V5.1
MQSeries for OS/2 Warp V5.1
MQSeries for Sun Solaris V5.1
MQSeries for Windows NT V5.1

MQSeries Queue Manager Clusters
MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

MQSeries for AIX V5.1

MQSeries for HP-UX V5.1
MQSeries for OS/2 Warp V5.1
MQSeries for OS/390 V2.1
MQSeries for Sun Solaris V5.1
MQSeries for Windows NT V5.1

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX Version 5 Release 1 Quick Beginnings, GC33-1867

MQSeries for AS/400

MQSeries for AS/400 Version 4 Release 2.1 Administration Guide, GC33-1956

MQSeries for AS/400 Version 4 Release 2 Application Programming Reference (RPG), SC33-1957

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX Version 2 Release 2 System Management Guide, SC33-1642

MQSeries for Digital OpenVMS

MQSeries for Digital OpenVMS Version 2 Release 2 System Management Guide, GC33-1791

MQSeries for Digital UNIX

MQSeries for Digital UNIX Version 2 Release 2.1 System Management Guide, GC34-5483

MQSeries for HP-UX

MQSeries for HP-UX Version 5 Release 1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp Version 5 Release 1 Quick Beginnings, GC33-1868

MQSeries for OS/390

MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377

MQSeries for OS/390 Version 2 Release 1 Program Directory

MQSeries for OS/390 Version 2 Release 1 System Management Guide, SC34-5374

MQSeries for OS/390 Version 2 Release 1 Messages and Codes, GC34-5375

MQSeries for OS/390 Version 2 Release 1 Problem Determination Guide, GC34-5376

MQSeries link for R/3

MQSeries link for R/3 Version 1 Release 2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx Version 2 Release 2 System Management Guide, GC33-1768

MQSeries for Sun Solaris

MQSeries for Sun Solaris Version 5 Release 1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel Version 2 Release 2 System Management Guide, GC33-1893

MQSeries for VSE/ESA

MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA Version 2 Release 1 System Management Guide, GC34-5364

MQSeries for Windows

MQSeries for Windows Version 2 Release 0 User's Guide, GC33-1822
MQSeries for Windows Version 2 Release 1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT Version 5 Release 1 Quick Beginnings, GC34-5389
MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387
MQSeries LotusScript Extension, SC34-5404

MQSeries Level 1 product publications

For information about the MQSeries Level 1 products, see the following publications:

MQSeries: Concepts and Architecture, GC33-1141

MQSeries Version 1 Products for UNIX Operating Systems Messages and Codes, SC33-1754

MQSeries for UnixWare Version 1 Release 4.1 User's Guide, SC33-1379

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

```
http://www.software.ibm.com/ts/mqseries/
```

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

```
http://www.software.ibm.com/ts/mqseries/
```

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

MQSeries Web site

The MQSeries product family Web site is at:

```
http://www.software.ibm.com/ts/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Where to find more information about LotusScript

Lotus provide the following documentation for LotusScript:

- *The LotusScript Programmers Guide*, Part No. 312106
- *The LotusScript Language Reference*, Part No. 12382

A further source of information is the Lotus home page on the Internet, located at:

```
http://www.lotus.com/
```

Also, the Lotus Enterprise Integration Business Unit Web site:

```
http://www.edge.lotus.com/
```

Chapter 1. Introduction

This book:

- Provides you with help when using the MQLSX
- Describes each of the MQLSX classes with their properties and methods

If you are not very familiar with the Message Queue Interface (MQI), you will find it useful to have a copy of the *MQSeries Application Programming Reference* manual.

MQLSX overview

The MQSeries Link LotusScript Extension (MQLSX) enables your Lotus Notes application to interact with other, non-Notes applications throughout your enterprise.

It enables your Notes LotusScript application the ability to run transactions and access data on any of your enterprise systems that you can access through MQSeries. It gives you integration between Lotus Notes and MQSeries software, extending the scope of Notes to include data and transactions that are part of other environments.

The MQLSX is an application programming interface that you call from LotusScript to access the MQI. It requires an MQSeries environment with an MQSeries application to process the messages that your Notes application generates.

The MQLSX code does not make any calls to Notes. Your applications handle the information updated in Notes, splitting the messages received from MQSeries into fields, and adding them to new or existing Notes documents.

MQLSX has no Notes dependency, just a LotusScript dependency. This allows you to use it from SmartSuite products in a Notes-free environment.

Four samples are provided:

MQLSX Starter sample. You are recommended to use this initially to check that your installation of the MQLSX is successful and that you have the basic MQSeries environment in place. The sample is also provided to demonstrate how LotusScript and the MQLSX can be used.

MQLSX Link sample application. This provides similar functionality to the now obsolete MQSeries Link, and is implemented using the MQLSX. It demonstrates how you can use the MQLSX and the Agent function within Notes to interact with an MQSeries application. The MQSeries sample program, amqslnk0, is the MQSeries application in this instance. See Appendix A, “MQLSX link sample application” on page 145.

MQLSX Link extra sample application. This provides similar functionality to the now obsolete MQSeries Link extra, and is implemented using the MQLSX. See Appendix B, “MQLSX link extra agent sample application” on page 163.

MQLSX Distribution List sample. This sample is provided to demonstrate how MQSeries Distribution Lists work and how to implement them in LotusScript. See Appendix C, “The MQLSX Distribution List sample application” on page 185.

Note: MQLSX can be used only with Notes Release 4.5.1 or later, or Lotus SmartSuite products containing LotusScript 3.1 or higher.

MQSeries environment support

To run the MQLSX in an MQSeries server environment you need the following installed on your system:

- MQSeries for Windows NT, Version 5.1

To run the MQLSX in an MQSeries client environment you need the following installed on your system:

- MQSeries client for Windows NT

Note: The MQSeries client requires access to at least one supporting MQSeries server.

MQSeries link LotusScript Extension or MQSeries Enterprise Integrator for Lotus Notes

You can use the MQSeries Enterprise Integrator (MQEI) to access MQSeries applications from LotusScript programs. The MQEI minimizes the amount of LotusScript programming required. For details visit

`http://www.software.ibm.com/ts/lotus_connections`

This section describes the strengths of each product to help you decide whether you should be using the MQLSX or the MQEI to connect to your enterprise.

MQSeries link LotusScript Extension (MQLSX)

The MQLSX:

- Incorporates the full power of the MQI.
- Allows MQSeries object model conformance. (This is useful if you are already familiar with the MQSeries object model.)
- Performs slightly better than MQEI because there is no database lookup at runtime. (This does depend on the speed of your network and systems where the databases are stored.)
- Does not depend on Notes, but does depend on LotusScript. This allows you to use it from SmartSuite products in a Notes-free environment.

MQSeries Enterprise Integrator for Lotus Notes

MQEI:

- Has a common API for accessing enterprise services regardless of the nature of the enterprise system. The API has a common set of verbs that is independent from the details of each enterprise system. The programmer needs to learn only this single API.

- Has LotusScript program independence from network configuration. For example, names of queue managers and queues are not coded into the LotusScript but into MQEI Service definitions within the MQEI Definition database.
- Has LotusScript program independence from message formats. Similarly, the exact format of messages is not coded into the LotusScript but into MQEI Message definitions within the MQEI Definition database. If you want to use an MQEI Message definition in several places, you need only a single definition that can be shared.
- Can use MQSeries or CICS as network transport.
- Automatically builds MQSeries IMS and CICS bridge headers when sending a message.
- Has integrated security features through the MQEI Security database allow you to sign on to your enterprise systems seamlessly.

Chapter 2. Configuring MQLSX

This chapter explains how to configure MQLSX for operation once you have installed MQSeries. Installation of the MQLSX takes place when you install Windows NT. This chapter is broken into the following sections:

Post-installation

The MQLSX components are installed on your system in the following directories, unless you have changed the base default directory or drive.

- c:\mqm\tools\mqlsx\samples
  A directory containing the MQLSX samples.
- c:\mqm\bin
  A directory containing the Windows NT version of the MQLSX. version of the MQLSX.
- c:\mqm\conv\table
  A directory containing the files that support character conversion.

Additionally, the installation process will have updated the following environment variables:

Variable	Value
GMQ_XLAT_PATH	c:\mqm\conv\table

Copying Notes databases on your Notes client

1. Copy the file gmqlsamp.nsf to the directory where your Notes database files are kept (usually c:\notes\data).
2. Start Notes and select the workspace tab where you want to keep the database.
3. From the Notes File menu, select Database - Open.
4. From the database list, select *MQLSX SAMPLE* (gmqlsamp.nsf) and click Add icon, then click Done.
5. The MQLSX Sample database icon will now appear in your Notes workspace.

To install gmqldist.nsf, gmqlclnt.nsf, gmqlagnt.nsf, gmqlxtra.nsf, mqlink.nsf, or mqlinkx.nsf repeat the steps 1 to 5, but replace gmqlsamp.nsf with the appropriate file name.

Copying a database to a Domino Server

If you are installing a database on multiple Notes workstations, consider storing the database on a server so that several people can access the database from just one copy. Work with your Notes administrator to determine on which server to place a database, because the administrator is aware of server resources, topology, and network protocols. The server on which you store the database should be one that database users can access and that has sufficient memory and disk space to support the database.

Copy databases to your server in the following way:

1. Start Notes and select the workspace where you have installed the database (as described in *Installing Notes databases on a Notes client*).
2. On your workspace, select the icon for the database you want to copy.
3. Select File - Database - New Copy.
4. Next to Server, click the list arrow to display the list of servers. Select the server on wich you want to place the copy.
5. In the Title box, enter a title for the database, if you want a different title.
6. In the File Name box, enter the file name, if you want a different file name. Limit the file name to eight characters plus the .nsf extension.
7. Click OK.

To install any other MQLSX notes database on a Domino Server, follow the steps 1 to 7 above, changing the file name accordingly.

MQLSX environment variables

The MQLSX recognises 5 environment variables:

- GMQ_TRACE
- GMQ_TRACE_LEVEL
- GMQ_TRACE_PATH
- GMQ_XLAT_PATH
- GMQ_MQ_LIB

Environment variables are set from the Windows NT control panel. Double click on the System icon then choose the Environment tab.

GMQ_TRACE

To use the trace facility to help you solve problems, enable it using the GMQ_TRACE environment variable. Unless you are having a problem, you are recommended to run with tracing set off to avoid any unnecessary overheads on your system resources.

For more information, see Chapter 5, “Troubleshooting” on page 33.

GMQ_TRACE_LEVEL

Use the GMQ_TRACE_LEVEL environment variable to set the level of detail you want recorded in your trace file.

For more information, see Chapter 5, “Troubleshooting” on page 33.

GMQ_TRACE_PATH

If you have switched the trace facility on (using the GMQ_TRACE environment variable), you specify the directory where you want the trace files to be stored in this variable. Filenames for the trace file are created at run time. If you do not specify a directory in the GMQ_TRACE_PATH environment variable, the trace files are written to the current working directory (usually c:\notes).

You can identify a trace file by the gmqnnnnn.trc file name (where nnnnn is a five-digit number that represents the process id).

For more information, see “Using trace” on page 33.

GMQ_XLAT_PATH

The GMQ_XLAT_PATH environment variable locates the data conversion tables that are used by the MQLSX. This is automatically set at c:\mqm\conv\table during installation.

For more information, see “Data conversion” on page 18.

GMQ_MQ_LIB

To set the GMQ_MQ_LIB environment variable only if you want to override the inbuilt mechanism for picking up MQSeries libraries, which are dynamically loaded by MQLSX at run time.

For example, set this to specify the MQSeries client library when both the MQSeries client and local queue manager libraries are available locally and you want to force your program to run as an MQSeries client application, even though it is local to the queue manager.

Under normal circumstances, you should not need to set this value.

Caution: Do not set this environment variable to point to the MQLSX. It must be used only to point to the MQSeries libraries. If you are in doubt about which shared library the MQLSX is using, you can run the MQLSX with trace on and look for the entry under EstablishEPS. See “Code-level tool” on page 33 for more information.

Setting up an environment to run the MQLSX

You must ensure that your hardware and software meet the minimum requirements for an MQLSX environment.

Hardware requirements

There are no additional hardware requirements above those listed for MQSeries and Lotus Notes.

Software requirements

- Lotus Notes Release 4.5.1 (Domino Server or Notes client) or later.
- MQSeries server or MQSeries client version 5.1.

Setting up your MQSeries environment

Before you run a LotusScript program using the MQLSX, check that you can start the queue manager to which your script will connect to, and that the necessary queues are in place. Do this by using the command DISPLAY QMGR from runmqsc. The command fails if the queue manager is not running. See “DISPLAY QMGR” in the *MQSeries Command Reference*.

For more information on how to define a queue, see “Rules for naming MQSeries objects” in the *MQSeries Command Reference*

Running an installation verification test

To verify that you have installed the software successfully, run the MQLSX Starter sample provided in the MQLSX package. The MQLSX Starter sample enables you to check that you have installed the software correctly.

```
Uselsx "mqlsx"
```

is used to load the MQLSX.

What is demonstrated in the sample

The sample demonstrates how to use the MQLSX in LotusScript to:

- Connect to a queue manager
- Access a queue
- Put messages on a queue
- Get messages from a queue

Preparing to run the sample

To run the sample you need:

- Lotus Notes (Domino Server or Notes Client)
- An MQSeries Server or an MQSeries Server and an MQSeries Client
- An MQSeries queue manager running
- An MQSeries queue already defined
- The MQLSX installed on the machine where the sample will be run
- The gmqlsamp.nsf database installed on your Notes Client or a Domino server

Running the MQLSX Starter sample

For details of creating and running a queue manager and defining a queue, refer to the *MQSeries System Administration* book.

Starting the sample

1. On your Notes workspace, select the icon for the MQLSX sample.
2. From the main menu, select Create - MQLSX sample. The MQLSX sample form is displayed on the screen.

The MQLSX sample form has the following fields and buttons:

- MQSeries Queue Manager name (field)
- MQSeries Queue name (field)
- Data to be sent (field)
- Data received (field)
- Put Msg on Queue (button)
- Get Msg from Queue (button)

Putting a message on the queue

1. In the field *MQSeries Queue Manager name*, enter the name of the queue manager that you have running. If you leave this field blank, it will not connect to the default queue manager.
2. In the field *MQSeries Queue name*, enter the name of a queue you have defined.
3. In the field *Data to be sent*, enter the message to be put on the queue.
4. Press the button *PutMsgonQueue* to send the message.

Note: The first time you put a message on the queue there may be a short delay while the MQLSX connects to the queue manager.

When the message has been successfully ut on the queue, the data in the field Data to be sent is cleared.

Getting a message from the queue

1. In the field *Queue Manager name*, enter the name of the queue manager that is running.
2. In the field *Queue name*, enter the name of the queue you have defined.
3. Press the button *Get Msg from Queue* to get the message.

The sample gets a message from the named queue and displays it in the *Data received* field. If there is no message to Get, the *Data received* field remains blank.

Exception handling

There are some common problems that you may have while running the sample:

- The queue manager name is incorrect
- The queue manager is not running
- The queue name is incorrect
- The queue does not exist

For common problems, the MQLSX Starter sample attaches a diagnostic text to the error message. Any other error messages include a Reason Code that you can look up in “Reason code” in the *MQSeries Application Programming Reference* book.

The MQLSX Starter sample script

The MQLSX Starter sample database contains sample code written in LotusScript and attached to a Notes form. The sample uses the MQLSX to enable you to create and retrieve messages from an MQSeries queue.

The sample is not intended to demonstrate general programming techniques, so some error checking that you may want to include in a production program has been omitted. However, this sample is suitable to use as a base for your own message queuing programs.

Viewing the MQLSX Starter sample code

If you have a full Notes license, you have full access to the source code of the MQLSX sample, which you can viewin the following way:

1. In your Notes workspace, click on the MQLSX Starter sample icon to select the sample database.
2. From the main menu select View - Design.
3. From the navigation pane, select Forms. The MQLSX sample form will appear in the view pane.
4. Double click on the MQLSX Starter sample form listed in the view pane. This opens the Form Builder window.
5. From the programming panel at the bottom of the Form Builder window you can select Define and Event items to look at the various elements of the sample form.

For example, if you select Define "Put Msg on Queue (Button)" and Event "Click", you can look at the script that will be run when the "Put Msg on Queue" button is pressed.

Select Define "(Globals) MQLSX Sample" to look at Events including:

- Scripts for various subs that handle the calls to the MQLSX.
- Declarations of Object variables.
- Declarations of MQSeries variables.

Chapter 3. Designing and programming using the MQLSX

This chapter complements the information provided by the MQSeries, Lotus Notes, and LotusScript documentation.

It includes:

Designing applications that access non-Notes applications

The advantages of designing and programming using the MQLSX, typical applications, and how to use it are described below.

Advantages

The MQLSX brings you benefits from using both Notes and MQSeries. It extends the scope of Notes to include data and transactions that are part of other environments. Enhancing the information management capabilities of Notes, MQSeries provides commercial messaging for many platforms and time-independent, once-only assured delivery of messages.

Typical applications

Some of the reasons for using MQLSX are:

- Avoiding duplicating or re-typing information; for instance, names and addresses need be held in only one place.
- Providing reporting information on the total enterprise situation, from information held on Notes and other applications.
- Processing financial transactions such as payment of an insurance premium, without depending on someone else entering the information into another system. You can do this from your Notes application using MQSeries to pass on the information, ensuring it is processed when the necessary transaction system is available.

Using the MQLSX

When designing a LotusScript application that uses the MQLSX, the most important item of information is the message that is sent to or received from the remote MQSeries system. Therefore, you must know the format of the items that will be inserted into the message.

You should also know:

- The code page which the remote system runs in
- The encoding that the remote system requires

To help to keep your code portable, it is good practice always to set the code page and encoding even if these are currently the same in both the sending and receiving systems.

Domino Server or Notes client

When you structure your system remember that your MQLSX scripts must run on the same machine on which you have MQSeries is installed. If you do not have MQSeries installed locally, you can make use of Lotus Notes agents. An action initiated on a remote Notes client can cause the triggering of an agent on the Domino server where MQSeries is installed. This mechanism is used in the MQLSX link sample application. You must do this explicitly, because Notes by default will run a script locally.

When your MQLSX script runs, it will need an MQSeries application that picks up the message your script has sent or one that puts a message on a queue for your script to get. For this to work, both your script and the MQSeries application need to know the structure of the message they are dealing with.

Accessing the MQLSX

In the LotusScript editor, under (options) event put the following:

```
Uselsx "mqlsx"
```

For information on the Uselsx statement, see the *LotusScript Language Reference.*

Programming hints and tips

The following hints and tips are in no significant order. They are subjects that, if relevant to the work you are doing, might save you time.

Using large messages

The MQLSX supports messages up to 4 MB long if memory is available. However, when using large messages, Notes restricts:

- Plain text to 64 KB.
- Fields added to the message buffer to 32KB.
- LotusScript string length to 32 000 characters.

  Note: To overcome this limitation, consider appending strings into a rich text field in Notes, which has no size restrictions. The MQLSX has been designed to take advantage of this feature. Copy the data 32000 characters at a time from an MQLSX message into LotusScript strings. From the LotusScript strings, append the data into a rich text field.

For example, where the message data is greater than 32 KB (the maximum length of a string in LotusScript), read the data in multiple parts. This code fragment assumes that the message, MyMsg, has already been taken from the queue using the get method of the MQQueue class and is less than 64 KB in length:

```
Dim MessagePartA As String

Dim MessagepartB As String

...

...

MessagePartA = MyMsg.ReadString(32000)

MessagePartB = MyMsg.ReadString(MyMsg.DataLength)
```

Writing large scripts

The maximum size of a LotusScript program is 32KB. If you need to write a script larger than this, consider subdividing your script code into functions or use the %include function within Notes.

For a full list of LotusScript limits, see the appendix in the *LotusScript Language Reference* manual.

Embedded nulls in a string

The MQSeries constants used for the initialization of three MQMessage properties:

- MQMI_NONE (24 NULL characters)
- MQCI_NONE (24 NULL characters)
- MQACT_NONE (32 NULL characters)

are not supported by the MQLSX, but the LotusScript String function allows you to do the same thing.

To set the MessageId of an MQMessage to MQMI_NONE:

```
mymessage.MessageId = String(24,0)
```

To set the CorrelationId of an MQMessage to MQCI_NONE:

```
mymessage.CorrelationId = String(24,0)
```

To set the AccountingToken property of an MQMessage to MQACT_NONE:

```
mymessage.AccountingToken = String(32,0)
```

Message Descriptor properties

Where an MQSeries application is the originator of a message and MQSeries generates the

- AccountingToken
- CorrelationId
- MessageId

you are recommended to use the AccountingTokenHex, CorrelationIdHex, and MessageIdHex properties if you want to look at their values, or manipulate them in any way, including passing them back in a message to MQSeries. The reason for this is that MQSeries-generated values are strings of bytes that have any value from zero through to 255 inclusive; they are not strings of printable characters.

Where your MQLSX script is the originator of a message and you generate the

- AccountingToken
- CorrelationId
- MessageId

you are recommended to use the AccountingToken, CorrelationId, and MessageId properties.

Object out of scope

It is good programming practice to delete an object before it goes out of scope.

For example:

```
Dim qms As MQSession

Dim qm As MQQueueManager

...

...

Set qm As qms.AccessQueueManager(MQ_queue_manager)

...

...

qm.Disconnect

Delete qm
```

The Delete call deletes the storage allocated during the AccessQueueManager method.

Receiving a message from MQSeries

There are several ways of receiving a message from MQSeries:

- Polling by issuing a GET followed by a wait, using the LotusScript TIMER function.
- Issuing a GET with the Wait option; you specify the wait duration by setting the WaitInterval property. This is recommended when, even though you set your system up to run in a multithreaded environment, the software running at the time may run only single threaded. This avoids your system locking up indefinitely.

  Note: Issuing a GET with the Wait option and setting the WaitInterval to MQWI_UNLIMITED causes your system to lock up until the GET call completes, if the process is single threaded.
- Issuing a GET without the Wait option. In this case, once your script has issued the call, control is passed to the next script waiting to run. This second script, and any other scripts that may run before the original script regains control, must not affect any of the objects that the original script expects to be the same as at the time it lost control.

Automatic buffer management

The MQLSX MQMessage object controls the size of a buffer, dynamically changing (within the limitations of the system) the buffer size to accommodate the data in a message.

The default size of a buffer is 2 KB. If you want to restrict how much of a message your application gets, and you want it to get more than 2 KB, you must use the ResizeBuffer method before getting the message with the MQGMO_ACCEPT_TRUNCATED option. However, if you do not wish to use the MQGMO_ACCEPT_TRUNCATED option, the MQLSX does handle all automatic buffer management.

Disconnecting from MQSeries

Within your LotusScript program you are recommended to use the Disconnect method before the program ends.

If you do not explicitly disconnect from the MQSeries queue manager, the results are unpredictable. For example, any messages placed on a queue under syncpoint may not be committed if the program does not explicitly call the Disconnect method.

Using the IMS™ Bridge

For applications that put or get messages that involve access through the MQSeries IMS bridge for OS/390, there are a few points to bear in mind.

- When you put a message destined for an IMS system, use the IMS bridge header (MQIIH). Set the MQIIH_Format to MQIMSVS and the MQMessage Format property to MQIMS.
- When you get a message back from the IMS bridge, either unpack the header within your application, or offset the message by 84 bytes to ignore the header and point to the start of the message from IMS. An offset of 89 bytes takes you to the first data byte of the message.

See “MQIIH - IMS bridge header” in the *MQSeries Application Programming Reference* for more information.

For more information about designing and writing applications to use the services MQSeries provides, see Chapter 2, “Overview of application design” in the *MQSeries Application Programming Guide*.

You may find the following code samples useful when using the IMS Bridge. However, these are code fragments and should not be taken as full programming solutions.

```
' Declare additional IMS Items.
These fields are needed to
submit an IMS transaction.

Dim II As Integer

Dim zz As String

Dim trancode As String
```

```
Dim trandata As String

' Additional MQSeries items

Dim MQITII_NONE As String

' The default value for TraninstanceID, not supplied in MQLSX

MQITII_NONE=string(16,0)

' 16 nulls

Dim MQIIH_Encoding As Long

Dim MQIIH_CodedCharSetId As Long

Dim MQIIH_Format As String

Dim MQIIH_Flags As Long

Dim MQIIH_LTermOverride As String

Dim MQIIH_MFSMapName As String

Dim MQIIH_ReplyToFormat As String

Dim MQIIH_Authenticator As String

Dim MQIIH_TranInstanceId As String

Dim MQIIH_TranState As String

Dim MQIIH_CommitMode As String

Dim MQIIH_SecurityScope As String

Dim MQIIH_Reserved As String

' Set additional MQIIH items.
This is to put default values
into the MQIIH fields.

' Note that MQIIH_Format and MQIIH_ReplyToFormat must be set to
"MQIMSVS".

MQIIH_Reserved As String

MQIIH_Encoding = Mqenc_native

MQIIH_CodedCharSetId = Mqccsi_q_mgr

MQIIH_Format = "MQIMSVS "

MQIIH_Flags = MQIIH_NONE

MQIIH_LTermOverride = "        "
```

```
MQIIH_MFSMapName = "        "

MQIIH_ReplyToFormat = "MQIMSVS "

MQIIH_Authenticator = MQIAUT_NONE

MQIIH_TranInstanceId = MQITII_NONE

MQIIH_TranState = MQITS_NOT_IN_CONVERSATION

MQIIH_CommitMode = MQICM_SEND_THEN_COMMIT

MQIIH_SecurityScope = MQISS_CHECK

MQIIH_Reserved = " "

' Write IIH to MQ message MQmsg.

MQmsg.writestring(Mqiih_struc_id)

MQMsg.writelong(Mqiih_version_1)

MQMsg.writelong(Mqiih_length_1)

MQMsg.writelong(MQIIH_Encoding)

MQMsg.writelong(MQIIH_CodedCharSetId)

MQMsg.writestring(MQIIH_Format)

MQMsg.writelong(MQIIH_Flags)

MQMsg.writestring(MQIIH_LTermoverride)

MQMsg.writestring(MQIIH_MFSMapname)

MQMsg.writestring(MQIIH_ReplyToFormat)

MQMsg.writestring(MQIIH_Authenticator)

MQMsg.writestring(MQIIH_TranInstanceId)

MQMsg.writestring(MQIIH_TranState)

MQMsg.writestring(MQIIH_CommitMode)

MQMsg.writestring(MQIIH_SecurityScope)

MQMsg.writestring(MQIIH_Reserved)

' Write IMS message to MQ message MQmsg.

ll = 4 + Cint(Len(trancode)) + Cint(Len(trandata))    ' total
length of message

zz = "  "

' reserved for IMS
```

```
MQMsg.writeshort(ll)

MQMsg.writestring(zz)

MQMsg.writestring(trancode)

MQMsg.writestring(trandata)

MQmsg.messagetype = 8

'Set message as DATAGRAM

MQmsg.format = "MQIMS   "

'Set Data Convert Format for IMS
Bridge.

MQmsg.ReplyToQueueManagerName = "VM03"

' MQM for reply message

MQmsg.ReplyToQueueName = "EF.VM03.SDRC.REMOTE"

' Reply Queue on above MQM

MQmsg.UserId = "MYUSER"

' 12 byte userid - default is blanks

' Unpack the reply message.

MQMsg.DataOffset = 89

' Jump past the IIH, ll, zz, & attribute byte

replymsg = MQMsg.ReadString(MQMsg.DataLength)
' get the reply from IMS
```

Data conversion

Data conversion is necessary when a message is created on one system and processed by another system where the character set and encoding are different.

Data conversion can be performed by:

- The queue manager. This can be the queue manager receiving the message, prior to an application issuing an MQGET call, or the queue manager sending the message to another queue manager. A queue manager is restricted to a set of built-in formats.
- Writing your own data-conversion exit. This is invoked when an application gets a message from a queue.
- Using the MQLSX read and write methods

When considering which of these is the most appropriate for your application, note that:

- MQSeries does not support data conversion on all platforms.
- Data conversion by a queue manager using the built-in formats converts the whole message.
- The MQLSX read and write methods convert individual fields within a message.

Data conversion by MQSeries

If you want the whole message converted when you put or get a message, you can request that MQSeries does the conversion rather than the MQLSX.

When you retrieve a message, use the MQGMO_CONVERT option on the Get method.

When you put a message on a queue, if you have not set the CharacterSet (which corresponds to the MQSeries CodedCharSetId attribute) and Encoding properties they are, by default, set to the values of the system that constructed the message. Use the MQGMO_CONVERT on the Put method within your script, or on the MQGET call within your MQSeries application.

If you set the MQGMO_CONVERT option on the MQGET call within your MQSeries application, MQSeries attempts to convert the message, unaware of any conversion that has already taken place within your MQLSX application. If you have set the CharacterSet (CodedCharSetId) and Encoding properties to match the contents of the message, data conversion will be attempted only when necessary.

See Chapter 11, “Writing data-conversion exits” in the *MQSeries Application Programming Guide* for more information about how data conversion works within MQSeries. It covers the rules a queue manager follows to determine if it is to do the conversion, as well as how to write and invoke your own data-conversion exit.

Data conversion by the MQLSX

The Get method defined in the MQQueue class retrieves a message from an MQSeries queue. This call copies the message from the queue into an internal MQLSX message object.

The Put method defined in the MQQueue class takes the message from the internal MQLSX message object and places it on an MQSeries queue.

In a message, the CharacterSet (CodedCharSetId) and Encoding properties define the code page and the numeric encodings used within your data part of the message.

Benefits of data conversion by MQSeries

You do not need to know the destination of a message.

If your application is dealing with large messages, or a large number of small messages, the server running MQSeries may have more resources available to handle the volume of data conversion, rather than possibly stretching your workstation to its limits by the conversion taking place within the MQLSX.

You can avoid the situation whereby a message has been removed from the MQSeries queue; however data conversion within the MQLSX fails because either there are missing entries in the gmqlccs.tbl or the conversion file is not available.

Benefits of data conversion using the MQLSX

There are read and write methods provided for the different data types. Therefore, the conversion is more specific to the contents of the individual field.

If your application uses only a small portion of a large message, the amount of data conversion is considerably reduced using the read methods.

You do not have to understand and write any data-conversion exit programs.

Using the MQLSX methods

Information on the MQLSX methods is described below.

Read methods

The read methods use the values held in the CharacterSet (CodedCharSetId) and the Encoding properties to carry out any necessary data conversion, so that the data is correctly presented to your LotusScript application.

Write methods

When constructing a message, your application should set the CharacterSet (CodedCharSetId) and the Encoding properties to match the requirements of the receiving system. (If you don't specify any values, they take those for the platform your application is running on.) These values are used by each subsequent write method as your application builds the message, as well as by the MQSeries application when it processes the message.

You should not modify the CharacterSet (CodedCharSetId) and Encoding properties after you have invoked the Get method, or after you have started constructing a message that will later be put on a queue. If you do change these properties when you are constructing a message, such that they do not match your message data, you must not use the MQGMO_CONVERT on the MQGET call unless you want your message data converted.

Two forms of data conversion are supported by the MQLSX:

- Numeric coding
- Character set Conversion

Numeric coding

If you set the Encoding Property, the following methods will convert between different numeric encoding systems:

- ReadLong Method
- WriteLong Method
- ReadShort Method
- WriteShort Method
- ReadFloat Method
- WriteFloat Method
- ReadDouble Method
- WriteDouble Method

- ReadUInt2 Method
- WriteUInt2 Method
- ReadUInt4 Method
- WriteUInt4 Method
- ReadDecimal2 Method
- WriteDecimal2 Method
- ReadDecimal4 Method
- WriteDecimal4 Method
- ReadInt2 Method
- WriteInt2 Method
- ReadInt4 Method
- WriteInt4 Method

MQSeries provides a header file, cmqc.h, that defines the data encoding for the platform you are running on:

```
#define MQENC_NATIVE 0x00000222L
// MQENC_NATIVE is further broken down into individual encodings
for binary integers.....
/* Encodings for Binary Integers */
#define MQENC_INTEGER_NORMAL  0x00000001L
#define MQENC_INTEGER_REVERSED  0x00000002L
```

These definitions can be used when you need to send a number either from a little endian (for example, Intel) to a big endian system (for example, System/370) or the other way around.

Example

To send an integer from an Intel system to a System/370 operating system:

```
Dim msg As New Mqmessage  ' Define an MQSeries message for our use...

Dim local_num As long     ' Define a long integer

myenc = msg.Encoding      ' Currently 546 (or 222 hex.)

Print myenc               ' Print the current Encoding property value for
                            information

msg.Encoding=273          ' Set the encoding property to 273 (or 111 in
                            hex.)

myenc = msg.Encoding      ' Get a copy

Print myenc               ' Print it to see the change

local_num = 1234          ' Set it

msg.WriteLong(local_num)  'Write the number into the message
```

The WriteLong method reads the encoding property and if it is different from the native local machine setting, it converts the integer accordingly.

Character set conversion

Character set conversion is necessary when you send a message from one system to another system where the code pages are different. Character set conversion is included in the methods:

- ReadString Method
- WriteString Method
- ReadNullTerminatedString Method
- WriteNullTerminatedString Method
- ReadUTF Method
- WriteUTF Method

Note: You must set the CharacterSet property to a supported character set value (CCSID). If the CharacterSet property is set to an invalid value, no error is reported until a ReadString or WriteString method is called.

The MQLSX uses conversion tables, installed in the conv directory, to perform Character set conversion.

Check that the GMQ_XLAT_PATH environment variable has been set to point to the directory on your system to which you copied the contents of the conv directory.

If you do not set the environment variable, the MQLSX looks for the conversion files in your current working directory (for instance, the NOTES directory when you are running NOTES).

Example

To convert strings automatically to Code Page 437:

```
Dim msg As New Mqmessage                 'Define an MQSeries message

msg.CharacterSet = 437                   'Set code page required

mymsg.Writestring "A character string"   'Put the character string in the
                                          message
```

The WriteString method converts the Unicode passed in from LotusScript to the character set associated with the message. This occurs before the string is put in the buffer that is sent to the MQSeries server when you invoke the put method.

Similarly, with the ReadString method, the incoming MQSeries message (using the Get method) has a code page associated with it (in the MQMD). The data is converted from this code page to Unicode before being passed to LotusScript.

Establishing a character set for an environment

As part of the initialization of the MQLSX, the default character set is established for the Notes instance under which the MQLSX initialization is taking place in Windows NT by a call to the system routine GetConsoleCP. The value returned is used as the CCSID, unless no value is returned, when the registry value for OEMCP from SYSTEM\CURRENTCONTROLSET\NLS\CODE PAGE is used. If a value cannot be found, the default value of 850 is used.

The role of the readme.ccs, gmqlccs.tbl and NNNNMMMM.tbl files

The readme.ccs file describes exactly which character sets are supported. The readme.ccs file is stored in the conv directory of the MQLSX package.

The gmqlccs.tbl file holds information concerning the relationship between character sets and code pages. The GMQ_XLAT_PATH environment variable enables the MQLSX to locate the gmqlccs.tbl file during the initialization of the MQLSX. The contents of this file are loaded into memory. When character conversion is required (when a ReadString or WriteString method is called), the information held in memory (from the gmqlccs.tbl file) is used to establish which conversion tables are required.

In this MQLSX release (where Unicode conversion is used), these tables are named 34B0mmmm.tbl and nnnn34B0.tbl. For example, where the character set is set to 437, a WriteString method needs 34B001B5.tbl, where 01B5 is the hexadecimal value of 437. The first time the MQLSX uses a table, the contents of the table are loaded into memory ready for when it is needed again.

Note: File names can be in either upper or lower case.

MQLSX character data conversion in detail

Strings created under Lotus Notes are stored in LMBCS (Lotus MultiByte Character Set, which is closely aligned with Unicode) format. When you issue the Uselsx "mqlsx" command within your LotusScript program, the MQLSX classes register themselves, together with their properties and methods.

In particular, the MQMessage class (which contains the WriteString and ReadString methods) registers itself as using Unicode. This means that when LotusScript invokes a method or property of the MQMessage class that involves an input or output string parameter, this is passed in or passed back using Unicode.

WriteString method

The WriteString method requires a string parameter, which is passed by Notes to the MQLSX as Unicode.

For example, if you enter a dollar sign ($) from your keyboard into a Notes field and subsequently pass this character in a Notes string to the MQLSX using the WriteString method, the MQLSX sees X'0024' - this being the Unicode assigned code-point for dollar. Similarly, if you have a keyboard that supports the pound sign (£), and pass this character in a Notes string to the MQLSX using the WriteString method, the MQLSX sees X'00A3'.

On receiving the method call, the MQLSX uses the gmqlccs.tbl together with the appropriate conversion table to convert from Unicode to the local code page or to another code page that you have specified using the CharacterSet property of the message.

For example, if the local code page is 437 (MS-DOS Latin US) or the CharacterSet property is set to 437 explicitly, the Unicode value for pound sign (X'00A3') is converted into the 437 code point for the pound sign, which is X'9C' using the conversion table 34B001B5.tbl. This is the data passed that is in the MQSeries message. The MQMD in the resultant MQSeries message indicates that the code page under which the message was generated is 437.

ReadString method

The ReadString method returns a string and uses the reverse mechanism to WriteString.

For example, If you receive an MQSeries message (using the Get method), the MQMD contains the code page that the message was generated under. If you get the message that was put out in the WriteString method example, any incoming pound signs (£) are present in the message as X'9C'. The ReadString method on the data containing these pound signs converts X'9C' to X'00A3' using the table 01B534B0.tbl and passes it back to Notes.

Losing data when using WriteString

If the data you enter into your string from Notes contains characters not supported in the code page to which you are converting, in the process of converting from Unicode to that code page they are converted to the substitute character (normally X'7F' for ASCII code pages), losing the original data.

In most cases, this is not a problem because the characters you can enter from your keyboard into a Notes field are all supported by the local code page. However, if you use the CharacterSet property to specify a different code page, some characters may be lost depending on the match between the local code page and the one to which you are converting. For ASCII code pages, the characters between X'20' and X'7F' should all be converted.

You risk losing information if you attempt to manipulate Strings to contain data by using the Chr or UChr LotusScript functions.

One way of preventing this loss is to set the CharacterSet property to 1200, which is Unicode. This prevents any conversion taking place; the Unicode string is passed to the MQLSX unconverted.

One implication of this is that any receiving MQSeries application must be able to support Unicode. Another implication is that the data takes two bytes for every character.

When data conversion fails

Data conversion fails if:

- You specify an invalid character set or one for which you do not have the conversion table.
- You have not set the GMQ_XLAT_PATH environment variable.
- The gmqlccs.tbl file is not found in the directory specified in GMQ_XLAT_PATH.

If the MQLSX WriteString method fails to convert the data in a field because the conversion tables are not available, you are likely to get the MQSeries return code MQRC_NOT_CONVERTED (2119). No data is written to the message.

If the MQLSX ReadString method fails to convert the data in a message, the message is no longer on the MQSeries queue (after the success of the GET method). The message is in the buffer used by your program, so you can use the PUT method to place it on an MQSeries queue if you wish to exit the program and

resolve the problem later. Alternatively, you could consider changing your application and do the conversion within MQSeries before using the GET method.

If the CCSID entry is missing from the gmqlccs.tbl, you are likely to get the MQSeries return code MQRC_TARGET_CCSID_ERROR (2115). The supported conversions are listed in the readme.ccs file provided in the conversion directory.

Note: When you run the MQLSX in the MQSeries Client environment and the MQSeries server does not support the CCSID, you must set the MQCCSID environment variable, otherwise the connection fails with MQSeries reason code 2059.

Chapter 4. Error handling

Each MQLSX object includes properties to hold error information and a method to reset them. The properties are:

- CompletionCode
- ReasonCode

The method is:

- ClearErrorCodes

Each object also raises events:

- Mqwarning
- Mqerror

How error handling works

Your MQLSX script or application invokes an MQLSX object's method, or accesses or updates a property of the MQLSX object:

- The ReasonCode and CompletionCode in the object concerned are updated.
- The ReasonCode and CompletionCode in the MQSession object are also updated with the same information.

If the CompletionCode is not equal to MQCC_OK:

- The MQLSX issues an Mqwarning or Mqerror event against the object concerned.
- The event passes to an event handler for the object, if there is one available. If your event handler has cleared the problem, use the ClearErrorCodes method within the error handler, which resets the ReasonCode to MQRC_NONE and the CompletionCode to MQCC_OK.
- On return from the event handler processing, if any, the MQLSX copies the object ReasonCode and CompletionCode once again to the MQSession object.

If the CompletionCode is still not equal to MQCC_OK:

- The MQLSX issues an Mqwarning or Mqerror event against the MQSession object.
- The event passes to an event handler for the MQSession object, if there is one available. If your event handler has cleared the problem, use the ClearErrorCodes method within the error handler which resets the ReasonCode to MQRC_NONE and the CompletionCode to MQCC_OK.

If the MQSession object CompletionCode is equal to MQCC_ERROR:

- The MQLSX generates a LotusScript error, number 32000. Use this within your script using the On Error statement to process it.
- Use the Error$ function to retrieve the associated error string. This is in the form:

  ```
  MQLSX: ReasonCode=nnnn
  ```

  where nnnn is the latest MQSession object ReasonCode.

For more information on how to use the On Event and On Error statements, see the *LotusScript Language Reference* manual.

Getting a property

This is a special case because the CompletionCode and ReasonCode are not always updated:

- If a property get succeeds, the object and MQSession object ReasonCode and CompletionCode remain unchanged.
- If a property get fails with a CompletionCode of warning, the ReasonCode and CompletionCode remain unchanged and no Mqwarning event is raised.
- If a property get fails with a CompletionCode of error, the ReasonCode and CompletionCode are updated to reflect the true values, and error processing proceeds as described.

Using Events and Error handlers

In general, MQLSX errors (unlike other LotusScript errors) can be handled using an Event or Error handler or a combination of these.

Error handling using Event handlers

Error and warning event handlers for a specific MQLSX object (an MQQueue object, for example) can be registered by the in line LotusScript code as shown:

```
Set MQq =
MQQmgr.AccessQueue(Queue_name.Text,OpenOptions,"","","")

On Event Mqwarning From MQq Call WarningFromMQq

On Event Mqerror From MQq Call ErrorFromMQq
```

If the CompletionCode from an MQLSX object method (or update) is not MQCC_OK, the MQLSX issues an Mqwarning or Mqerror event (depending on severity) for the object concerned, giving control to the appropriate event handler. An event handler might typically perform all required processing for an Mqwarning event, allowing the in line LotusScript code to continue as if no warning had occurred. For an Mqerror event, however, the in line code may require access to the reason code or some other indication that the operation was unsuccessful. MQRC_NO_MSG_AVAILABLE, for example, indicates that no message was returned from a 'Get' but in many cases this is not an error. The following examples illustrate how this can be accomplished using event handlers.

```
Examples of Mqwarning and Mqerror event handlers

'* Mqwarning event handler

Sub WarningFromMQq(MQq As MQQueue)

     Messagebox "Warning From MQq, reason code: " &MQq.ReasonCode

     MQq.ClearErrorCodes  '* clears reason code, completion
code, and event

End Sub

'* Mqerror event handler

Sub ErrorFromMQq (MQq As MQQueue)

     If MQq.ReasonCode = MQRC_UNKNOWN_OBJECT_NAME Then

          Messagebox "Error From MQq, invalid queue name"

          uidoc.FieldClear ("MQqueue_name")

     Elseif    MQq.ReasonCode = MQRC_NO_MSG_AVAILABLE Then

          Messagebox "No message available"

          uidoc.FieldClear ("MsgRecvd")

     Else

          Messagebox "Error From MQq, reason code: "
&MQq.ReasonCode

          uidoc.FieldClear ("MQqueue_name")

     End If

     GlobalReasonCode = MQq.ReasonCode

     GlobalCompletionCode = MQq.CompletionCode

     MQq.ClearErrorCodes  '* clears reason code, completion
code, and event

End Sub
```

Note: The ClearErrorCodes method must be issued by the Mqerror event handler to prevent a LotusScript error being raised when the event completes. Thus status information can only be passed from the event handler to the in line LotusScript code using global variables. In this example, the reason and completion codes are preserved; other examples might use a fatal error indication (such as the MQLSX sample programs, for example, that use MQFatalError).

If Mqwarning and Mqerror event handlers are not registered for the object concerned or the completion code is not cleared using ClearErrorCodes, control is given to the appropriate MQSession event handler. Mqwarning and Mqerror event handlers for the MQSession object are constructed in a similar way to those described and can be registered as shown in the example:

```
Set MQqms = New MQSession

On Event Mqwarning From MQqms Call WarningFromMQqms

On Event Mqerror From MQqms Call ErrorFromMQqms
```

Error handling using error handlers

If an Mqerror event handler is not registered for the MQSession object concerned or the completion code is not cleared by the event handler issuing ClearErrorCodes, the MQLSX generates a LotusScript error 32000. If an Mqwarning event handler is not registered for the MQSession object concerned, or if the completion code is not cleared by the event handler issuing ClearErrorCodes, no LotusScript error is generated. MQLSX warnings can thus be handled using an event handler as described above, or, if no Mqwarning event handler is registered, the completion and reason codes associated with the warning are preserved for handling by the in line LotusScript code.

To handle MQLSX errors exclusively using error handlers, no Mqerror event handler should be registered. This approach offers the advantages handling both MQLSX and non-MQLSX LotusScript errors in a similar way and making completion and reason code information available to the in-line LotusScript code without the use of global variables.

Error handlers can be registered in the top-level or lower-level procedures as follows:

```
    On Error GoTo HandleError
```

'HandleError' processing might typically be positioned at the end of the procedure as shown (for an objected-oriented implementation):

```
'*****************************************************************
'*  Handle errors
'*****************************************************************
     Exit Sub
HandleError:
     Dim ErrRcd As ErrorRecord
     Set  ErrRcd = New ErrorRecord
     End
End Sub
```

The associated class definition might appear as follows. This can clearly be modified to log the error or perform whatever action is most appropriate.

```
Const MQLSX_ERROR = 32000
Class ErrorRecord
     MQqms As MQSession
     Sub New
          If Err = MQLSX_ERROR Then
               Set MQqms = New MQSession
               If MQqms.ReasonCode <> MQRC_NO_MSG_AVAILABLE  Then
                    Print "MQLSX Error ", Error(), Err(), Erl()
               End If
               MQqms.ClearErrorCodes
          Else
              Print "LotusScript Error ", Error(), Err(), Erl()
          End If
     End Sub
End Class
```

For a procedural implementation, the body of the subroutine 'New' above would replace the 'dim' and 'set' statements below the 'HandleError' label.

Note that an error in a called subroutine that contains no 'On Error' statement will be caught by an 'On Error' statement in a higher or top-level procedure.

To ignore an MQSeries warning (such as MQRC_NO_MSG_AVAILABLE) each affected procedure must issue a statement of the form:

```
On Error MQLSX_ERROR resume next
```

The MQLSX reason code can then be examined by the in line LotusScript code and handled (ignored or otherwise) or returned to the caller for processing as appropriate. The following example is an extract from a class method that attempts to get a message using the MQQueue 'Get' method. RCode& and CCode& are properties of the same (current) class definition as the method containing the code fragment shows, and are used here to preserve the completion and reason codes from the 'Get' for subsequent access by the user of the object. A second On Error MQLSX statement is included to reinstate error handling for subsequent MQLSX methods.

```
On Error MQLSX_ERROR Resume Next

.

MQreplyq.Get MQMsg, MQgmo

.

CCode& = MQreplyq.CompletionCode

RCode& = MQreplyq.ReasonCode

.

On Error MQLSX_ERROR GoTo HandleError
```

Using MQLSX from Domino web agents with multithreading

If the DominoAsynchronizeAgents parameter is set to 1 in the notes.ini file, the Domino http server process runs multithreaded, each web agent running under a separate thread. This gives a much improved performance for web-based applications.

MQLSX is capable of being used in such an environment. However, it is strongly recommended that you run at least Domino 4.6.1.

Chapter 5. Troubleshooting

This chapter explains:

Code-level tool

You may be asked by the IBM Service team what level of code you have installed.

To find this out, run the utility program.

From the command prompt, change to the directory containing the

`mqlsx.dll`

or add the full path name and enter:

```
gmqlevel mqlsx.dll > xxxxx.xxx
```

where xxxxx.xxx is the name of the output file.

If you do not specify an output file, the detail is displayed on the screen.

Using trace

The MQLSX includes a trace facility to help the service organization identify what is happening when you have a problem. It shows the paths taken when you run your MQLSX script. Unless you have a problem, you are recommended to run with tracing set off to avoid any unnecessary overheads on your system resources.

There are three environment variables that you set to control trace:

- GMQ_TRACE
- GMQ_TRACE_PATH
- GMQ_TRACE_LEVEL

You set these variables in one of two ways.

1. From a command prompt, from which you must subsequently start Notes, because this is only effective locally.
2. By putting the information into your system startup file. This is effective globally.
    - Select Main - Control Panel on Windows NT

Note: When deciding where you want the trace files written, ensure that the user has sufficient authority to write to, not just read from, the disk.

If you have tracing switched on, it will slow down the running of the MQLSX, but it will not affect the performance of your Notes or MQSeries environments. When you no longer need a trace file, it is your responsibility to disable it.

You must stop Notes running to change the status of the GMQ_TRACE variable.

Note: The MQLSX trace environment variable is different to the trace environment variable used within the MQSeries range of products. Within the MQSeries range of products, the trace environment variable is used to specify the name of the trace file. Within the MQLSX, the trace environment variable turns tracing on. If you set the variable to any string of characters, tracing will remain switched on. It is not until you set the variable to null that tracing is turned off.

Trace filename and directory

The trace file name takes the form GMQnnnnn.trc, where nnnnn is the id of the Notes process running at the time.

Command	Effect
SET GMQ_TRACE_PATH=drive:\directory	Sets the trace directory where the trace file will be written
SET GMQ_TRACE_PATH=	Removes the GMQ_TRACE_PATH environment variable, the trace file is written to the current working directory (when Notes is started)
SET GMQ_TRACE_PATH	Displays the current setting of the trace directory path
ECHO %GMQ_TRACE_PATH%	Displays the current setting of the trace directory path on Windows NT
SET GMQ_TRACE=xxxxxxxx	This sets tracing ON. You switch tracing on by putting one or more characters after the '=' sign For example: SET GMQ_TRACE=yes or SET GMQ_TRACE=no In both of these examples, tracing will be set ON
SET GMQ_TRACE=	Sets tracing OFF
SET GMQ_TRACE	Displays the contents of the environment variable
ECHO %GMQ_TRACE%	Displays the contents of the environment variable on Windows NT
SET	Displays the contents of all the environment variables on Windows NT

Trace level

The environment variable GMQ_TRACE_LEVEL allows you to control how much detail is recorded in the trace file. It can be set to any numeric value greater than zero, although any value above nine does not provide any more information.

In addition, you can suffix the value with a + (plus) or - (minus) sign. Using the plus sign, the trace includes all control block dump information and all informational messages. Using the minus sign includes only the entry and exit points in the trace, (that is, no control block information or text is output to the trace file).

The default value of GMQ_TRACE_LEVEL is 2.

Note: When running multi-threaded the numeric value component of the GMQ_TRACE_LEVEL environment variable only controls the first 40 threads used. Subsequent threads are traced at level 9.

Example trace

The example trace following shows typical trace output. It has been annotated and edited in order to illustrate the key features you might want to look for.

```
Trace for program  ---- MQSeries MQLSX ----  started at Tue Sep 22 09:31:47 1998

@(!)      ***** Code Level is 5.1 *****
   ! BuildDate Aug  7 1998
   ! Trace Level is 9+

************************************************************************
The head of the trace contains details of the MQLSX level, when the trace
was generated, when the MQLSX was built and the current trace level.
These can be important factors when resolving problems (eg. checking which
version of MQLSX is being used).                                        .
************************************************************************

(00001)@09:31:47.530
   -->xmq_xxxInitialize

************************************************************************
The number in brackets is the thread Id (in this case 00001 - thread 1)
and the number following the @ symbol is the time the trace entry was made
(in this case at 9:31 and 47 seconds). These times are based on the system
clock and cannot be relied upon for performance measurements.

Entries starting with --> indicate a call to the function named and should
have matching exit points indicated by <--. The number of dashes indicates
the calling depth of the function in the code. Every 2 dashes equates to one
level (in this case we are at the top level - function level 1).
The trace level is used to specify how many levels deep we wish to trace
function calls Note: The trace level is set via the GMQ_TRACE_LEVEL
environment variable.
************************************************************************

   ---->ObtainSystemCP
   ! Code page is 850

   <----ObtainSystemCP (rc= OK)
   ! XLAT_PATH is C:\MQM\MQLSX\CONV
   ! Successfully opened C:\MQM\MQLSX\CONV\GMQLCCS.TBL

   <--xmq_xxxInitialize (rc= OK)

   -->LSX: MainEntryPoint
   ! LSX: Version 2.0

   <--LSX: MainEntryPoint (rc= OK)
```

```
   -->LSX: MQLSX_MessageProc
   ! LSX: LSX_MSG_SETPATH received; library loaded from C:\NOTES\mqlsx.DLL

   <--LSX: MQLSX_MessageProc (rc= OK)

   -->LSX: MQLSX_MessageProc
   ! LSX: LSX_MSG_INITIALIZE received
   ! LSX: SUCCESS on ClassRegistration of MqQueueManager
   ! LSX: SUCCESS on ClassRegistration of MqGetMessageOptions
   ! LSX: Class MqMessage is using Unicode
   ! LSX: SUCCESS on ClassRegistration of MqMessage
   ! LSX: SUCCESS on ClassRegistration of MqPutMessageOptions
   ! LSX: SUCCESS on ClassRegistration of MqQueue
   ! LSX: SUCCESS on ClassRegistration of MqSession
   ! LSX: SUCCESS on ClassRegistration of MqProcess
   ! LSX: SUCCESS on ClassRegistration of MqDistributionList
   ! LSX: SUCCESS on ClassRegistration of MqDistributionListItem

   ---->LSX: RegisterCMQC
   <----LSX: RegisterCMQC (rc= OK)

   ---->LSX: RegisterCMQCFC
   <----LSX: RegisterCMQCFC (rc= OK)

   ---->LSX: RegisterIMQTYPE
   <----LSX: RegisterIMQTYPE (rc= OK)

   <--LSX: MQLSX_MessageProc (rc= OK)

*************************************************************************
The above lines show the MQLSX registering its internal classes and constants
with Lotus Notes. This enables us to write LotusScript code that can utilize
the features of MQSeries. Under normal circumstances you should never see
an error as this would indicate a problem with the MQLSX code or build.

Notice that during registration of the MQMessage class we tell LotusScript
that message data is to be held in unicode format (we will translate the
data to the appropriate character set when we send messages).
*************************************************************************

1.   -->LSX: Class entry point
2.   ! LSX: LSI_ADTMSG_CREATE received for class:MqSession
3.   ! LSX: Could not find MQSession for threadID = [1]
4.   ! LSX: >>> MEM >>> new: 0x16209b4
5.   ! LSX: Adding MQSession for threadID = [1]
6.   ! LSX: LotusScript >>>        Set [X] = new MqSession  [0x16209b4]
7.   <--LSX: Class entry point (rc= OK)

     -->LSX: ClassControl
8.   ! LSX: LSI_ADTMSG_ADDREF received for class:MqSession refCount = 1
     <--LSX: ClassControl (rc= OK)
```

```
************************************************************************
The above lines show what happens when the LotusScript creates a new MQSession
object:
1. We Call into the classes main function (top level function).
2. A message is sent from LotusScript telling the MQLSX what type of operation
   is being performed.
3. A session doesn't yet exist.
4. Allocate the memory for a new session.
5. Add the session to a list (as we have one session per thread of execution).
6. Indicate the LotusScript statement that was issued. Note: we are not
   actually passed the name of the LotusScript variable so we always use 'X'.
   Also the address of the allocated memory for the class is shown so that you
   can match up which session is which throughout the trace.
7. Shows the function exiting.
8. LotusScript keeps a reference count of how many instances of an object
   exist. We can use this to decide when to delete an object (eg. if it's
   reference count goes down to 0).
************************************************************************

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_EVENT_REG received for class:MqSession event = MQWARNING
   ! LSX: LotusScript >>>        On Event MQWARNING From MqSession[0x16209b4] Call [X]
   <--LSX: ClassControl (rc= OK)

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_EVENT_REG received for class:MqSession event = MQERROR
   ! LSX: LotusScript >>>        On Event MQERROR From MqSession[0x16209b4] Call [X]
   <--LSX: ClassControl (rc= OK)

************************************************************************
The above code shows the LotusScript program registering to Event handlers
for the session, an Error handler (MQERROR) and a warning handler (MQWARNING)
Therefore any errors that occur will call into the error event handler and
any warnings will call the warning event handler.
************************************************************************

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_METHOD received for class:MqSession; method:AccessQueueManager[3]
   ! LSX: LotusScript >>>        MQSession[0x16209b4].accessQueueManager("elmes")
   ! LSX: >>> MEM >>> new: 0x1620ad4

   ------>gmqdyn0a:MQCONN
   ! >>>Queue Manager Name...
         0000  65 6C 6D 65 73 00 00 00 00 00 00 00 00 00 00 00  : elmes...........
         0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  : ................
         --- 1 lines identical to above ---
   ! GMQDYN0A : About to try and find a dynamic library
```

```
   -------->EstablishEPs
   ! Using mqm

   <--------EstablishEPs (rc= 1)
   ! GMQDYN0A : About to go off to real MQCONN
   ! GMQDYN0A: Back from real MQCONN
   ! <<<Queue Manager Name...
         0000  65 6C 6D 65 73 00 00 00 00 00 00 00 00 00 00 00  : elmes...........
         0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  : ................
         --- 1 lines identical to above ---
   ! <<<HConn...
         0000  60 4B 38 01                                      :  K8.
   ! <<<Completion Code...
         0000  00 00 00 00                                      : ....
   ! <<<Reason Code...
         0000  00 00 00 00                                      : ....
   <------gmqdyn0a:MQCONN (rc= OK)

   <--LSX: ClassControl (rc= OK)

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_ADDREF received for class:MqQueueManager refCount = 1
   <--LSX: ClassControl (rc= OK)

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_EVENT_REG received for class:MqQueueManager event = MQWARNING
   ! LSX: LotusScript >>>        On Event MQWARNING From MqQueueManager[0x1620ad4] Call [X]
   <--LSX: ClassControl (rc= OK)

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_EVENT_REG received for class:MqQueueManager event = MQERROR
   ! LSX: LotusScript >>>        On Event MQERROR From MqQueueManager[0x1620ad4] Call [X]
   <--LSX: ClassControl (rc= OK)

**************************************************************************
The above code shows the LotusScript program accessing an MQSeries
Queue Manager (called 'elmes' in this case), via the accessQueueManager
method (which is a method call of the session we created earlier).
The accessQueueManager method automatically connects to the Queue Manager
via the MQCONN call. This is an actual MQSeries function and the trace shows
the MQLSX locating the MQSeries DLL and the corresponding entry point of the
MQCONN function.
**************************************************************************
```

```
************************************************************************

Trace lines removed for clarity

************************************************************************

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_METHOD received for class:MqMessage; method:WriteString[29]
   ! LSX: LotusScript >>>        MQMessage[0xef052c].writeString( ... )
   ! LSX: LotusScript >>>        MQMessage.writeString( ... )
         0000  73 00 74 00 72 00 69 00 6E 00 67 00             : s.t.r.i.n.g.
   ! LSX: Notes has passed us 6 (unicode characters)

   ---->ConvertStrFromDefault
   ! string before conv:
         0000  73 00 74 00 72 00 69 00 6E 00 67 00             : s.t.r.i.n.g.
   ! MQLSX/AX: WriteString instrlen = 12, outstrlen = 24

   ------>xmq_xcsConvertString
   ! fromCCSID:1200 toCCSID:437
   ! in length:12, out length:24

************************************************************************
The above lines show the LotusScript program performing a WriteString
operation on an MQMessage object (the word 'string' is being stored in the
message). LotusScript passes the characters to the MQLSX as unicode characters.
It is up to the MQLSX to convert these characters to the character set of the
system we will be sending the MQMessage to. In this case we are converting
from character set 1200 (unicode) to 437 (NT).
************************************************************************

   -------->xmq_OpenConversion

   ---------->xmq_xxxOpenConv

   ------------>xmq_xxxGetTable

   <------------xmq_xxxGetTable  (rc= OK)

   <----------xmq_xxxOpenConv (rc= OK)

   <--------xmq_OpenConversion (rc= OK)
   ! translated out length:6
   ! Data output
         0000  73 74 72 69 6E 67                               : string

   <------xmq_xcsConvertString (rc= OK)
   ! MQLSX/AX: CharacterSet conversion from 1200 to 437, rc = 0
   ! string after conv:
         0000  73 74 72 69 6E 67                               : string

   <----ConvertStrFromDefault (rc= OK)
   <--LSX: ClassControl (rc= OK)
```

```
**************************************************************************
The character set conversion tables are stored on disk. The above lines
show the character conversion using the appropriate table. Note: The tables
are typically stored in the C:\MQM\CONV directory.
**************************************************************************

   -->LSX: ClassControl
   ! LSX: LSI_ADTMSG_DROPREF received for class:MqQueueManager refCount = 0
   ! LSX: cleanup() called for class:MqQueueManager

   ------>gmqdyn0a:MQDISC
   ! >>>HConn...
         0000  60 4B 38 01                                        :  K8.
   ! GMQDYN0A : About to go off to real MQDISC
   ! GMQDYN0A: Back from real MQDISC
   ! <<<HConn...
         0000  FF FF FF FF                                        : ....
   ! <<<Completion Code...
         0000  00 00 00 00                                        : ....
   ! <<<Reason Code...
         0000  00 00 00 00                                        : ....

   <------gmqdyn0a:MQDISC (rc= OK)

   ! LSX: >>> MEM >>> delete: 0x17e08c4

   <--LSX: ClassControl (rc= OK)

   -->LSX: MQLSX_MessageProc
   ! LSX: LSX_MSG_TERMINATE received
   ! LSX: Removing MQSession for threadID = [1]

   <--LSX: MQLSX_MessageProc (rc= OK)

**************************************************************************
The above lines of trace show the LotusScript code terminating.
The queueManager object is deleted. With the object deleted, MQSeries
disconnects from the real Queue Manager. Finally, the MQSession object is
removed from the session list.
**************************************************************************
```

When your MQLSX script fails

When your MQLSX fails, a first failure symptom report is produced. This is detailed below.

First failure symptom report

Independently of the trace facility, for unexpected and internal errors, a first failure symptom report is produced.

This report is found in a file named GMQnnnnn.fdc, where nnnnn is the id of the Notes process that was running at the time. You find this file in the working directory from which you started Notes or the name of the path specified in the GMQ_TRACE_PATH environment variable.

Other sources of information

MQSeries puts error information in the Windows NT event log.

Common pitfalls

There are some problems that may be unexpected, depending on your experience of using LotusScript and MQSeries. The following are those identified so far. For example:

LotusScript events LotusScript treats events differently within the Notes Client and the Notes Server environments.

In the Notes Client environment, events are posted immediately, however within the Notes Server environment there is a delay in an event being posted. When you are using remote agents within Notes, this can give the impression that events are not always posted when you would expect them.

Uselsx failures Uselsx failures are most likely to occur because the libmqlsx shared library (which is a LotusScript extension) cannot be found. Check that this is in a directory where LotusScript expects to find LotusScript extensions.

Data conversion Your program can fail while it is trying to convert data on a read or write. See “Data conversion” on page 18 for more information.

Reason codes

The following reason codes can occur in addition to those documented for the MQSeries MQI. For further codes, refer to Chapter 6, “MQSeries constants” in the *MQSeries Application Programming Reference* and Appendix C, “Reason codes” in the *MQSeries Using C++* book.

Reason code	Explanation
MQRC_LIBRARY_LOAD_ERROR (6000)	One or more of the MQSeries libraries could not be loaded. Check that all MQSeries libraries are in the correct search path on the system you are using.
MQRC_CLASS_LIBRARY_ERROR (6001)	One of the MQSeries class library calls returned an unexpected ReasonCode / CompletionCode. Check the First Failure Symptom Report for details. Take note of the last method/property and class being used and inform IBM Support of the problem.
MQRC_STRING_LENGTH_TOO_BIG (6002)	A ReadString or WriteString call tried to read a string of more than 32000 characters (16000 characters if you are running the Win16 version). Find the ReadString call in your application and correct the call.
MQRC_WRITE_VALUE_ERROR (6003)	A write method has detected a data overflow. Correct the data passed to the Write method to ensure it is within the acceptable boundaries for the data type in question.
MQRC_REOPEN_EXCL_INPUT_ERROR (6100)	An open object does not have the correct OpenOptions and requires one or more additional options. An implicit re-open is required but closure has been prevented. Set the OpenOptions explicitly to cover all eventualities so that implicit re-opening is not required. Closure has been prevented because the queue is open for exclusive input and closure would present a window of opportunity for others to potentially gain access to the queue.
MQRC_REOPEN_TEMPORARY_Q_ERROR (6103)	An open object does not have the correct OpenOptions and requires one or more additional options. An implicit re-open is required but closure has been prevented. Set the OpenOptions explicitly to cover all eventualities so that implicit re-opening is not required. Closure has been prevented because the queue is a local queue of the definition type MQQDT_TEMPORARY_DYNAMIC, that would be destroyed by closure.
MQRC_ATTRIBUTE_LOCKED (6104)	An attempt has been made to change the value of an attribute of an object whilst that object is open. Certain attributes, such as AlternateUserId, cannot be changed whilst an object is open.
MQRC_CURSOR_NOT_VALID (6105)	The browse cursor for an open queue has been invalidated since it was last used by an implicit re-open. Set the OpenOptions explicitly to cover all eventualities so that implicit re-opening is not required.
MQRC_NULL_POINTER (6108)	A null pointer has been supplied where a non-null pointer is either required or implied. This denotes an internal consistency that should not occur.

Reason code	Explanation
MQRC_NO_CONNECTION_REFERENCE (6109)	The MQQueue object has lost its connection to the MQQueueManager. This will occur if the MQQueueManager is disconnected. Delete the MQQueue object.
MQRC_NO_BUFFER (6110)	No buffer is available. For an MQMessage object, one cannot be allocated, denoting an internal inconsistency in the object state that should not occur.
MQRC_BINARY_DATA_LENGTH_ERROR (6111)	The length of the binary data is inconsistent with the length of the target attribute. Zero is a correct length for all attributes. 24 is the correct length for a CorrelationId and for a MessageId. 32 is the correct length for an AccountingToken. Some properties must be supplied in full, such as CorrelationId.
MQRC_INSUFFICIENT_BUFFER (6113)	There is insufficient buffer space available after the data pointer to accommodate the request. This could be because the buffer cannot be resized.
MQRC_INSUFFICIENT_DATA (6114)	There is insufficient data after the data pointer to accommodate the read request. Reduce the buffer to the correct size and read the data again.
MQRC_DATA_TRUNCATED (6115)	Data has been truncated when copying from one buffer to another. This could be because the target buffer cannot be resized, or because there is a problem addressing one or other buffer, or because a buffer is being downsized with a smaller replacement.
MQRC_ZERO_LENGTH (6116)	A zero length has been supplied where a positive length is either required or implied.
MQRC_NEGATIVE_LENGTH (6117)	A negative length has been supplied where a zero or positive length is required.
MQRC_NEGATIVE_OFFSET (6118)	A negative offset has been supplied where a zero or positive offset is required.
MQRC_INCONSISTENT_OBJECT_STATE (6120)	There is an inconsistency between this object, which is open, and the referenced MQQueueManager object, which is not connected.

Dynamic loading

The shared library for MQSeries is dynamically loaded when your application calls the MQSession AccessQueueManager method of the MQQueueManager Connect method. If you have a problem on either of these calls (typically returning the reason code MQRC_LIBRARY_LOAD_ERROR) the following information may help.

Loading the MQLSX library

When you specify:

```
Uselsx "mqlsx"
```

in your LotusScript program, Notes uses standard system services to locate your MQLSX library (mqlsx.dll).

For example, on Windows NT it looks for an object called mqlsx.dll in the current working directory or on a directory in the search path specified by the PATH environment variable.

See the Lotus documentation for more information.

Tip: One way of identifying whether your search path is in error is to change the Uselsx to include the full path name; for example,

```
uselsx "c:\notes\mqlsx.dll"
```

Loading the MQSeries library

The MQLSX generates at various points within its code, standard API calls to MQSeries (MQCONN). These are “trapped” within the MQLSX code at entry points with the same names as the MQSeries entry points.

If this is the **first call** to MQSeries within the application, the MQLSX code tries to dynamically load the DLL (Dynamic Link Library) containing the real MQSeries code.

If the necessary object is found, the MQLSX detects and remembers the entry points of the real MQSeries functions. The “trapped” call is passed to the real function entry point.

After the initial call, subsequent calls to MQSeries are “trapped” in the MQLSX code and immediately passed to the remembered entry points.

Which MQSeries library to load?

On Windows NT the MQLSX dynamically detects and uses either the MQSeries server library (mqm.dll as supplied with either the MQSeries for Windows NT server) or the MQSeries 32-bit client library (mqic32.dll) or the MQSeries 16-bit client library (mqic.dll). The MQLSX searches for and uses these as follows:

1. mqm.dll
2. mqic32.dll
3. mqic.dll.

You can override this search order using the GMQ_MQ_LIB environment variable.

The GMQ_MQ_LIB environment variable

The use of GMQ_MQ_LIB is primarily in a development environment. In a production environment you would not normally need to set this as the MQLSX finds the appropriate MQSeries library automatically via the standard system dynamic load mechanisms.

GMQ_MQ_LIB enables you to have multiple copies of the MQSeries product installed (either at different levels, under test, or where you want to develop more than one application) on the same hardware. Setting this environment variable enables you to bypass the normal search routes.

You may also want to set the GMQ_MQ_LIB environment variable when you want to force the call to MQSeries to use the MQSeries client even though you have the MQSeries server library in your system path:

For example:

```
set GMQ_MQ_LIB=C:\MQM\BIN\MQIC32.DLL
```

Note: If you override the dll to be loaded, by using the GMQ_MQ_LIB environment variable, do not forget that this dll picks up other DLLs, which need to be on the PATH.

Chapter 6. MQLSX reference

This chapter describes the classes of the MQSeries link LotusScript Extension (MQLSX), developed for Lotus Notes. The classes enable you to write Notes applications that can access other applications running in your non-Notes environments, using MQSeries.

MQLSX objectives

The MQSeries link LotusScript Extension (MQLSX) is designed to:

- Provide an infrastructure to enable you to develop applications that integrate your Lotus Notes environment with your traditional transaction system applications and their data.
- Give you access to all the functions and features of the MQSeries API, permitting full interconnectivity to other MQSeries platforms.
- Conform to the normal conventions expected of a LotusScript extension.
- Enable you to take advantage of the following benefits provided by MQSeries:
  - Access to enterprise application logic, not just the data
  - Access to a wide variety of platforms
  - Queued entry into high throughput asynchronous messaging environments

The LotusScript MQSeries interface

The LotusScript interface to MQSeries is supplied as a LotusScript Extension module (LSX) that provides the following classes:

- MQDistributionList class
- MQDistributionListItem class
- MQGetMessageOptions class
- MQMessage class
- MQProcess class
- MQPutMessageOptions class
- MQQueue class
- MQQueueManager class
- MQSession class

In addition the MQLSX provides predefined LotusScript constants (such as MQFMT_NONE) needed to use the classes. The constants are a subset of those defined in the MQSeries C header files (cmqc*.h) with some additional MQLSX reason codes.

About MQLSX classes

This information should be read in conjunction with the *MQSeries Application Programming Reference* manual.

There are MQLSX classes called MQGetMessageOptions class, MQMessage class, MQProcess class, MQPutMessageOptions class, MQQueue class,

MQQueueManager class, MQSession class, MQDistributionList class, and MQDistributionListItem class.

The MQSession class provides a root object that contains the status of the last action performed on any of the MQLSX objects, see “How error handling works” on page 27 for more information.

The MQQueueManager, MQQueue, MQDistributionList, MQDistributionListItem, and MQProcess classes provide access to the underlying MQSeries objects. Methods or property accesses against these classes will in general result in calls being made across the MQSeries API.

The MQMessage, MQPutMessageOptions, and MQGetMessageOptions classes encapsulate the MQMD, MQPMO, and MQGMO data structures respectively, and are used to help you put messages to queues and retrieve messages from them.

Parameter passing

Parameters on method invocations are all passed by value, except where that parameter is an object, in which case it is a reference that is passed.

The class definitions provided list the Data Type for each parameter or property. If the LotusScript variable used is not of the required type, the value will be automatically converted to or from the required type - providing such a conversion is possible. This follows standard LotusScript conversion rules.

Many of the methods take fixed-length string parameters, or return a fixed-length character string. The conversion rules are as follows:

- If the user supplies a fixed length string of the wrong length, as an input parameter or a return value, the value is truncated or padded with trailing spaces as required.
- If the user supplies a variable length string of the wrong length as an input parameter, the value is truncated or padded with trailing spaces.
- If the user supplies a variable length string of the wrong length as a return value, the string is adjusted to the required length (because returning a value destroys the previous value in the string anyway).

Strings provided as input parameters may contain embedded nulls.

Object access methods

These methods do not relate directly to any single MQSeries call. Each of these methods create an object in which reference information is then held, followed by connecting or opening an MQSeries object:

- When a connection is made to a queue manager or a process object, it holds the object handle generated by MQSeries.
- When a queue is opened, it holds the connection handle generated by MQSeries.

These MQSeries attributes are explained in the *MQSeries Application Programming Reference manual.*

Errors

Syntactic errors on parameter passing are detected by LotusScript at compile time and runtime errors can be trapped using On Error.

The MQSeries LotusScript classes all contain two special read-only properties - ReasonCode and CompletionCode. These can be read at any time.

An attempt to access any other property, or to issue any method call could potentially generate an error.

If a property set or method invocation succeeds, then the owning object's ReasonCode and CompletionCode fields are set to MQRC_NONE and MQCC_OK respectively.

If the property access or method invocation does not succeed then appropriate error or warning codes are set in these fields.

MQSession class

This is the root class for the MQSeries link LotusScript Extension.

There is always only one MQSession object per LotusScript instance.

An attempt to create a second object creates a second reference to the original object.

Properties:

- CompletionCode property
- ReasonCode property
- ReasonName property

Methods:

- AccessGetMessageOptions method
- AccessMessage method
- AccessPutMessageOptions method
- AccessProcess method
- AccessQueueManager method
- ClearErrorCodes method
- ReasonCodeName method

LotusScript events:

- Mqerror
- Mqwarning

Creation:

New creates a new MQSession object reference.

Syntax:

Dim *mqsess* **As New MQSession** or **Set** *mqsess* = New MQSession

CompletionCode property

Read-only. Returns the MQSeries completion code set by the most recent method or property access issued against any MQSeries object.

It is reset to MQCC_OK when a call, other than a property Get, is made successfully against any MQLSX object.

An error event handler can inspect this property to diagnose the error, without having to know which object was involved.

Defined in: MQSession class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax: To get: *completioncode& = MQSession.***CompletionCode**

ReasonCode property

Read-only. Returns the reason code set by the most recent method or property access issued against any MQSeries object.

An error event handler can inspect this property to diagnose the error, without having to know which object was involved.

Defined in: MQSession class

Data Type: Long

Values: See the *MQSeries Application Programming Reference* and the additional MQLSX values listed under “Reason codes” on page 41

Syntax:
To get: *reasoncode& = MQSession.***ReasonCode**

ReasonName property

Read-only. Returns the symbolic name of the latest reason code. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQSession class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname& = MQsession.***ReasonName**

AccessGetMessageOptions method

Creates a new MQGetMessageOptions object.

Defined in: MQSession class

Syntax: To set: *gmo = MQSession.***AccessGetMessageOptions**

AccessMessage method

Creates a new MQMessage object.

Defined in: MQSession class

Syntax:
To set: *msg = MQSession.***AccessMessage**

AccessPutMessageOptions method

Creates a new MQPutMessageOptions object.

Defined in: MQSession class

Syntax:
To set: *MQSession.***AccessPutMessageOptions**

AccessQueueManager method

Creates a new MQQueueManager object and connects it to a real queue manager via the MQSeries client or server.

If successful it sets the MQQueueManager's ConnectionStatus to TRUE.

A queue manager can be connected to by at most one MQQueueManager object per LotusScript instance.

If the connection to the object fails, an error event is raised, the object's ReasonCode and CompletionCode are set, and the MQSession object's ReasonCode and CompletionCode are set.

Defined in: MQSession class

Syntax:
Set *qm = MQSession.***AccessQueueManager** *(Name$)*

Parameter: *Name$* String.Name of queue manager to be connected to.

AccessProcess method

Creates a new MQProcess object.

Defined in: MQSession class

Syntax:
To set: *Process = MQSession.***AccessProcess**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE.

Defined in: MQSession class

Syntax:
Call *MQSession.***ClearErrorCodes**

ReasonCodeName method

Returns the name of the reason code with the given numeric value. It is still useful to give clearer indications of error conditions to users. The name is still somewhat cryptic (for example, ReasonCodeName(2059) is **MQRC_Q_MGR_NOT_AVAILABLE**), so where possible errors should be caught and replaced with descriptive text appropriate to the application.

Defined in: MQSession class

Syntax:
*errname = MQSession.***ReasonCodeName** *(ReasonCode)*

MQQueueManager class

This represents a connection to a queue manager. The queue manager may be running locally (an MQSeries server) or remotely with access provided by the MQSeries client. An application must create an object of this class and connect it to a queue manager. When an object of this class is destroyed it is automatically disconnected from its queue manager.

Containment:

MQProcess and MQQueue objects are associated with this class.

Creation:

New creates a new MQQueueManager object and sets all the properties to initial values.

Syntax:

Dim *mgr* **As New MQQueueManager**

set *mgr* = **New MQQueueManager**

Alternatively, use the AccessQueueManager method of the MQSession class.

Properties:

- AlternateUserId property
- AuthorityEvent property
- BeginOptions property
- ChannelAutoDefinition property
- ChannelAutoDefinitionEvent property
- ChannelAutoDefinitionExit property
- CharacterSet property
- CloseOptions property
- CommandInputQueueName property
- CommandLevel property
- CompletionCode property
- ConnectionHandle property
- ConnectionStatus property
- ConnectOptions property
- DeadLetterQueueName property
- DefaultTransmissionQueueName property
- Description property
- DistributionLists property
- InhibitEvent property
- IsOpen property
- LocalEvent property
- MaximumHandles property
- MaximumMessageLength property
- MaximumPriority property
- MaximumUncommittedMessages property
- Name property
- ObjectHandle property
- PerformanceEvent property
- Platform property
- ReasonCode property

- ReasonName property
- RemoteEvent Property
- StartStopEvent property
- SyncPointAvailability property
- TriggerInterval property

Methods:

- AccessProcess method
- AccessQueue method
- AddDistributionList method
- Backout method
- Begin method
- ClearErrorCodes method
- Commit method
- Connect method
- Disconnect method

LotusScript Events:

- Mqerror
- Mqwarning

Creation:

New creates a new MQQueueManager object. If you do not want to connect to the default queue manager, you must set the name of the newly created queue manager before accessing any of the properties, other than those listed under Property Access. If you do not name the queue manager and access one of the properties outside the list, for example the Description property, the MQLSX will attempt to connect implicitly to the default queue manager.

A new MQQueueManager object can also be created by using the AccessQueueManager method on the MQSession object.

Syntax:

Dim *qmname* **As New MQQueueManager** or

Set *qmname* = **New MQQueueManager**

Property Access:

The following properties can be accessed at any time

- AlternateUserId
- CompletionCode
- ConnectionStatus
- ReasonCode
- ReasonCodeName

The remaining properties can be accessed only if the object is connected to a queue manager, and the user ID is authorized for inquire against that queue manager. If an alternate userid is set and the current userid is authorized to use it, the alternate user ID is checked for authorization for inquire instead.

If these conditions do not apply, the MQLSX will attempt to connect to the queue manager and open it for inquire automatically. If this is unsuccessful the call will set a CompletionCode of MQCC_FAILED and one of the following ReasonCodes:

- MQRC_CONNECTION_BROKEN
- MQRC_NOT_AUTHORIZED
- MQRC_QMGR_NAME_ERROR
- MQRC_QMGR_NOT_AVAILABLE

The Backout, Commit, Connect, and Disconnect methods set errors matching those set by the MQI calls MQBACK, MQCMIT, MQCONN, and MQDISC respectively.

AlternateUserId property

Read-Write. The alternate user ID to be used to validate access to the queue manager attributes.

This property may not be set if ConnectionStatus is TRUE.

Defined in: MQQueueManager class

Data Type: String of 12 characters

Syntax:
To get: *altuser$ = MQQueueManager*.**AlternateUserId**

To set: *MQQueueManager*.**AlternateUserId** = *altuser$*

AuthorityEvent property

Read-only. The MQI AuthorityEvent attribute. Controls whether authorization events are generated. If the value is set to MQEVR_ENABLED, an event is generated when unauthorized access to the queue manager is attempted.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *authevent& = MQQueueManager.*.**AuthorityEvent**

BeginOptions property

Read-Write. These are the options that apply to the Begin method.
Initially MQBO_NONE.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQBO_NONE

Syntax:
To get: *beginoptions&=MQQueueManager*.**BeginOptions**

To set: *MQQueueManager*.**BeginOptions**= *beginoptions&*

ChannelAutoDefinition property

Read-only. This controls whether automatic channel definition is permitted.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQCHAD_DISABLED
- MQCHAD_ENABLED

Syntax:
To get: *channelautodef&=MQQueueManager.***ChannelAutoDefinition**

ChannelAutoDefinitionEvent property

Read-only. This controls whether automatic channel definition events are generated.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *channelautodefevent&=MQQueueManager.***ChannelAutoDefinitionEvent**

ChannelAutoDefinitionExit property

Read-only. The name of the user exit used for automatic channel definition.

Defined in: MQQueueManager class

Data Type: String

Syntax:
To get: *channelautodefexit$= MQQueueManager.***ChannelAutoDefinitionExit$**

CharacterSet property

Read-only. The MQI CodedCharSetId attribute. This identifies the character set used by the queue manager for all character strings defined in the MQI. This includes the names of objects, and queue creation date and time. It does not include application data carried in the message.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *characterset& = MQQueueManager.***CharacterSet**

CloseOptions property

Read-write. Options used to control what happens when the distribution list is closed. The initial value is MQCO_NONE.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQCO_NONE

Syntax:
To get: *closeopt& = MQQueueManager*.**CloseOptions**

To set: *MQQueueManager*.**CloseOptions** = *closeopt&*

CommandInputQueueName property

Read-only. The MQI CommandInputQName attribute. This is the name of the queue to which applications, if authorized, can send commands.

Defined in: MQQueueManager class

Data Type: String of 48 characters

Syntax:
To get: *commandinputqname$ = MQQueueManager*.**CommandInputQueueName**

CommandLevel property

Read-only. Returns the version and level of the MQSeries queue manager implementation (MQI CommandLevel attribute)

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *level& = MQQueueManager*.**CommandLevel**

CompletionCode property

Read-only. Returns the completion code set by the last method or property access issued against the object.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = MQQueueManager*.**CompletionCode**

ConnectionHandle property

Read-only. The connection handle for the MQSeries queue manager.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *hconn& = MQQueueManager.***ConnectionHandle**

ConnectionStatus property

Read-only. Indicates if the object is connected to its queue manager or not.

Defined in: MQQueueManager class

Data Type: Long

Values:

- TRUE (-1)
- FALSE

Syntax:
To get: *status& = MQQueueManager.***ConnectionStatus**

ConnectOptions property

Read-Write. These are the options that apply to the Connect method. Initially MQCNO_NONE.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQCNO_STANDARD_BINDING
- MQCNO_FASTPATH_BINDING
- MQCNO_NONE

Syntax:
To get: *connectoptions&=MQQueueManager.***ConnectOptions**

To set: *MQQueueManager***.ConnectOptions**= *connectoptions&*

DeadLetterQueueName property

Read-only. The MQI DeadLetterQName attribute. This is the name of a queue defined on the local queue manager. Messages are sent to this queue if they cannot be routed to their correct destination.

Defined in: MQQueueManager class

Data Type: String of 48 characters

Syntax:
To get: *dlqname$ = MQQueueManager.***DeadLetterQueueName**

DefaultTransmissionQueueName property

Read-only. The MQI DefXmitQName attribute. This is the name of the transmission queue that holds messages prior to them being sent to another queue manager, if no other transmission queue is specified.

Defined in: MQQueueManager class

Data Type: String of 48 characters

Syntax:
To get: *defxmitqname$ = MQQueueManager.***DefaultTransmissionQueueName**

Description property

Read-only. The MQI QMgrDesc attribute. Use this property to hold a descriptive commentary. The content of this property has no significance to the queue manager. It must not contain any null characters; it is padded with blanks when necessary.

Defined in: MQQueueManager class

Data Type: String of 64 characters

Syntax:
To get: *description$ = MQQueueManager.***Description**

DistributionLists property

Read-Only. This is the capability of the queue manager to support distribution lists.

Defined in: MQQueueManager class

Data Type: Boolean (2-bytes)

Values:

- TRUE (-1)
- FALSE (0)

Syntax:
To get: *distributionlists=MQQueueManager.***DistributionLists**

InhibitEvent property

Read-only. The MQI InhibitEvent attribute. This determines whether Inhibit Get and Inhibit Put events are generated.

If it is set to MQEVR_ENABLED, an event is generated when the Get method is used against a queue that prevents any messages being removed from it. Similarly, an event is generated when the Put method is used against a queue that prevents any messages being placed on it.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *inhibevent& = MQQueueManager*.**InhibitEvent**

IsConnected property

Read-only. Indicates if the object is connected to its queue manager or not. This property is exactly the same as the ConnectStatus property.

Defined in: MQQueueManager class

Data Type: Long

Values:

- TRUE (-1)
- FALSE

Syntax:
To get: *status& = MQQueueManager*.**IsConnected**

IsOpen property

Read-only. A value that indicates whether the distribution list is currently open.

Defined in: MQQueueManager class

Data Type: Boolean (2-bytes)

Values:

- TRUE (-1)
- FALSE (0)

Syntax:
To get: *IsOpen = MQQueueManager*.**IsOpen**

LocalEvent property

Read-only. The MQI LocalEvent attribute. This determines whether local events are generated. A local event is generated when an application is unable to access a local queue. For more information see the *MQSeries Programmable System Management* manual.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *localevent& = MQQueueManager*.**LocalEvent**

MaximumHandles property

Read-only. The MQI MaxHandles attribute. This is the maximum number of open handles that any one task can have at the same time.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *maxhandles& = MQQueueManager*.**MaximumHandles**

MaximumMessageLength property

Read-only. The MQI MaxMsgLength Queue Manager attribute. This is the maximum length of a message, in bytes, that the queue manager can handle.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *maxmessagelength& = MQQueueManager***.MaximumMessageLength**

MaximumPriority property

Read-only. The MQI MaxPriority attribute. This is the maximum message priority supported by the queue manager, zero being the lowest.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *maxpriority& = MQQueueManager*.**MaximumPriority**

MaximumUncommittedMessages property

Read-only. The MQI MaxUncommittedMsgs attribute. This is the maximum number of uncommitted messages that can exist within a unit of work.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *maxuncommitted& = MQQueueManager.***MaximumUncommittedMessages**

Name property

Read-write. The MQI QMgrName attribute. This is the name of the queue manager to which an application is connected.

This property cannot be written once the MQQueueManager is Connected.

Defined in: MQQueueManager class

Data Type: String of 48 characters

Syntax:
To get: *name$ = MQQueueManager.***Name**

To set: *MQQueueManager.***Name** = *name$*

ObjectHandle property

Read-only. The Object handle for the MQSeries queue object.

Defined in: MQQueueManager class

Data Type Long

Syntax:
To get: *hobj& = MQQueueManager.***ObjectHandle**

Note: Required in MQAX only (not LotusScript) for MS Transaction Server

PerformanceEvent property

Read-only. The MQI PerformanceEvent attribute. This determines whether performance events are generated. For more information see the *MQSeries Programmable System Management* manual.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *perfevent& = MQQueueManager.***PerformanceEvent**

Platform property

Read-only. The MQI Platform attribute. This is the platform that the queue manager is running on.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQPL_AIX
- MQPL_MVS
- MQPL_OS2
- MQPL_OS400
- MQPL_UNIX
- MQPL_WINDOWS_NT

Syntax:
To get: *platform& = MQQueueManager.***Platform**

ReasonCode property

Read-only. Returns the reason code set by the last method or property access issued against the object.

Defined in: MQQueueManager class

Data Type: Long

Values: See the *MQSeries Application Programming Reference* manual

Syntax:
To get: *reasoncode& = MQQueueManager.***ReasonCode**

ReasonName property

Read-only. Returns the symbolic name of the latest reason code. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQQueueManager class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname& = MQQueueManager.***ReasonName**

RemoteEvent property

Read-only. The MQI RemoteEvent attribute. This controls whether or not remote events are generated. A remote event is raised when an application cannot access a queue on another queue manager. For more information see the *MQSeries Programmable System Management* manual.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *remoteevent& = MQQueueManager*.**RemoteEvent**

StartStopEvent property

Read-only. The MQI StartStopEvent attribute. This controls whether start and stop events are raised. A start event is raised when a queue manager is started. A stop event is raised when a request is made for a queue manager to stop or quiesce. For more information see the *MQSeries Programmable System Management* manual.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *strstpevent& = MQQueueManager*.**StartStopEvent**

SyncPointAvailability property

Read-only. The MQI SyncPoint attribute. This indicates whether or not the queue manager supports units or work and syncpointing when you use the Put and Get methods in the MQQueue class.

Defined in: MQQueueManager class

Data Type: Long

Values:

- MQSP_AVAILABLE
- MQSP_NOT_AVAILABLE

Syntax:
To get: *syncpointavailability& = MQQueueManager*.**SyncPointAvailability**

TriggerInterval property

Read-only. The MQI TriggerInterval attribute. This is a time interval (in milliseconds) used to restrict the number of trigger messages that are generated. This is only relevant when the value of the MQQueue class property, TriggerType, is MQTT_FIRST.

Defined in: MQQueueManager class

Data Type: Long

Syntax:
To get: *trigint& = MQQueueManager*.**TriggerInterval**

AccessProcess method

Creates a new MQProcess object and associates it with this MQQueueManager object. It sets the name property and the alternate user ID of the MQProcess object, and attempts to open it for inquire.

If you do not want to use alternate user ID validation, set this parameter to "".

Defined in: MQQueueManager class

Syntax:
Set *process = MQQueueManager*.**AccessProcess** *(Name$, AlternateUserId$)*

Parameters: *Name$*

String. Name of MQSeries process definition.

AlternateUserId$

String. The alternate user ID to validate access to the process object.

AccessQueue method

Creates a new MQQueue object and associates it with this MQQueueManager object. It sets the Name, OpenOptions, DynamicQueueName, and AlternateUserId properties of the MQQueue object to the values provided, and attempts to open it.

If the open is unsuccessful the call fails. An error event is raised against the object, the object's ReasonCode and CompletionCode are set, and the MQSession ReasonCode and CompletionCode are set.

All parameters are mandatory, but DynamicQueueName, QueueManagerName, and AlternateUserId may be set to the default of "" if they are not needed.

OpenOptions control the operations that can be performed on a queue. The OpenOption MQOO_INQUIRE is optional, it is automatically added to the options your application supplies. The options are listed in the *MQSeries Application Programming Reference* , under the MQOPEN call. They include:

- MQOO_INPUT_SHARED - allowing more than one application to get messages from this queue, where they also open the queue using the MQOO_INPUT_SHARED option.
- MQOO_INPUT_EXCLUSIVE - gives your application exclusive access to a queue.
- MQOO_SET - enables you to change the attributes of the queue after you have opened it.

Set the QueueManagerName to "" if the queue to be opened is local. Otherwise, it should be set to the name of the remote queue manager that owns the queue, and an attempt is made to open a local definition of the remote queue.

See the *MQSeries Application Programming Reference* for more information on remote queue name resolution and queue manager aliasing.

If the Name property is set to a model queue name, specify the name of the dynamic queue to be created in the DynamicQueueName$ parameter. If the value provided in the DynamicQueueName$ parameter is "", the value set into the queue object and used on the open call is "AMQ.*".

See the *MQSeries Application Programming Reference* for more information on naming dynamic queues.

Defined in: MQQueueManager class

Syntax:
Set *queue = MQQueueManager*.**AccessQueue** *(Name$, OpenOptions&, QueueManagerName$, DynamicQueueName$, AlternateUserId$)*

Parameter: *Name$*

String. Name of MQSeries queue.

OpenOptions&

Long. Options to be used when queue is opened.

See the *MQSeries Application Programming Reference* manual for more information.

QueueManagerName$

String. Name of the queue manager that owns the queue to be opened. A value of "" implies the queue manager is local.

DynamicQueueName$

String. The name assigned to the dynamic queue at the time the queue is opened when the Name$ parameter is specified as a model queue.

AlternateUserId$

String. The alternate user ID used to validate access when opening the queue.

AddDistributionList method

Creates a new MQDistributionList object and sets the connection reference to the owning queue manager.

Defined in: MQQueueManager class

Syntax:
Set *distributionlist = MQQueueManager*.**AddDistributionList**

Backout method

Backs out any uncommitted message puts and gets that have occurred as part of a unit of work since the last syncpoint.

Defined in: MQQueueManager class

Syntax:
Call *MQQueueManager*.**Backout**

Begin method

Begins a unit of work. The begin options affect the behavior of this method.

Defined in: MQQueueManager class

Syntax
Call *MQQueueManager*.**Begin**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQQueueManager class and the MQSession class.

Defined in: MQQueueManager class

Syntax:
Call *MQQueueManager*.**ClearErrorCodes**

Commit method

Commits any message puts and gets that have occurred as part of a unit of work since the last syncpoint.

Defined in: MQQueueManager class

Syntax:
Call *MQQueueManager*.**Commit**

Connect method

Connects the MQQueueManager object to a real queue manager via the MQSeries client or server.

Sets ConnectionStatus to TRUE.

A maximum of one MQQueueManager object per LotusScript instance is allowed to connect to a queue manager.

Defined in: MQQueueManager class

Syntax:
Call *MQQueueManager*.**Connect**

Disconnect method

Disconnects the MQQueueManager object from the queue manager.

Sets ConnectionStatus to FALSE.

All queue objects associated with the MQQueueManager object are made unusable and cannot be re-opened.

Any uncommitted changes (message puts and gets) are committed.

Defined in: MQQueueManager class

Syntax:
Call *MQQueueManager*.**Disconnect**

MQQueue class

This represents a connection to an MQSeries queue. This connection is provided by an associated MQQueueManager object. When an object of this class is destroyed it is automatically closed.

Containment:

Contained by the MQQueueManager class.

Properties:

- AlternateUserId property
- BackoutRequeueName property
- BackoutThreshold property
- BaseQueueName property
- CloseOptions property
- CompletionCode property
- CreationDateTime property
- CurrentDepth property
- DefaultInputOpenOption property
- DefaultPersistence property
- DefaultPriority property
- DefinitionType property
- DepthHighEvent property
- DepthHighLimit property
- DepthLowEvent property
- DepthLowLimit property
- DepthMaximumEvent property
- Description property
- DynamicQueueName property
- HardenGetBackout property
- InhibitGet property
- InhibitPut property
- InitiationQueueName property
- MaximumDepth property
- MaximumMessageLength property
- MessageDeliverySequence property
- Name property
- OpenInputCount property
- OpenOptions property
- OpenOutputCount property
- OpenStatus property
- ProcessName property
- QueueManagerName property
- QueueType property
- ReasonCode property
- ReasonName property
- RemoteQueueManagerName property
- RemoteQueueName property
- RetentionInterval property
- Scope property
- ServiceInterval property
- ServiceIntervalEvent property
- Shareability property

- TransmissionQueueName property
- TriggerControl property
- TriggerData property
- TriggerDepth property
- TriggerMessagePriority property
- TriggerType property
- Usage property

Methods:

- ClearErrorCodes method
- Open method
- Close method
- Get method
- Put method
- GetConnectionReference method
- SetConnectionReference method

LotusScript Events:

- Mqerror
- Mqwarning

Creation:

Use the AccessQueue method from the MQQueueManager class.

Property Access

If the queue object is not connected to a queue manager, you can read the following properties:

- AlternateUserId
- CompletionCode
- Name
- OpenOptions
- OpenStatus
- ReasonCode

and you can write to:

- CloseOptions

If the queue object is connected to a queue manager, you can read all the properties.

Note: Reading a property not listed above, such as TriggerControl, will cause an implicit connection to the underlying queue manager.

Queue Attribute properties:

Properties not listed in the previous section are all attributes of the underlying MQSeries queue. They can be accessed only if the object is connected to a queue manager, and the user's user ID is authorized for Inquire or Set against that queue. If an alternate user ID is set and the current user ID is authorized to use it, then the alternate user ID is checked for authorization instead.

The property must be an appropriate property for the given QueueType (see the *MQSeries Application Programming Reference*).

If these conditions do not apply, the property access will set a CompletionCode of MQCC_FAILED and one of the following ReasonCodes:

- MQRC_CONNECTION_BROKEN
- MQRC_NOT_AUTHORIZED
- MQRC_QUEUE_MGR_NAME_ERROR
- MQRC_QUEUE_MGR_NOT_CONNECTED
- MQRC_SELECTOR_NOT_FOR_TYPE (CompletionCode is MQCC_WARNING)

Opening a queue

The only way to create an MQQueue object is by using the MQQueueManager AccessQueue method, unless an implicit connection has taken place. The MQQueue object remains open (OpenStatus=TRUE) until it is deleted. The value of the MQQueue CloseOptions property controls the behavior of the close operation that takes place when the MQQueue object is deleted.

The MQQueueManager AccessQueue method opens the queue using the OpenOptions parameter with the automatic addition of the MQOO_INQUIRE value. MQSeries validates the OpenOptions against the user authorization as part of the open queue process.

Tip: Check that the OpenOptions are appropriate for the actions to be performed on a queue (for example, get or put). In certain circumstances, if the OpenOptions are insufficient, the MQLSX attempts to close and reopen the queue with additional OpenOptions:

- A property get is always allowed, because the OpenOptions always include MQOO_INQUIRE.
- If a queue property set is attempted and the OpenOptions do not include MQOO_SET, the queue is closed and reopened with MQOO_SET added to the OpenOptions.
- If a put is attempted and the OpenOptions do not include MQOO_OUTPUT, the queue is closed and reopened with MQOO_OUTPUT added to the OpenOptions.
- If a put is attempted and the MQPMO options include MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT, but the OpenOptions do not permit this, the queue is closed and reopened with MQOO_SET_IDENTITY_CONTEXT or MQOO_SET_ALL_CONTEXT, as appropriate.
- If a get is attempted and the OpenOptions do not include any of the MQOO_INPUT_* options, the queue is closed and reopened with MQOO_INPUT_AS_Q_DEF added to the OpenOptions.
- If a get is attempted and the MQGetMessageOptions request MQGMO_BROWSE_*, but the OpenOptions do not include MQOO_BROWSE, the queue is closed and reopened with MQOO_BROWSE added to the OpenOptions.

Note: A queue will not be reopened if it is a temporary dynamic queue, or the queue is already open for exclusive input. In these cases, the method call to reopen the queue will fail with either:
MQRC_REOPEN_TEMPORARY_Q_ERROR or
MQRC_REOPEN_EXCL_INPUT_ERROR, as appropriate.

If a queue is opened for Browse and a reopen occurred for one of the reasons listed, subsequent attempts to do a get with one of the following:

- MQGMO_BROWSE_NEXT
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_MSG_UNDER_CURSOR

will be rejected with MQRC_CURSOR_NOT_VALID.

AlternateUserId property

Read-only. The alternate user ID used to validate access to the queue when it was opened.

Defined in: MQQueue class

Data Type: String of 12 characters

Syntax:
To get: *altuser$ = MQQueue.***AlternateUserId**

BackoutRequeueName property

Read-only. The MQI BackOutRequeueQName attribute. This is the name of a queue that an application can use to put a message that is causing a unit of work to fail.

The message concerned is identified by the value of its BackoutCount property, which is incremented each time it causes a unit of work to fail. The application can test for the value held in the BackoutCount property to be greater than the value held in the BackoutThreshold property, and move the message to the queue named in the BackoutRequeueName property.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *backoutrequeuename$ = MQQueue.***BackoutRequeueName**

BackoutThreshold property

Read-only. The MQI BackoutThreshold attribute. This property is provided for an application to use. It is used in conjunction with the BackoutRequeueName and BackoutCount (this is a property of the MQSeries message). The value held in this property is available for comparison with the BackoutCount value, by an application, to enable the application to remove a problem message that is part of a unit of work.

Defined in: MQQueue class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *backoutthreshold& = MQQueue.***BackoutThreshold**

BaseQueueName property

Read-only. The queue name to which the alias resolves.

Valid only for alias queues.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *baseqname$ = MQQueue.***BaseQueueName**

CloseOptions property

Read-Write. Options used to control what happens when the queue is closed.

Defined in: MQQueue class

Data Type: Long

Values:

- MQCO_NONE
- MQCO_DELETE
- MQCO_DELETE_PURGE

MQCO_DELETE and MQCO_DELETE_PURGE are valid only for dynamic queues.

Syntax:
To get: *closeopt& = MQQueue.***CloseOptions**

To set: *MQQueue.***CloseOptions** = *closeopt&*

CompletionCode property

Read-only. Returns the completion code set by the last method or property access issued against the object.

Defined in: MQQueue class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = MQQueue.***CompletionCode**

CreationDateTime property

Read-only. Date and time this queue was created.

Defined in: MQQueue class

Data Type: Variant of type 7 (date/time) or EMPTY

Syntax:
To get: *datetime = MQQueue*.**CreationDateTime**

CurrentDepth property

Read-only. The MQI CurrentQDepth attribute. The number of messages currently on the queue.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *currentdepth& = MQQueue*.**CurrentDepth**

DefaultInputOpenOption property

Read-only. The MQI DefInputOpenOption attribute. This controls the way that the queue is opened if the OpenOption specifies MQOO_INPUT_AS_Q_DEF.

Defined in: MQQueue class

Data Type: Long

Values:

- MQOO_INPUT_EXCLUSIVE
- MQOO_INPUT_SHARED

Syntax:
To get: *defaultinop& = MQQueue*.**DefaultInputOpenOption**

DefaultPersistence property

Read-only. The MQI DefPersistence attribute. The default persistence for messages on a queue.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *defpersistence& = MQQueue*.**DefaultPersistence**

DefaultPriority property

Read-only. The MQI DefPriority attribute. The default priority for messages on a queue.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *defpriority& = MQQueue*.**DefaultPriority**

DefinitionType property

Read-only. The MQI DefinitionType attribute. This describes the type of queue. A model queue is needed to create a dynamic queue. A queue of type MQQDT_PREDEFINED is a permanent queue.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQDT_PREDEFINED
- MQQDT_PERMANENT_DYNAMIC
- MQQDT_TEMPORARY_DYNAMIC

Syntax:
To get: *deftype& = MQQueue*.**DefinitionType**

DepthHighEvent property

Read-only. The MQI QDepthHighEvent attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *depthhighevent& = MQQueue*.**DepthHighEvent**

DepthHighLimit property

Read-only. The MQI QDepthHighLimit attribute.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *depthhighlimit& = MQQueue*.**DepthHighLimit**

DepthLowEvent property

Read-only. The MQI QDepthLowEvent attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *depthlowevent& = MQQueue*.**DepthLowEvent**

DepthLowLimit property

Read-only. The MQI QDepthLowLimit attribute.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *depthlowlimit& = MQQueue*.**DepthLowLimit**

DepthMaximumEvent property

Read-only. The MQI QDepthMaxEvent attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQEVR_DISABLED
- MQEVR_ENABLED

Syntax:
To get: *depthmaximumevent& = MQQueue*.**DepthMaximumEvent**

Description property

Read-only. A description of the queue.

Defined in: MQQueue class

Data Type: String of 64 characters

Syntax:
To get: *description$ = MQQueue*.**Description**

DynamicQueueName property

Read-write, read-only when the queue is open.

This controls the dynamic queue name used when a model queue is opened. It may be set with a wildcard by the user either as a property set (only when the queue is closed) or as a parameter to QueueManager.AccessQueue(..).

The actual name of the dynamic queue is found by querying Queue.Name.

Defined in: MQQueue class

Data Type: String

Values: Any valid MQSeries queue name.

Syntax:
To set: *MQQueue.***DynamicQueueName** = *dynamicQueuename&*

To get: *dynamicqueuename& = MQQueue*.**DynamicQueueName**

HardenGetBackout property

Read-only. Whether to maintain an accurate backout count.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQA_BACKOUT_HARDENED
- MQQA_BACKOUT_NOT HARDENED

Syntax:
To get: *hardengetback& = MQQueue*.**HardenGetBackout**

InhibitGet property

Read-write. The MQI InhibitGet attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQA_GET_INHIBITED
- MQQA_GET_ALLOWED

Syntax:
To get: *getstatus& = MQQueue*.**InhibitGet**

To set: *MQQueue*.**InhibitGet** = *getstatus&*

InhibitPut property

Read-write. The MQI InhibitPut attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQA_PUT_INHIBITED
- MQQA_PUT_ALLOWED

Syntax:
To get: *putstatus& = MQQueue*.**InhibitPut**

To set: *MQQueue*.**InhibitPut** = *putstatus&*

InitiationQueueName property

Read-only. Name of initiation queue.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *initqname$ = MQQueue*.**InitiationQueueName**

IsOpen property

Read-only. Indicates if the queue is Opened or not. Initial value is TRUE after AccessQueue method. This property is exactly the same as the OpenStatus property.

Defined in: MQQueue class

Data Type: Long

Values:

- TRUE (-1)
- FALSE

Syntax:
To get: *status& = MQQueue*.**IsOpen**

MaximumDepth property

Read-only. Maximum queue depth.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *maxdepth& = MQQueue*.**MaximumDepth**

MaximumMessageLength property

Read-only. The MQI MaxMsgLength attribute. This is the maximum length message, in bytes, that this queue will accept.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *maxmlength& = MQQueue*.**MaximumMessageLength**

MessageDeliverySequence property

Read-only. Message delivery sequence.

Defined in: MQQueue class

Data Type: Long

Values:

- MQMDS_PRIORITY
- MQMDS_FIFO

Syntax:
To get: *messdelseq& = MQQueue*.**MessageDeliverySequence**

Name property

Read-only. The MQI QName attribute. This is the name of a queue defined on the local queue manager.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *name$ = MQQueue*.**Name**

OpenInputCount property

Read-only. Number of opens for input.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *openincount& = MQQueue*.**OpenInputCount**

OpenOptions property

Read-only. These are the options for which the queue is initially opened (specified in the AccessQueue method in the MQQueueManager class). However, these may be extended if an implicit re-open is performed.

Defined in: MQQueue class

Data Type: Long

Values: See the *MQSeries Application Programming Reference* under the MQOPEN call

Syntax:
To get: *openopt& = Queue*.**OpenOptions**

OpenOutputCount property

Read-only. Number of opens for output.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *openoutcount& = MQQueue*.**OpenOutputCount**

OpenStatus property

Read-only. Indicates if the queue is Opened or not. Initial value is TRUE after AccessQueue method.

Defined in: MQQueue class

Data Type: Long

Values:

- TRUE (-1)
- FALSE

Syntax:
To get: *status& = MQQueue*.**OpenStatus**

ProcessName property

Read-only. The MQI ProcessName attribute.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *procname$ = MQQueue*.**ProcessName**

QueueManagerName property

Read-write. The MQSeries queue manager name.

Defined in: MQQueue class

Data Type: String

Syntax:
To get: *String$ = MQQueue*.**QueueManagerName**

To set: *MQQueue*.**QueueManagerName** *= String$*

QueueType property

Read-only. The MQI QType attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQT_ALIAS
- MQQT_LOCAL
- MQQT_MODEL
- MQQT_REMOTE

Syntax:
To get: *queuetype& = MQQueue*.**QueueType**

ReasonCode property

Read-only. Returns the reason code set by the last method or property access issued against the object.

Defined in: MQQueue class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasoncode& = Queue*.**ReasonCode**

ReasonName property

Read-only. Returns the symbolic name of the latest reason code. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQQueue class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname& = MQQueue*.**ReasonName**

RemoteQueueManagerName property

Read-only. Name of remote queue manager.

Valid for remote queues only.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *remqmanname$ = MQQueue*.**RemoteQueueManagerName**

RemoteQueueName property

Read-only. The name of the queue as it is known on the remote queue manager.

Valid for remote queues only.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *remqname$ = MQQueue*.**RemoteQueueName**

RetentionInterval property

Read-only. The period of time for which the queue should be retained.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *retinterval& = MQQueue*.**RetentionInterval**

Scope property

Read-only. Controls whether an entry for this queue also exists in a cell directory.

Defined in: MQQueue class

Data Type: Long

Values:

- MQSCO_Q_MGR
- MQSCO_CELL

Syntax:
To get: *scope& = MQQueue*.**Scope**

ServiceInterval property

Read-only. The MQI QServiceInterval attribute.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *serviceinterval& = MQQueue*.**ServiceInterval**

ServiceIntervalEvent property

Read-only. The MQI QServiceIntervalEvent attribute.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQSIE_HIGH
- MQQSIE_OK
- MQQSIE_NONE

Syntax:
To get: *serviceintervalevent& = MQQueue*.**ServiceIntervalEvent**

Shareability property

Read-only. Queue shareability.

Defined in: MQQueue class

Data Type: Long

Values:

- MQQA_SHAREABLE
- MQQA_NOT SHAREABLE

Syntax:
To get: *shareability& = MQQueue*.**Shareability**

TransmissionQueueName property

Read-only. Transmission queue name.

Valid for remote queues only.

Defined in: MQQueue class

Data Type: String of 48 characters

Syntax:
To get: *transqname$ = MQQueue*.**TransmissionQueueName**

TriggerControl property

Read-write. Trigger control.

Defined in: MQQueue class

Data Type: Long

Values:

- MQTC_OFF
- MQTC_ON

Syntax:
To get: *trigcontrol& = MQQueue*.**TriggerControl**

To set: *MQQueue*.**TriggerControl** = *trigcontrol&*

TriggerData property

Read-write. Trigger data.

Defined in: MQQueue class

Data Type: String of 64 characters

Syntax:
To get: *trigdata$ = MQQueue*.**TriggerData**

To set: *MQQueue*.**TriggerData** = *trigdata$*

TriggerDepth property

Read-write. The number of messages that have to be on the queue before a trigger message is written.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *trigdepth& = MQQueue*.**TriggerDepth**

To set: *MQQueue*.**TriggerDepth** = *trigdepth&*

TriggerMessagePriority property

Read-write. Threshold message priority for triggers.

Defined in: MQQueue class

Data Type: Long

Syntax:
To get: *trigmesspriority& = MQQueue*.**TriggerMessagePriority**

To set: *MQQueue*.**TriggerMessagePriority** = *trigmesspriority&*

TriggerType property

Read-write. Trigger type.

Defined in: MQQueue class

Data Type: Long

Values:

- MQTT_NONE
- MQTT_FIRST
- MQTT_EVERY
- MQTT_DEPTH

Syntax:
To get: *trigtype& = MQQueue*.**TriggerType**

To set: *MQQueue*.**TriggerType** = *Trigtype&*

Usage property

Read-only. Indicates what the queue is used for.

Defined in: MQQueue class

Data Type: Long

Values:

- MQUS_NORMAL
- MQUS_TRANSMISSION

Syntax:
To get: *usage& = MQQueue.***Usage**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQQueue class and the MQSession class.

Defined in: MQQueue class

Syntax:
Call *MQQueue.***ClearErrorCodes**

Close method

Closes a queue using the current values of CloseOptions.

Defined in: MQQueue class

Syntax:
Call *MQQueue.***Close()**

Get method

Retrieves a message from the queue.

This method takes an MQMessage object as a parameter. It uses some of the fields in this object's MQMD as input parameters - in particular the MessageId(MsgId) and CorrelationId(CorrelId), so it is important to ensure that these are set as required see the *MQSeries Application Programming Reference* manual for details).

If the method succeeds then the MQMD and Message Data portions of the MQMessage object are completely replaced with the MQMD and Message Data from the incoming message. The MQMessage control properties are set as follows:

- MessageLength is set to length of the MQSeries Message
- **DataLength** is set to length of the MQSeries Message
- **DataOffset** is set to zero

Defined in: MQQueue class

Syntax:
Call *MQQueue*.**Get** *(Message , GetMsgOptions&)*

Parameters: *MQMessage*

MQMessage Object representing message to be retrieved.

GetMsgOptions&

MQGetMessageOptions object to control the get operation.

GetConnectionReference method

The Queue Manager object associated with this Queue.

Defined in: MQQueue class

Data Type: MQQueueManager

Syntax:
queuemanager = MQQueue.**GetconnectionReference**

Open method

Opens a queue using the current values of:

1. QueueName
2. QueueManagerName
3. AlternateUserId
4. DynamicQueueName

Defined in: MQQueue class

Syntax:
Call *MQQueue*.**Open()**

SetConnectionReference property

Sets the Queue Manager object associated with this Queue.

Defined in: MQQueue class

Data Type: MQQueueManager

Syntax:
Call *MQQueue*.**SetConnectionReference** *(queuemanager)*

Put method

Places a message onto the queue.

This method takes an MQMessage object as a parameter. The Message Descriptor (MQMD) properties of this object may be altered as a result of this method. The values they have immediately after this method has run are the values that were put onto the MQSeries queue.

Modifications to the MQMessage object after the Put has completed do not affect the actual message on the MQSeries queue.

Defined in: MQQueue class

Syntax:
Call *MQQueue*.**Put** *(Message, PutMsgOptions&)*

Parameters: *Message* MQMessage Object representing message to be put.

PutMsgOptions&

MQPutMessageOptions object containing options to control the put operation.

MQMessage class

This class represents an MQSeries message. It includes properties to encapsulate the MQSeries message descriptor (MQMD), and provides a buffer to hold the application-defined message data.

The class includes methods (Write methods) to copy data from a LotusScript application to an MQMessage object and similarly methods (Read methods) to copy data from an MQMessage object to a LotusScript application. The class manages the allocation and deallocation of memory for the buffer automatically. The application does not have to declare the size of the buffer when an MQMessage object is created as the buffer grows to accommodate data written to it.

Note: You will not be able to place a message onto an MQSeries queue if the buffer size exceeds the MaximumMessageLength property of that queue.

Once it has been constructed an MQMessage Object may be Put onto an MQSeries queue using the MQQueue.Put method. This method takes a copy of the MQMD and message data portions of the object and places that copy on the queue - so the application may modify or delete an MQMessage object after the Put, without affecting the message on the MQSeries queue. The queue manager may adjust some of fields in the MQMD when it copies the message on to the MQSeries queue. These adjustments are made to the copy of the MQMD held in the MQMessage object so that, unless subsequently modified, the MQMessage object is a true copy of what actually went onto the MQSeries queue.

An incoming message may be read into an MQMessage Object using the MQQueue.Get method. This replaces any MQMD or message data that may already have been in the MQMessage object with values from the incoming message, adjusting the size of the MQMessage object's data buffer to match the size of the incoming message data.

Containment:

Messages are contained by the MQSession class.

Properties:

The control properties are:

- CompletionCode property
- DataLength property
- DataOffset property
- GroupId property
- GroupIdHex property
- MessageFlags property
- MessageLength property
- MessageSequenceNumber property
- Offset property
- OriginalLength property
- ReasonCode property
- ReasonName property
- TotalMessageLength property

The Message Descriptor Properties are:

- AccountingToken property
- AccountingTokenHex property
- ApplicationIdData property

- ApplicationOriginData property
- BackoutCount property
- CharacterSet property
- CorrelationId property
- CorrelationIdHex property
- Encoding property
- Expiry property
- Feedback property
- Format property
- GroupId property
- GroupIdHex property
- MessageId property
- MessageIdHex property
- MessageType property
- Persistence property
- Priority property
- PutApplicationName property
- PutApplicationType property
- PutDateTime property
- ReplyToQueueManagerName property
- ReplyToQueueName property
- Report property
- UserId property

Methods:

- ClearErrorCodes method
- ClearMessage method
- GetMessageData method
- ReadBoolean method
- ReadByte method
- ReadDecimal2 method
- ReadDecimal4 method
- ReadDouble method
- ReadFloat method
- ReadInt2 method
- ReadInt4 method
- ReadLong method
- ReadNullTerminatedString method
- ReadShort method
- ReadString method
- ReadUTF method
- ReadUInt2 method
- ReadUnsignedByte method
- ResizeBuffer method
- WriteBoolean method
- WriteByte method
- WriteDecimal2 method
- WriteDecimal4 method
- WriteDouble method
- WriteFloat method
- WriteInt2 method
- WriteInt4 method
- WriteLong method
- WriteNullTerminatedString method

- WriteShort method
- WriteString method
- WriteUTF method
- WriteUInt2 method
- WriteUnsignedByte method

LotusScript Events:

- Mqerror
- Mqwarning

Creation:

New creates a new MQMessage object. Its Message Descriptor properties are initially set to default values, and its Message Data buffer is empty

Syntax:

Dim *msg* **As New MQMessage** or **Set** *msg* **= New MQMessage**

Property Access:

All properties can be read at any time.

The control properties are read-only, except for **DataOffset** which is read-write. The Message Descriptor properties are all read-write, except BackoutCount which is read-only.

Note however that some of the MQMD properties may be modified by the queue manager when the message is put onto an MQSeries queue. See the *MQSeries Application Programming Reference* for details.

You can pass binary data to an MQSeries message by setting the CharacterSet property to the Coded Character Set Identifier of the queue manager (MQCCSI_Q_MGR), and passing it a string. You can use the chr$ function to set non-character data into the string.

Data Conversion:

The ReadLong, ReadShort,WriteLong, and WriteShort methods perform data conversion. They convert between the LotusScript internal formats, and the MQSeries message formats as defined by the Encoding and CharacterSet properties from the message descriptor. When writing a message you should, if possible, set values into Encoding and CharacterSet that match the characteristics of the recipient of the message before issuing a WriteLong or WriteShort method. When reading a message this is not normally required as these values will have been set from those in the incoming MQMD.

CompletionCode property

Read-only. Returns the MQSeries completion code set by the most recent method or property access issued against this object.

Defined in: MQMessage class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = MQMessage***.CompletionCode**

DataLength property

Read-only. This property returns the value

```
Message.MessageLength - Message.DataOffset
```

It can be used before a Read method, to check that the expected number of characters are actually present in the buffer.

The initial value is zero.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *bytesleft& = MQMessage***.DataLength**

DataOffset property

Read-write. The current position within the Message Data portion of the message object.

The value is expressed as a character offset from the start of the message data buffer; the first character in the buffer corresponds to a **DataOffset** value of zero.

A read or write method commences its operation at the character referenced by **DataOffset**. These methods process data in the buffer sequentially from this position, and update **DataOffset** to point to the character (if any) immediately following the last character processed.

DataOffset may only take values in the range 0..MessageLength inclusive. When DataOffset = MessageLength it is pointing to the end (first invalid character) of the buffer. Write methods are permitted in this situation - they extend the data in the buffer and increase MessageLength by the number of characters added. Reading beyond the end of the buffer is not supported.

The initial value is zero.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *currpos& = MQMessage*.**DataOffset**

To set: *MQMessage*.**DataOffset** = *currpos&* .

GroupId property

Read-write. The GroupId to be included in the MQPMR of a message when put on a queue, also the Id to be matched against when getting a message from a queue. Its initial value is all NULLS.

Defined in: MQMessage class

Data Type: String of 24 characters

Syntax:
To get: *groupid$ = MQMessage*.**GroupId**

To set: *MQMessage*.**GroupId** = *groupid$*

GroupIdHex property

Read-write. The GroupId to be included in the MQPMR of a message when put on a queue, also the Id to be matched against when getting a message from a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B" and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQMessage class

Data Type: String of 48 hexadecimal characters representing 24 ASCII characters.

Syntax:
To get: *groupidh$ = MQMessage*.**GroupIdHex**

To set: *MQMessage*.**GroupIdHex** = *groupidh$*

MessageFlags property

Read-Write. Message flags specifying Segmentation control information. The initial value is 0.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *messageflags& = MQMessage*.**MessageFlags**

To set: *MQMessage*.**MessageFlags** = *messageflags&*

MessageLength property

Read-only. Returns the total length of the Message Data portion of the message object in characters, irrespective of the value of **DataOffset**.

The initial value is zero. It is set to the incoming Message Length after a Get method invocation that referenced this message object. It is incremented if the application uses a Write method to add data to the object. It is unaffected by Read methods.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *msglength& = MQMessage*.**MessageLength**

MessageSequenceNumber property

Read-Write. Sequence information identifying a message within a group. The initial value is 1.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *sequencenumber& = MQMessage*.**SequenceNumber**

To set: *MQMessage*.**SequenceNumber** = *sequencenumber&*

Offset property

Read-Write. The offset in segmented message. The initial value is 0.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *offset& = MQMessage*.**Offset**

To set: *MQMessage*.**Offsets** = *offset&*

OriginalLength property

Read-Write. The original length of a segmented message. The initial value is MQOL_UNDEFINED.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *originallength& = MQMessage*.**OriginalLength**

To set: *MQMessage*.**OriginalLength** = *originallength&*

ReasonCode property

Read-only. Returns the reason code set by the most recent method or property access issued against this object.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasoncode& = MQMessage*.**ReasonCode**

ReasonName property

Read-only. Returns the symbolic name of the latest reason code. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQMessage class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname& = MQMessage*.**ReasonName**

TotalMessageLength property

Read-Only. Returns the original length of a message (that is, before truncation).

Defined in: MQMessage class

Syntax:
To get: *length& = MQMessage.***TotalMessageLength**

AccountingToken property

Read-write. The MQMD AccountingToken, part of the message Identity Context.

Note: When setting this property you must specify all 32 characters. Its initial value is all nulls.

Defined in: MQMessage class

Data Type: String of 32 characters

Syntax:
To get: *actoken$ = MQMessage*.**AccountingToken**

To set: *MQMessage*.**AccountingToken** *= actoken$*

AccountingTokenHex property

Read-write. The MQMD AccountingToken, part of the message Identity Context.

Every two characters represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B", and so on.

You must supply 64 valid hexadecimal characters.

Its initial value is "0..0"

Defined in: MQMessage class

Data Type: String of 64 hexadecimal characters representing 32 ASCII characters

Syntax:
To get: *actokenh$ = MQMessage*.**AccountingTokenHex**

To set: *MQMessage.***AccountingTokenHex =** *actokenh$*

ApplicationIdData property

Read-write. The MQMD ApplIdentityData, part of the message Identity Context.

Its initial value is all blanks.

Defined in: MQMessage class

Data Type: String of 32 characters

Syntax:
To get: *applid$ = MQMessage*.**ApplicationIdData**

To set: *MQMessage*.**ApplicationIdData** *= applid$*

ApplicationOriginData property

Read-write. The MQMD ApplOriginData, part of the message origin context.

Its initial value is all blanks.

Defined in: MQMessage class

Data Type: String of 4 characters

Syntax:
To get: *applor$ = MQMessage*.**ApplicationOriginData**

To set: *MQMessage*.**ApplicationOriginData** *= applor$*

BackoutCount property

Read-only. The MQMD BackoutCount.

Its initial value is 0

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *backoutct& = MQMessage*.**BackoutCount**

CharacterSet property

Read-write. The MQMD CodedCharSetId. This specifies the character set used for the application data in the message.

Its initial value is the special value MQCCSI_Q_MGR.

If CharacterSet is set to MQCCSI_Q_MGR (the CharacterSet for the data is the same as that for the queue manager), the WriteString method will not perform code page conversion.

For example:

```
msg.CharacterSet = MQCCSI_Q_MGR
msg.WriteString(chr$(n))
```

where 'n' is greater than or equal to zero and less than or equal to 255, results in a single byte of value of 'n' being written to the buffer.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *ccid& = MQMessage*.**CharacterSet**

To set: *MQMessage*.**CharacterSet** *= ccid&*

Example: If you want the string written out in code page 437, issue

```
Message.CharacterSet = 437
Message.WriteString ("string to be written")
```

Set the value you want in the CharacterSet before issuing any WriteString calls.

CorrelationId property

Read-write. The CorrelId to be included in the MQMD of a message when put on a queue, also the Id to be matched against when getting a message from a queue.

When setting this property you must specify all 24 characters.

Its initial value is all nulls.

Defined in: MQMessage class

Data Type: String of 24 characters

Syntax:
To get: *correlid$ = MQMessage*.**CorrelationId**

To set: *MQMessage*.**CorrelationId** *= correlid$*

CorrelationIdHex property

Read-write. The CorrelId to be included in the MQMD of a message when put on a queue, also the CorrelId to be matched against when getting a message from a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B", and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQMessage class

Data Type: String of 48 hexadecimal characters representing 24 ASCII characters

Syntax:
To get: *correlidh$ = MQMessage*.**CorrelationIdHex**

To set: *MQMessage*.**CorrelationIdHex** *= correlidh$*

Encoding property

Read-write. The MQMD field that identifies the representation used for numeric values in the application message data.

Its initial value is the special value MQENC_NATIVE, which varies by platform.

Setting this property to MQENC_INTEGER_UNDEFINED has the same results as setting it to the default, MQENC_NATIVE.

This property is used by the following methods:

- ReadLong
- WriteLong
- ReadShort
- WriteShort

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *encoding& = MQMessage*.**Encoding**

To set: *MQMessage*.**Encoding** = *encoding&*

If you are preparing to write data to the message buffer, you should set this field to match the characteristics of the receiving queue manager platform if you know what it is.

Expiry property

Read-write. The MQMD expiry time field, expected in tenths of a second.

Its initial value is the special value MQEI_UNLIMITED

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *expiry& = MQMessage*.**Expiry**

To set: *MQMessage*.**Expiry** = *expiry&*

Feedback property

Read-write. The MQMD feedback field.

Its initial value is the special value MQFB_NONE.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *feedback& = MQMessage*.**Feedback**

To set: *MQMessage*.**Feedback** = *feedback&*

Format property

Read-write. The MQMD format field. Gives the name of a built-in or user-defined format that describes the nature of the Message Data.

Its initial value is the special value MQFMT_NONE.

Defined in: MQMessage class

Data Type: String of 8 characters

Syntax:
To get: *format$ = MQMessage*.**Format**

To set: *MQMessage*.**Format** = *format$*

GroupId property

Read-write. The GroupId to be included in the MQPMR of a message when put on a queue, also the Id to be matched against when getting a message from a queue.

Its initial value is all nulls.

Defined in: MQMessage class

Data Type: String of 24 characters

Syntax:
To get: *groupid$ = MQMessage.*.**GroupId**

To set: *MQMessage.*.**GroupId** = *groupid$*

GroupIdHex property

Read-Write. The GroupId to be included in the MQPMR of a message when put on a queue, also the Id to be matched against when getting a message from a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", and the pairs "6" and "2" represent the single character "B" and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQMessage class

Data Type: String of 48 hexadecimal characters representing 24 ASCII characters

Syntax:
To get: *groupid$ = MQMessage.***GroupIdHex**

To set: *MQMessage.***GroupIdHex** *= groupid$*

MessageId property

Read-write. The MessageId to be included in the MQMD of a message when put on a queue, also the Id to be matched against when getting a message from a queue.

Note: When setting this property you must specify all 24 characters.

Its initial value is all nulls.

Defined in: MQMessage class

Data Type: String of 24 characters

Syntax:
To get: *messageid$ = MQMessage***.MessageId**

To set: *MQMessage***.MessageId** *= messageid$*

MessageIdHex property

Read-write. The MessageId to be included in the MQMD of a message when put on a queue, also the MessageId to be matched against when getting a message from a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "a", the pair of characters "6" and "2" represent the single character "b", and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQMessage class

Data Type: String of 48 hexadecimal characters representing 24 ASCII characters

Syntax:
To get: *messageidh$ = MQMessage*.**MessageIdHex**

To set: *MQMessage*.**MessageIdHex** *= messageidh$*

MessageType property

Read-write. The MQMD MsgType field.

Its initial value is MQMT_DATAGRAM.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *msgtype& = MQMessage*.**MessageType**

To set: *MQMessage*.**MessageType** *= msgtype&*

Persistence property

Read-write. The message's persistence setting.

Its initial value is MQPER_PERSISTENCE_AS_Q_DEF.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *persist& = MQMessage*.**Persistence**

To set: *MQMessage*.**Persistence** *= persist&*

Priority property

Read-write. The message's priority.

Its initial value is the special value MQPRI_PRIORITY_AS_Q_DEF.

Defined in: MQMessage class

Data Type: Long

Syntax:
To get: *priority& = MQMessage*.**Priority**

To set: *MQMessage*.**Priority** = *priority&*

PutApplicationName property

Read-write. The MQMD PutApplName - part of the Message Origin context.

Its initial value is all blanks.

Defined in: MQMessage class

Data Type: String of 28 characters

Syntax:
To get: *putapplnm$ = MQMessage*.**PutApplicationName**

To set: *MQMessage*.**PutApplicationName** = *putapplnm$*

PutApplicationType property

Read-write. The MQMD PutApplType - part of the Message Origin context.

Its initial value is MQAT_NO_CONTEXT.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *putappltp& = MQMessage*.**PutApplicationType**

To set: *MQMessage*.**PutApplicationType** = *putappltp&*

PutDateTime property

Read-write. This property combines the MQMD PutDate and PutTime fields. These are part of the Message Origin context that indicate when the message was put.

The LotusScript Extension converts between LotusScript date/time format and the Date and Time formats used in an MQSeries MQMD. If a message is received that has an invalid PutDate or PutTime, the PutDateTime property after the get method will be set to EMPTY.

Its initial value is EMPTY

Defined in: MQMessage class

Data Type: Variant of type 7 (date/time) or EMPTY.

Syntax:
To get: *datetime = MQMessage*.**PutDateTime**

To set: *MQMessage*.**PutDateTime**. = *datetime*

ReplyToQueueManagerName property

Read-write. The MQMD ReplyToQMgr field.

Its initial value is all blanks

Defined in: MQMessage class

Data Type: String of 48 characters

Syntax:
To get: *replytoqmgr$ = MQMessage*.**ReplyToQueueMana gerName**

To set: *MQMessage*.**ReplyToQueueManagerName =** *replytoqmgr$*

ReplyToQueueName property

Read-write. The MQMD ReplyToQ field.

Its initial value is all blanks

Defined in: MQMessage class

Data Type: String of 48 characters

Syntax:
To get: *replytoq$ = MQMessage*.**ReplyToQueueName**

To set: *MQMessage*.**ReplyToQueueName =** *replytoq$*

Report property

Read-write. The message's Report options.

Its initial value is MQRO_NONE.

Defined in: MQMessage class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *report& = MQMessage***.Report**

To set: *MQMessage***.Report** = *report&*

UserId property

Read-write. The MQMD UserIdentifier - part of the message Identity Context.

Its initial value is all blanks.

Defined in: MQMessage class

Data Type: String of 12 characters

Syntax:
To get: *userid$ = MQMessage***.UserId**

To set: *MQMessage***.UserId** = *userid$*

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQMessage class and the MQSession class.

Defined in: MQMessage class

Syntax:
Call *MQMessage***.ClearErrorCodes**

ClearMessage method

Clears the Message Data buffer portion of the MQMessage object. All data in the data buffer is lost. **MessageLength**, **DataLength**, and **DataOffset** are all reset to zero.

The Message Descriptor (MQMD) portion is unaffected; an application may need to modify some of the MQMD fields before reusing the MQMessage object. If you want to set the MQMD fields back to initial values you should use New to replace the object with a new instance.

Defined in: MQMessage class

Syntax:
*Message***.ClearMessage**

ReadBoolean method

Reads a 1-byte Boolean value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as an integer (2-byte signed) value: True(-1) or False(0).

Note: True is displayed as -1 in LotusScript but is stored as 1 in the Message Data buffer.

This method fails if MQMessage.**DataLength** is 0 when it is issued.

The **DataOffset** property is incremented by 1 and **DataLength** is decremented by 1 if the method call succeeds. The 1 byte of data is assumed to be a signed binary integer.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

value% = MQMessage.**ReadBoolean**

Note: True is actually stored in the message buffer as 1 and false as 0.

ReadByte method

Reads a 1-byte signed character value from the Message Data buffer, from the pointed to by the **DataOffset** property, and returns it as an integer (2-byte signed) value in the range -128 to 127.

The method fails if Message.**DataLength** is less than 1 when it is issued.

DataOffset is incremented by 1 and **DataLength** is decremented by 1 if the method succeeds.

The 1 byte of data is assumed to be a signed binary integer.

Defined in: MQMessage class.

Syntax:
Dim value% as Integer

value% = MQMessage.**ReadByte**

ReadDecimal2 method

Reads a 2-byte packed decimal value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as an integer (2-byte signed) value.

This method fails if MQMessage.**DataLength** is less than 2 when it is issued.

The **DataOffset** property is incremented by 2 and **DataLength** is decremented by 2 if the method call succeeds.

The 2 bytes of data are assumed to be a signed binary integer whose encoding is specified by the MQMessage.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

value% = MQMessage.**ReadDecimal2**

ReadDecimal4 method

Reads a 4-byte packed decimal value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as a long (4-byte) signed value.

This method fails if MQMessage.**DataLength** is less than 4 when it is issued.

The **DataOffset** property is incremented by 4 and **DataLength** is decremented by 4 if the method call succeeds.

The 4 bytes of data are assumed to be a signed binary long value whose encoding is specified by the MQMessage.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value& as Long

Call *value& = MQMessage*.**ReadDecimal4**

ReadDouble method

Reads an 8-byte signed floating point value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as a double (signed 8-byte floating point) value.

The method fails if Message.**DataLength** is less than 8 when it is issued.

DataOffset is incremented by 8 and **DataLength** is decremented by 8 if the method succeeds.

The 8 bytes of data are assumed to be a signed binary double value whose encoding is specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value# as Double

*doublev# = MQMessage.***ReadDouble**

ReadFloat method

Reads a 4-byte floating point value from the Message Data buffer, from the position pointed to by the **DataOffset** and returns it as a single (signed 4-byte floating point) value.

The method fails if Message.**DataLength** is less than 4 when it is issued.

DataOffset is incremented by 4 and **DataLength** is decremented by 4 if the method succeeds.

The 4 bytes of data are assumed to be a floating point value whose encoding is specified by the Message.Encoding property.

Note: Conversion from System 360 format is not supported.

Defined in: MQMessage class

Syntax:
Dim value! as Single

*value! = MQMessage.***ReadFloat**

ReadInt2 method

Reads a 2-byte signed integer value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as an integer (signed 2-byte) value.

Note: This method of functionally identical to the ReadShort method.

The method fails if Message.**DataLength** is less than 2 when it is issued.

DataOffset is incremented by 2 and **DataLength** is decremented by 2 if the method succeeds.

The 2 bytes of data are assumed to be an integer value whose encoding is specified by the Message.Encoding property.

Note: Conversion from System/360 format is not supported.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

*value% = MQMessage.***ReadInt2**

ReadInt4 method

Reads a 4-byte signed long value from the Message Data buffer, starting from the position pointed to by the **DataOffset** property and returns it as a long (signed 4-byte) value.

Note: this method is functionally identical to the ReadLong method.

The method fails if Message.**DataLength** is less than 4 when it is issued.

DataOffset is incremented by 4 and **DataLength** is decremented by 4 if the method succeeds.

The 4 bytes of data are assumed to be a binary integer value whose encoding is specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value& as Long

*value& = MQMessage.***ReadInt4**

ReadLong method

Reads a 4-byte signed long value from the Message Data buffer, from the position pointed to by the **DataOffset** and returns it as a Long (signed 4-byte) value.

Note: This method is functionally identical to the ReadInt4 method.

The method fails if *Message*.**DataLength** is less than 4 when it is issued.

DataOffset is incremented by 4 and **DataLength** is decremented by 4 if the method succeeds.

The 4 bytes of data are assumed to be a binary integer value whose encoding is specified by the *Message***.Encoding** property. Conversion to LotusScript representation is performed for the application.

Defined in: MQMessage class

Syntax:
Dim value& as Long

*value& = MQMessage***.ReadLong**

ReadNullTerminatedString method

This is for use in place of ReadString if the string may contain embedded null characters.

Reads the specified number of characters from the Message Data buffer starting with the character referred to by **DataOffset** and returns it as a LotusScript string. If the string contains an embedded null before the end then the length of the returned string is reduced to reflect only those characters before the null.

DataOffset is incremented and Datalength is decremented by the value specified regardless of whether or not the string contains embedded null characters.

The characters in the message data are assumed to be a string whose code page is specified by the Message.Characterset property.

Defined in: MQMessage class

Syntax:
string$ = *MQMessage*.**ReadString**(length$)

Parameters: *length$* String. Length of string field in bytes.

ReadShort method

Reads a 2-byte signed integer value from the Message Data buffer, from the position pointed to by the **DataOffset** and returns it as an integer (signed 2-byte) value.

Note: This method id functionally identical to the ReadInt2 method.

The method fails if *Message*.**DataLength** is less than 2 when it is issued.

DataOffset is incremented by 2 and **DataLength** is decremented by 2 if the method succeeds.

The 2 bytes of data are assumed to be an integer value whose encoding is specified by the *Message*.**Encoding** property.

Defined in: MQMessage class

Syntax:
Dim value% as Integer *value% = Message***.ReadShort**

ReadString method

This method reads n characters from the Message Data buffer (where n can be any number in the range 1 to 32000 inclusive) starting with the character referred to by **DataOffset** and returns it as a LotusScript string.

The method fails if *Message*.**DataLength** is less than n when it is issued.

DataOffset is incremented by n and **DataLength** is decremented by n if the method succeeds.

The n characters of message data are assumed to be a string whose code page is specified by the *Message***.CharacterSet** property. Conversion to LotusScript format is performed automatically.

Defined in: MQMessage class

Syntax:
*firstname$ = Message***.ReadString** *(length&)*

ReadUTF method

Reads n character bytes (where n is 1 to 32000) from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as a LotusScript string.

A UTF-8 (where UTF stands for Unicode Transformation Format) string is a formatted string that contains the data length of the data stored in it as the first 2-bytes of data followed by the actual data itself.

This method fails if MQMessage.**DataLength** is less than n when issued.

DataOffset property is incremented by n and **DataLength** is decremented by n if the method call succeeds.

The n characters of message data are assumed to be a string whose code page is specified by the MQMessafe.CharacterSet property. Conversion to LotusScript format (Unicode) is performed automatically.

Defined in: MQMessage class

Syntax:
Dim value$ as String

Dim length& as Long *value$ = MQMessage*.**ReadUTF** Length&

ReadUInt2 method

Reads a 2-byte unsigned integer value from the Message Data buffer, from the position pointed to by the **DataOffset** property and returns it as a Long (signed 4-byte) value.

The method fails if Message.**DataLength** is less than 2 when it is issued.

DataOffset is incremented by 2 and **DataLength** is decremented by 2 if the method succeeds.

The 2 bytes of data are assumed to be an integer value whose encoding is specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value& as Long

value& = MQMessage.**ReadUInt2**

ReadUnsignedByte method

Reads a 1-byte unsigned character value from the Message Data buffer, from the position pointed to by the **DataOffset** property, and returns it as an integer (2-byte signed) value in the range 0 to 255.

The method fails if the *Message*.**DataLength** is less than 1 when it is issued.

DataOffset is incremented by 1 and **DataLength** is decremented by 1 if the method succeeds.

The 1 byte of data is assumed to be a signed binary integer.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

*value% = MQMessage.***ReadUnsignedByte**

ResizeBuffer method

This method alters the amount of storage currently allocated internally to hold the Message Data Buffer. It gives the application some control over the automatic buffer management, in that if the application knows that it is going to deal with a large message, it can ensure that a sufficiently large buffer is allocated. The application does not need to use this call - if it does not, the automatic buffer management code will grow the buffer size to fit.

Caution: If you resize the buffer to be smaller that the current **MessageLength**, you risk losing data. If you do lose data, the method returns a **CompletionCode** of MQCC_WARNING and a ReasonCode of MQRC_DATA_TRUNCATED.

If you resize the buffer to be smaller than the value of the **DataOffset** property the:

- **DataOffset** property is changed to point to the end of the new buffer
- **DataLength** property is set to zero
- **MessageLength** property is changed to the new buffer size

The default buffer size if 2K bytes.

Defined in: MQMessage class

Syntax:
Call *MQMessage*.**ResizeBuffer***(length&)*

WriteBoolean method

Takes a 2-byte LotusScript integer value (0=False, -1=True) and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 1-byte binary number (0=False, 1=True). It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

Note: True(-1) is converted to a '1' in the actual Message Data buffer.

DataOffset is incremented by 1 if the method call succeeds.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

where value% =

Call *MQMessage*.**WriteBoolean**(value%)

True or False

WriteByte method

Takes a 2-byte LotusScript integer value (from -128 to +127) and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 1-byte character. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (Message.MessageLength) if necessary.

DataOffset is incremented by 1 if the method succeeds.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *MQMessage*.**WriteByte** *(value%)*

WriteDecimal2 method

Takes a signed 2-byte integer value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 2-byte packed decimal number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

DataOffset is incremented by 2 if the method call succeeds.

This method converts the data to the representation specified by the MQMessage.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *MQMessage*.**WriteDecimal2**(value%)

WriteDecimal4 method

Takes a signed 4-byte long value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 4-byte packed decimal number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

DataOffset is incremented by 4 if the method call succeeds.

This method converts the data to the representation specified by the MQMessage.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value% as Long

Call *MQMessage*.**WritedDecimal4**(value&) *value& Long.* Value to be written

WriteDouble method

Takes a signed 8-byte double floating point value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as an 8-byte floating point number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

DataOffset is incremented by 8 if the method call succeeds.

The method converts to the floating point representation specified by the Message.Encoding property.

Note: Conversion to System/360 format is not supported.

Defined in: MQMessage class

Syntax:
Dim value# as Double

Call *MQMessage.***WriteDouble***(value#)*

WriteFloat method

Takes a signed 4-byte double floating point value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 4-byte floating number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (Message.MessageLength) if necessary.

DataOffset is incremented by 4 if the method succeeds.

This method converts to the floating point representation specified by the Message.Encoding property.

Note: Conversion to System/360 format is not supported

Defined in: MQMessage class

Syntax:
Dim value! as Single

Call *MQMessage.***WriteFloat***(value!)*

WriteInt2 method

Takes a signed 2-byte integer value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 2-byte binary number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

Note: This method is functionally identical to the WriteShort method.

DataOffset is incremented by 2 if the method succeeds.

This method converts to the representation specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *MQMessage.***WriteInt2***(value%)*

WriteInt4 method

Takes a signed 4-byte long value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 4-byte binary number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (Message.Messagelength) if necessary.

Note: This method is functionally identical to the WriteLong method.

DataOffset is incremented by 4 if the method succeeds.

The method converts the data to the representation specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value& as Long

Call *MQMessage.***WriteInt4***(value&)*

WriteLong method

Takes a signed 4-byte long value and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 4-byte binary number. It replaces any data already at these positions in the buffer, and extends the length of the buffer (*Message*.**MessageLength**) if necessary.

DataOffset is incremented by 4 if the method succeeds.

The method converts the data to the representation specified by the *Message*.**Encoding** property.

Note: A message with the Encoding property set to MQENC_INTEGER_UNDEFINED is processed in the same way as a message with the Encoding property set to MQENC_NATIVE.

Defined in: MQMessage class

Syntax:
Dim value& as Long

Call *MQMessage*.**WriteLong***(value&*)

WriteNullTerminatedString method

This method performs a normal WriteString and pads any remaining bytes up to the specified length with null. If the number of bytes written by the initial write string is equal to the specified length then no nulls are written. If the number of bytes exceeds the specified length then an error (reason code MQRC_WRITE_VALUE_ERROR) is set.

Defined in: MQMessage class

Syntax:
call *MQMessage.* **WriteNullTerminatedString***(value¢, length&)*

Parameters: value¢ String. Value to be written.

length& Long. Length of string field in bytes.

WriteShort method

Takes a signed 2-byte integer value(from -32768 to +32767) and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 2-byte binary number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (*Message*.**MessageLength**) if necessary.

Note: This method is functionally identical to the WriteInt2 method.

DataOffset is incremented by 2 if the method succeeds.

The method converts the data to the representation specified by the *Message***.Encoding** property.

Note: A message with the Encoding property set to MQENC_INTEGER_UNDEFINED is processed in the same way as a message with the Encoding property set to MQENC_NATIVE.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *Message***.WriteShort***(value%)*

WriteString method

This method takes a LotusScript string and writes it into the Message Data buffer starting at the character referred to by **DataOffset**. It replaces any data already at these positions in the buffer, and will extend the length of the buffer (*Message***.MessageLength**) if necessary.

DataOffset is incremented by the length of the string in characters if the method succeeds.

The method converts characters into the code page specified by the *Message***.CharacterSet** property.

Defined in: MQMessage class

Syntax:
Dim value$ as String

Call *Message***.WriteString***(value$)*

WriteUTF method

Takes a LotusScript string, calculates the length (n) of the string and copies the length (2 bytes) into the Message Data buffer at the position pointed to by the **DataOffset** property followed by the actual data itself and extends the length of the buffer (MQMessage.MessageLength) if necessary.

DataOffset is incremented by 2+n if the method call succeeds.

This method converts characters into the code page specified by the MQMessage.CharacterSet property.

Defined in: MQMessage class

Syntax:
Dim value$ as String

Call *MQMessage*.**WriteUTF**(value$)

WriteUInt2 method

Takes a signed 4-byte integer value (from 0 to +65535) and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 2-character binary number. It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

Note: This method is functionally identical to the WriteShort method.

DataOffset is incremented by 2 if the method succeeds.

This method converts to the binary representation specified by the Message.Encoding property.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *MQMessage.***WriteUInt2***(value%)*

WriteUnsignedByte method

Takes a 2-byte LotusScript integer value (from 0 to +255) and copies it into the Message Data buffer at the position pointed to by the **DataOffset** property as a 1-byte character.

It replaces any data that may have existed at the **DataOffset** position prior to the method call and extends the length of the buffer (MQMessage.MessageLength) if necessary.

DataOffset is incremented by 1 if the method call succeeds.

Defined in: MQMessage class

Syntax:
Dim value% as Integer

Call *Message*.**WriteUnsignedByte***(value%)*

MQPutMessageOptions class

This class encapsulates the various options that control the action of putting a message onto an MQSeries queue.

Containment: Contained by the MQSession class.

Properties:

- CompletionCode property
- Options property
- ReasonCode property
- ReasonName property
- RecordFields property
- ResolvedQueueManagerName property
- ResolvedQueueName property

Methods:

- ClearErrorCodes method

LotusScript Events:

- Mqerror
- Mqwarning

Creation: New creates a new MQPutMessageOptions object and sets all its properties to initial values.

Syntax:
Dim *pmo* **As New MQPutMessageOptions** or

Set *pmo* **= New MQPutMessageOptions**

CompletionCode property

Read-only. Returns the completion code set by the last method or property access issued against the object.

Defined in: MQPutMessageOptions class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = PutOpts*.**CompletionCode**

Options property

Read-write. The MQPMO Options field that controls the putting a message on a queue.

The options are listed in the *MQSeries Application Programming Reference* manual. For example:

- MQPMO_SYNCPOINT - the message is not visible until the unit of work is completed.
- MQPMO_NO_SYNCPOINT - the message is available immediately and cannot be deleted by backing out a unit of work.
- MQPMO_DEFAULT_CONTEXT - the queue manager sets the context fields in the message.

The initial value is MQPMO_NONE.

Note: The options not supported are:

- MQPMO_PASS_IDENTITY_CONTEXT
- MQPMO_PASS_ALL_CONTEXT

Defined in: MQPutMessageOptions class

Data Type: Long

Syntax:
To get: *options& = PutOpts*.**Options**

To set: *PutOpts*.**Options** *= options&*

ReasonCode property

Read-only. Returns the reason code set by the last method or property access issued against the object.

Defined in: MQPutMessageOptions class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasoncode& = PutOpts*.**ReasonCode**

ReasonName property

Read-only. Returns the symbolic name of the latest reason code. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQPutMessageOptions class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname& = MQPutMessageOptions.*.**ReasonName**

RecordFields property

Read-write. Flags indicating which fields are to be customized on a per-queue basis when putting a message to a distribution list. The initial value is zero.

Defined in: MQPutMessageOptions class

Data Type: Long

Syntax:
To get: *recordfields& = MQGetMessageOptions* **.RecordFields**

To set: *MQGetMessageOptions*.**RecordFields** = *recordfields&*

ResolvedQueueManagerName property

Read-only. The MQPMO ResolvedQMgrName field. See the *MQSeries Application Programming Reference* for details. The initial value is all blanks.

Defined in: MQPutMessageOptions class

Data Type: String of 48 characters

Syntax:
To get: *qmgr$ = PutOpts***.ResolvedQueueManagerName**

ResolvedQueueName property

Read-only. The MQPMO ResolvedQName field. See the *MQSeries Application Programming Reference* for details. The initial value is all blanks.

Defined in: MQPutMessageOptions class

Data Type: String of 48 characters

Syntax:
To get: *qname$ = PutOpts* **.ResolvedQueueName**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQPutMessageOptions class and the MQSession class.

Defined in: MQPutMessageOptions class

Syntax:
Call *PutOpts***.ClearErrorCodes**

MQGetMessageOptions class

This class encapsulates the various options that control the action of getting a message from an MQSeries Queue.

Containment:
Contained by the MQSession class.

Properties:

- CompletionCode property
- MatchOptions property
- Options property
- ReasonCode property
- ReasonName property
- ResolvedQueueName property
- WaitInterval property

Methods:
ClearErrorCodes method

LotusScript Events:

- Mqerror
- Mqwarning

Creation:
New creates a new MQGetMessageOptions object and sets all its properties to initial values.

Syntax:
Dim *gmo* **As New MQGetMessageOptions** or

Set *gmo* **= New MQGetMessageOptions**

CompletionCode property

Read-only. Returns the completion code set by the last method or property access issued against the object.

Defined in: MQGetMessageOptions class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = GetOpts***.CompletionCode**

MatchOptions property

Read-write. Options controlling selection criteria used for MQGET. The initial value is MQMO_MATCH_MSG_ID + MQMO_MATCH_CORREL_ID.

Defined in: MQGetMessageOptions class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *matchoptions& = MQGetMessageOptions*. **MatchOptions**

To set: *MQGetMessageOptions*.**MatchOptions** = *matchoptions&*

Options property

Read-write. The MQGMO Options field that controls the action of getting a message from a queue.

The options are listed in the *MQSeries Application Programming Reference*

For example:

- MQGMO_SYNCPOINT - the message is marked as being unavailable to other applications, but only deleted from the queue when the unit of work is committed. The message is made available again if the unit of work is backed out.
- MQGMO_NO_SYNCPOINT - the message is retrieved and deleted from the queue immediately.
- MQGMO_ACCEPT_TRUNCATED - allows you to retrieve a message when the buffer is too small, although the complete message cannot be put in the buffer. Unless you have resized the buffer, you will get 2 KB of the message.
- MQGMO_CONVERT - requests that the application data in the message is converted to conform to the values of the CharacterSet and Encoding properties.

The initial value is MQGMO_NO_WAIT.

Defined in: MQGetMessageOptions class

Data Type: Long

Syntax:
To get: *options& = GetOpts*.**Options**

To set: *GetOpts*.**Options** = *options&*

ReasonCode property

Read-only. Returns the reason code set by the last method or property access issued against the object.

Defined in: MQGetMessageOptions class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasoncode& = GetOpts*.**ReasonCode**

ReasonName property

Read-only. The MQGMO ResolvedQName field. See the *MQSeries Application Programming Reference* for details. The initial value is all blanks.

Defined in: MQGetMessageOptions class.

Data Type: String of 48 characters.

Syntax:
To get: *qname$ = GetOpts.***ResolvedQueueName**

ResolvedQueueName property

Read-only. The MQGMO ResolvedQName field. See the *MQSeries Application Programming Reference* for details. The initial value is all blanks.

Defined in: MQGetMessageOptions class

Data Type: String of 48 characters

Syntax:
To get: *qname$ = GetOpts*.**ResolvedQueueName**

WaitInterval property

Read-write. The MQGMO WaitInterval field. The maximum time, in milliseconds, that the Get will wait for a suitable message to arrive if wait action has been requested by the Options property. See the *MQSeries Application Programming Reference* manual for details. Initial value is 0.

Defined in: MQGetMessageOptions class

Data Type: Long

Syntax:
To get: *wait& = GetOpts*.**WaitInterval**

To set: *GetOpts*.**WaitInterval** *= wait&*

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQGetMessageOptions class and the MQSession class.

Defined in: MQGetMessageOptions class

Syntax:
Call *GetOpts*.**ClearErrorCodes**

MQProcess class

This represents an MQSeries process definition object (used with triggering). Using the MQLSX, you can interrogate the properties of the process definition object within your script.

For more information about triggering, see the *MQSeries Application Programming Guide* book.

Containment: Contained by the MQQueueManager class.

Properties:

- AlternateUserId property
- ApplicationId property
- ApplicationType property
- CompletionCode property
- Description property
- EnvironmentData property
- Name property
- OpenStatus property
- ReasonCode property
- ReasonName property
- UserData property

Methods:

- ClearErrorCodes method

LotusScript Events:

- Mqerror
- Mqwarning

Property Access: All properties are read-only

AlternateUserId property

Read-only. The alternate user ID used to validate access to the Process definition when it was opened.

Defined in: MQProcess class

Data Type: String of 12 characters

Syntax:
To get: *altuser$ = Process*.**AlternateUserId**

ApplicationId property

Read-only. The MQI AppId attribute.

Defined in: MQProcess class

Data Type: String of 256 characters

Syntax:
To get: *identifier$ = Process*.**ApplicationId**

ApplicationType property

Read-only. The MQI ApplType attribute.

Defined in: MQProcess class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *appltype& = Process*.**ApplicationType**

CompletionCode property

Read-only. Returns the completion code set by the last method or property access issued against the object.

Defined in: MQProcess class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode& = Process*.**CompletionCode**

Description property

Read-only. The MQI ProcessDesc attribute.

Defined in: MQProcess class

Data Type: String of 64 characters

Syntax:
To get: *description$ = Process*.**Description**

EnvironmentData property

Read-only. The MQI EnvData attribute.

Defined in: MQProcess class

Data Type: String of 128 characters

Syntax:
To get: *env$ = Process*.**EnvironmentData**

Name property

Read-only. The MQI ProcessName attribute. This is the name of a process definition object, defined on the local queue manager.

Defined in: MQProcess class

Data Type: String of 48 characters

Syntax:
To get: *name$ = process*.**Name**

OpenStatus property

Read-only. Indicates if the process is Open or not.

Defined in: MQProcess class

Data Type: Long

Values:

- TRUE (-1)
- FALSE

Syntax:
To get: *status& = Process*.**OpenStatus**

ReasonCode property

Read-only. Returns the reason code set by the last method or property access issued against the object.

Defined in: MQProcess class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasoncode& = Process*.**ReasonCode**

ReasonName property

Read-only. The symbolic name for the ReasonCode. For example, “MQRC_QMGR_NOT_AVAILABLE”.

Defined in: MQProcess class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname$ = MQProcess.*.**ReasonName**

UserData property

Read-only. The MQI UserData attribute.

Defined in: MQProcess class

Data Type: String of 128 characters

Syntax:
To get: *user$ = Process*.**UserData**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQProcess class and the MQSession class.

Defined in: MQProcess class

Syntax:
Call *Process*.**ClearErrorCodes**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the ReasonCode to MQRC_NONE for both the MQQueueManager class and the MQSession class.

Defined in: MQQueueManager class.

Syntax:
Call *MQQueueManager*.**ClearErrorCodes()**

MQDistributionList class

This class encapsulates a collection of queues - local, remote, or alias, for output.

Properties:

- AlternateUserId property
- CloseOptions property
- CompletionCode property
- IsOpen property
- OpenOptions property
- ReasonCode property
- ReasonName property

Methods:

- AddDistributionListItem method
- ClearErrorCodes method
- Close method
- GetConnectionReference method
- GetFirstDistributionListItem method
- Open method
- Put method
- SetConnectionReference method

Creation:

New creates a new MQDistributionListItem object.

Syntax:

distributionlist = New **MQDistributionList**

AlternateUserId property

Read-write. The alternate user ID used to validate access to the list of queues when they are opened

Defined in: MQDistributionList class

Data Type: String of 12 characters

Syntax:

To get: *altuser$ = MQDistributionList*.**AlternateUs erId**

To set: *MQDistributionList.***AlternateUserId** = *altuser$*

CloseOptions property

Read-write. Options used to control what happens when the distribution list is closed. The initial value is MQCO_NONE.

Defined in: MQDistributionList class

Data Type: Long

Values:

- MQCO_NONE
- MQCO_DELETE
- MQCO_DELETE_PURGE

Syntax:
To get: *closeopt& = MQDistributionList.***CloseOptions**

To set: *MQDistributionList.***CloseOptions** = *closeopt&*

CompletionCode property

Read-only. The completion code set by the last method or property access issued against the object.

Defined in: MQDistributionList class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode$ = MQDistributionList.***CompletionCode**

IsOpen property

Read-only. A value that indicates whether or not the distribution list is currently open.

Defined in: MQDistributionList class

Data Type: Boolean (integer)

Values:

- TRUE (-1)
- FALSE (0)

Syntax:
To get: *isopen% = MQDistributionList.***IsOpen**

OpenOptions property

Read-write. Options to be used when the distribution list is opened. The default is MQOO_INQUIRE

Defined in: MQDistributionList class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *openopt& = MQDistributionList.***OpenOptions**

ReasonCode property

Read-only. The completion code set by the last method or property access issued against the object.

Defined in: MQDistributionList class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *reasoncode& = MQDistributionList.***ReasonCode**

ReasonName property

Read-only. The symbolic name for the ReasonCode. For example "MQRC_QMGR_NOT_AVAILABLE".

Defined in: MQDistributionList class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname$ = MQDistributionList.* **ReasonName**

AddDistributionListItem method

Creates a new MQDistributionListItem object and associates it with the Distribution List object. The name parameter is mandatory. The DistributionList and QueueManagerName properties are set from the owning Distribution List and the FirstDistributionListItem property for the Distribution List is set to reference this new Distribution List Item. The PreviousDistributionListItem property is set to nothing and the NextDistributionListItem property is set to reference any Distribution List Item that was previously first, or nothing if there was none previously (that is, the new one is inserted in front of those that exist already).

This will return an error if the Distribution List is open.

Defined in: MQDistributionList class

Syntax:
MQDistributionList. = **AddDistributionListItem** (QName$, QMgrName$)

Parameters: *QName$*
String. Name of the MQSeries queue.

QMgrName$
String. Name of the MQSeries queue manager.

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the Reason Code to MQRC_NONE for both the MQDistributionListItem class and the MQSession class.

Defined in: MQDistributionList class

Syntax:
MQDistributionList.**ClearErrorCodes**

Close method

Closes a distribution list using the current value of CloseOptions.

Defined in: MQDistributionList class

Syntax:
MQDistributionList.**Close**

GetConnectionReference method

The queue manager object associated with this Distribution List.

Defined in: MQDistributionList class

Data Type: MQQueueManager

Syntax:
set *queuemanager = MQDistributionList*.**GetConnectionReference()**

GetFirstDistributionListItem method

The first Distribution List Item object associated with this Distribution List.

Defined in: MQDistributionList class

Syntax:
set *distributionlistitem = MQDistributionList*.**GetFirstDistributionListItem**

Open method

Opens each of the queues specified by the Name and (where appropriate) QueueManagerName properties of the Distribution List Items associated with the current object using the current value of:

AlternateUserId

Defined in: MQDistributionList class

Syntax:
MQDistributionList.**Open**

Put method

Places a message on each of the queues identified by the distribution list items associated with the current object.

This method takes an MQMessage object as a parameter. The following distribution list item properties may be altered as a result of this method:

- CompletionCode
- ReasonCode
- ReasonName
- MessageId
- MessageIdHex
- CorrelationId
- CorrelationIdHex
- GroupId
- GroupIdHex
- Feedback
- AccountingToken
- AccountingTokenHex

Defined in: MQDistributionList class

Syntax:
Call *MQDistributionList*.**Put**(Message, PutMsgOptions)

Parameters:

Message
MQMessage object representing the message to be put.

PutMsgOptions
MQPutMessageOptions object containing options to control the put operation.

SetConnectionReference method

Sets the queue manager object associated with this Distribution List.

Defined in: MQDistributionList class

Data Type: MQQueueManager

Syntax:
MQDistributionList.**SetConnectionReference**

MQDistributionListItem class

This class encapsulates the MQOR, MQRR, and MQPMR structures and associates them with an owning distribution list.

Properties:

- AccountingToken property
- AccountingTokenHex property
- CompletionCode property
- CorrelationId property
- CorrelationIdHex property
- Feedback property
- GroupId property
- GroupIdHex property
- MessageId property
- MessageIdHex property
- QueueManagerName property
- QueueName property
- ReasonCode property
- ReasonName property

Methods:

- ClearErrorCodes method
- GetDistributionList method
- GetNextDistributionListItem method
- GetPreviousDistributionListItem method

Creation:

Use the AddDistributionListItem Method from the MQDistributionList class

Syntax:

set *distributionlist* = **MQDistributionList.** *AddDistributionListItem*(qname$, qmname$)

Parameters:

qname$ - name of the associated queue

qmname$ - name of the associated Queue Manager

AccountingToken property

Read-write. The AccountingToken to be included in the MQPMR of a message when put on a queue. Its initial value is all nulls.

Defined in: MQDistributionListItem class

Data Type: String of 32 characters

Syntax:

To get: *accountingtoken$ = MQDistributionListItem.***AccountingToken**

To set: *MQDistributionListItem.***AccountingToken** = *accountingtoken$*

AccountingTokenHex property

Read-write. The AccountingToken to be included in the MQPMR of a message when put on a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B" and so on.

You must supply 64 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQDistributionListItem class

Data Type: String of 48 hexadecimal characters representing 24 ASCII characters.

Syntax:
To get: *accountingtokenh$ = MQDistributionListItem.***AccountingTokenHex**

To set: *MQDistributionListItem.***AccountingTokenHex** = *accountingtokenh$*

CompletionCode property

Read-only. The completion code set by the last open or put request issued against the owning distribution list object.

Defined in: MQDistributionListItem class

Data Type: Long

Values:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *completioncode&= MQDistributionListItem.***CompletionCode**

CorrelationId property

Read-write. The CorrelId to be included in the MQPMR of a message when put on a queue. Its initial value is all NULLS.

Defined in: MQDistributionListItem class

Data Type: String of 24 characters

Syntax:
To get: *correlid$ = MQDistributionListItem* **.CorrelationId**

To set: **CorrelationId** =*correlid$*

CorrelationIdHex property

Read-write. The CorrelId to be included in the MQPMR of a message when put on a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B" and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQDistributionListItem class

Data Type: String of 48 hexadecimal characters reqpresenting 24 ASCII characters.

Syntax:
To get: *correlidh$ = MQDistributionListItem.***CorrelationIdHex**

To set: *MQDistributionListItem.***CorrelationIdHex** = *correlidh$*

DistributionList property

Read-only. The distribution list with which this distribution list item is associated.

Defined in: MQDistributionListItem class

Data Type: MQDistributionList

Syntax:
To get: *set distributionlist = MQDistributionListItem.***DistributionList**

Feedback property

Read-write. The Feedback value to be included in the MQPMR of a message when put on a queue.

Defined in: MQDistributionListItem class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *feedback& = MQDistributionListItem.***Feedback**

To set: *MQDistributionListItem.***Feedback** = *feedback&*

GroupId property

Read-write. The GroupId to be included in the MQPMR of a message when put on a queue. Its initial value is all NULLS.

Defined in: MQDistributionListItem class

Data Type: String of 24 characters

Syntax:
To get: *groupid$ = MQDistributionListItem*.**GroupId**

To set: *MQDistributionListItem*.**GroupId** = *groupid$*

GroupIdHex property

Read-write. The GroupId to be included in the MQPMR of a message when put on a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B" and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQDistributionListItem class

Data Type: String of 48 hexadecimal characters reqpresenting 24 ASCII characters.

Syntax:
To get: *groupidh$ = MQDistributionListItem*.**GroupIdHex**

To set: *MQDistributionListItem*.**GroupIdHex** = *groupidh$*

MessageId property

Read-write. The MessageId to be included in the MQPMR of a message when put on a queue. Its initial value is all NULLS.

Defined in: MQDistributionListItem class

Data Type: String of 24 characters

Syntax:
To get: *messageid$ = MQDistributionListItem*.**MessageId**

To set: *MQDistributionListItem*.**MessageId** = *messageid$*

MessageIdHex property

Read-write. The MessageId to be included in the MQPMR of a message when put on a queue.

Every two characters of the string represent the hexadecimal equivalent of a single ASCII character. For example, the pair of characters "6" and "1" represent the single character "A", the pair of characters "6" and "2" represent the single character "B" and so on.

You must supply 48 valid hexadecimal characters.

Its initial value is "0..0".

Defined in: MQDistributionListItem class

Data Type: String of 48 hexadecimal characters reqpresenting 24 ASCII characters.

Syntax:
To get: *messageidh$ = MQDistributionListItem*.**MessageIdHex**

To set: *MQDistributionListItem*.**MessageIdHex** = *messageidh$*

QueueManagerName property

Read-write. The MQSeries queue manager name.

Defined in: MQDistributionListItem class

Data Type: String of 48 characters.

Syntax:
To get: *qmname$ = MQDistributionListItem*.**QueueManagerName**

To set: *MQDistributionListItem*.**QueueManagerName** = *qmname$*

QueueName property

Read-write. The MQSeries queue name.

Defined in: MQDistributionListItem class

Data Type: String of 48 characters.

Syntax:
To get: *qname$ = MQDistributionListItem*.**QueueName**

To set: *MQDistributionListItem*.**QueueName** = *qname$*

ReasonCode property

Read-only. The completion code set by the last open or put issued to the owning distribution list object.

Defined in: MQDistributionListItem class

Data Type: Long

Values: See the *MQSeries Application Programming Reference*

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

Syntax:
To get: *reasoncode& = MQDistributionListItem*.**ReasonCode**

ReasonName property

Read-only. The symbolic name for the ReasonCode. For example: "MQRC_QMGR_NOT_AVAILABLE".

Defined in: MQDistributionListItem class

Data Type: String

Values: See the *MQSeries Application Programming Reference*

Syntax:
To get: *reasonname$ = MQDistributionListItem*.**ReasonName**

ClearErrorCodes method

Resets the CompletionCode to MQCC_OK and the reason code to MQRC_NONE for both the MQDistributionListItem class and the MQSession class.

Defined in: MQDistributionListItem class

Syntax:
MQDistributionListItem.**ClearErrorCodes**

GetDistributionList method

The Distribution List with which this Distribution List Item is associated.

Defined in: MQDistributionListItem class

Syntax:
distributionlist = MQDistributionListItem.**GetDistributionList**

GetNextDistributionListItem method

The next Distribution List Item object associated with this Distribution List.

Defined in: MQDistributionListItem class

Syntax:
*distributionlistitem = MQDistributionListItem.***GetNextDistributionListItem**

GetPreviousDistributionListItem method

The previous Distribution List Item object associated with this Distribution List.

Defined in: MQDistributionListItem class

Syntax:
*distributionlistitem = MQDistributionListItem.***GetPreviousDistributionListItem**

Appendix A. MQLSX link sample application

This appendix describes the MQLSX link sample application. This application is designed to show how, from a Notes environment, you can run a program in a non-Notes environment and get data returned to you. It demonstrates how you can use the MQLSX with a Notes agent and the link database. The design of the link database is the same as the link database in the original MQSeries link product (now obsolete).

In summary, what happens is:

- A Notes program takes the text you input, using the Notes form, and creates a Notes document in the Agent database
- A Notes agent creates an MQSeries message from the document and passes it to the MQSeries environment
- An MQSeries program changes the text in the message and creates an MQSeries reply message
- The Notes agent picks up the MQSeries reply message and updates the original document with the data

This sample is a way of implementing the MQSeries link (mqlink) using LotusScript and the MQLSX. The same link database is used in both cases.

This appendix describes:

- The components of the application, what they contain and the role of each individual part
- What you must do before you run the application
- How to run the application
- What happens when you run the application showing the movement of documents and messages
- Customization to change the data being passed

Look at the section on “What happens when you run the MQLSX link sample” on page 152, for the diagram that shows how the components fit together and their relationship with Notes and MQSeries.

MQSeries LotusScript Extension link sample application

The MQSeries LotusScript Extension link sample application (MQLSX link sample) allows you to develop a Notes application that can access an existing MQSeries application without the need to make any changes to the existing application.

It enables you to send data from a Notes application and get a reply, containing data, back from an MQSeries application. It demonstrates how you can use the MQLSX in conjunction with a Notes agent and the mqlink.nsf database, to interact with applications outside of Notes, using MQSeries.

Within your Notes application you write a structured message document that the link sample reads and processes in accordance with the requested entry in the link database.

It is designed to provide a base that you can change and extend to create your own applications.

Design of the MQLSX link sample

The components are:

- Notes LotusScript application client database (gmqlclnt.nsf)
- Agent database (gmqlagnt.nsf)
- link database (mqlink.nsf)
- MQSeries application (amqslnk0 sample program)
- MQSeries queue to receive messages (system.sample.notes.inqueue)
- MQSeries queue to return reply messages (system.sample.notes.outqueue)

Limitations

The limitations of this sample are:

- Both the message you must send and the reply message must be structured, of a predetermined length
- You can send the message to only one destination
- The reply message only updates the original Notes document

Limitations have been put on this sample to maintain a simple model that shows you how Notes and MQSeries can work together, and how the MQLSX can be used. The limitations are not related to the MQLSX.

Notes LotusScript application client database (gmqlclnt.nsf)

The Notes LotusScript application client database is made up of:

- A form named MQLSX Client Request.
  This is the form you use to enter the data you want to send to your MQSeries application. The agent database and link database document names are automatically displayed for you, but you can override these if you wish. You should enter the name of your agent server database.
  Enter data in the 'Data to be sent' field and select OK.

  Note: The type of data you enter must match the entry in the link database document you select.

  You also use this form to ask for a reply to a document you sent earlier. Display the document you sent earlier and select OK.
- A form named MQREQUEST.
  This is the form that the client application uses to create the document that it sends to the agent. The document it creates is held in the agent database and contains the data that is put into the MQSeries message. This document is also updated with the reply data (sent in a MQSeries message that is retrieved from the specified reply queue). The sample detects when a reply is available by looking at the status in the MQREQUEST document on the agent database. The values the status field can take are:

Value	Status of the request
1	Waiting
2	Available - reply message available
3	None - no reply needed
4	Error (for example, link entry error or send failure)

This form is displayed in the list when you run the sample, however it is for the sole use of the Notes sample program and it is not used for data entry.

- A form named MQCOMPLETE.
  This form is used to create a document when the reply data has been read from the MQREQUEST document on the agent database. The MQCOMPLETE document is sent to the agent database to indicate that the MQREQUEST document can be deleted.

  This form is displayed in the list when you run the sample, however it is for the sole use of the Notes sample program and it is **not** used for data entry.

- A form named MQSTARTAGENT.
  This form is used to create a document when you select OK on a MQREQUEST document, held on your own database, that you completed previously. The document is sent to the agent database, causing the agent to run and check to see if a reply has been received for this, or any other, MQREQUEST document.

  This form is displayed in the list when you run the sample, however it is for the sole use of the Notes sample program and it is not used for data entry.

- Any documents you create and save whilst running this application.

MQSeries sample program (amqslnk0)

This program is also provided by the majority of the MQSeries family of products. It reads a message from the SYSTEM.SAMPLE.NOTES.INQUEUE queue. If a reply is required (signified by the presence of a reply queue name in the document on the mqlink database), amqslnk0 changes the order of the input data and creates a reply message that it puts on the specified reply queue, for example, SYSTEM.SAMPLE.NOTES.OUTQUEUE.

MQLSX Agent database (gmqlagnt.nsf)

The MQLSX Agent database is made up of:

- The sample Agent program.
  The agent runs "If Documents Have Been Created or Modified".
  The agent accesses the specified 'link database document' to obtain the:
  - Format of the message expect by the MQSeries application.
  - Names of the queues to be used.
  - Details of the fields in the MQREQUEST document and the MQSeries reply message.
- An agent parameters form.
  This form is used to create an agent parameters document. Should you need to recreate it, it must contain the following information (needed by the agent):
  - Name of Server that the link database is on (or blank for local)
  - Name of the Queue Manager
  - Name of the link database
  - Whether or not character fields should be padded with spaces

  A sample agent parameters document is provided. You should modify it by changing the name of the server and queue manager to names that are correct for your system. This document must be available to the agent. Otherwise, an error is returned.

- A form named MQREPLYCONTEXT.
  This form is used to create a document that holds the control information that the agent needs to process a reply message for a specific request. An MQREPLYCONTEXT document is created for every message sent that requires a reply. The information held in this document is:
  - Request Document NoteID
  - Link Entry Document NoteID
  - MQSeries MsgId

  This enables an MQSeries reply message to be matched up with the correct MQREQUEST document. This form is for the sole use of the Notes sample program and it is not used for data entry.
- An MQREQUEST document.
  This document is sent by the client application and is a copy of the one held on the application database. It holds the:
  - Message data
  - Name of the link database document

  The agent adds the:
  - Reply data
  - Time the message was processed by the MQSeries application
  - Error messages if any errors were encountered
- The agent also maintains the status of the request document:

  This form is for the sole use of the Notes sample program and it is not used for data entry.
- An MQSTARTAGENT document.
  This document is sent by the client application when a reply to a previously sent message is requested. It activates the agent. This form is displayed in the list when you run the MQLSX link sample. However, it is for the sole use of the Notes program and it is not used for data entry.
- An MQCOMPLETE document.
  This document is sent by the client application when the reply to an MQREQUEST has been received. It indicates to the agent that the copy of the MQREQUEST document it identifies can now be deleted. This form is displayed in the list when you run the sample. However, it is for the sole use of the Notes sample program and it is not used for data entry.

The link database (mqlink.nsf)

The link database holds details of each 'request type' you use when you run the sample application. The 'request type' is the 'link database document name' you specify when you run the client application. The information in each link database document defines the mapping between the fields of the input form and the MQSeries message offsets. Each link database document also holds field type and offset information that is used by the agent to create an MQSeries message, and to process the reply message, if one has been requested.

The link database is the same as the one supplied with the MQSeries products. There is no need to start with an empty link database if you already have one and can make use of it.

A sample link database entry:

Each link database document you create must follow this structure. It is important to get the start and end positions of the fields within the MQSeries request message and reply message correct. Remember that the first character is in position zero.

- Entry
  The link database document name, the entry identifier. The MQENTRY name is used by the sample application.
- Database Information
  Information required by the existing MQSeries Lotus Notes link application. This is not used by the MQLSX link sample.
- Request Offsets
  This describes the layout of the message you send to the MQSeries application. The name you give each field is of no significance to the agent. However, the order you list the fields must match the order in which they appear in the request document, and the format must follow that shown.

  In this example:
  function 0 2 CHAR (the sample Notes program puts the characters Msg in this field)
  Msg 3 104 CHAR (this is where the data you entered is held)

  Note: You can have as many fields as you like to describe your message, however you can also have as little as one field. If each field is of the same type, there is more flexibility by having a single entry (more messages are likely to be able to use it).
- Reply Offsets
  This describes the layout of the data in the reply message. In each case you must describe how the information is held, whether it is character or numeric. If it is numeric, you specify whether it is big-endian (S390-Binary), or little-endian (Intel-Binary).

  For example:
  RData 0 101 CHAR (this is where the "Data received" information is in the reply message)
  Time 102 127 CHAR (this is where the "Time" information is in the reply message)
- Message Queuing Parameters
  This is where you specify the MQSeries queues you want to use and the format of data that the MQSeries message will contain.
  Queues
  If you do **not** want a reply from your MQSeries application, leave the reply queue name blank. If you do want a reply, the queue you specify must be local to your MQSeries application.

Message format
The format field is used when data conversion is necessary. The options are:

- Blank - Use this when the message contains non-character data.
- MQSTR - Use this when the fields in the message are all of type CHAR.
- User-defined format - Use this option when:
  - The message contains a mixture of character and non-character data
  - A user exit program is available at the server to enable MQSeries to provide the required conversion

- Error handling
  Here you enter the validation you want to take place before your Notes document is updated, and specify what information you want reported with the error . When the error conditions are met, an error message is displayed in the originating (MQLSX Client Request) document.

  For example:
  Error condition: client 0 5 CHAR
  Syntax for additional information: RData 0 101 CHAR
  In this case, a Notes mail memo, containing the reply message data (but not the time), is created if the reply message has the word client as the first six characters of the reply message. If you want to check for a 'not equal' value, prefix the character string with an exclamation mark (!).

  Note: The comparison of characters **is** case sensitive.

Before you run the MQLSX link sample

This sample will not run successfully until you have completed the following:

- Install the MQLSX Agent database (gmqlagnt.nsf) and the link database (mqlink.nsf) on your Notes server. Select File - Database - New Copy.
- Install the MQLSX Client database (gmqlclnt.nsf) on your Notes Client.
- Ensure the queue manager you want to communicate with is running.
- Update the MQSeries Agent Parameters document, in the MQLSX Agent database, to reflect the name of the link database, the Notes server (initially set to Your Server Name), and the MQSeries queue manager you are using. The link database must be local and the name blank if your agent is running on a server.
  Check the box if you want the message data you send each time to be padded out with blanks.
- Change the name of the Notes server (that the MQLSX agent will run on) in the MQLSX Agent database. Initially the name of the server is given as Your Server Name. To change this:
  - Open the MQLSX Agent database (gmqlagnt.nsf)
  - Select Agents - MQLSX Agent - Schedule
  - Change the box 'Run only on .....'
  - Save
- Create the queues to hold the MQSeries request and the reply messages. The sample uses SYSTEM.SAMPLE.NOTES.INQUEUE and SYSTEM.SAMPLE.NOTES.OUTQUEUE. To create these queues, process

amqslnk0.tst using the MQSeries runmqsc commands utility (runmqsc QueueManagerName < amqslnk0.tst).
If you want to use different queues, you must change the Message Queuing Parameters in the MQENTRY document or use a different link database document.

- There is only one sample document in the link database (mqlink.nsf) called MQENTRY. If you want to add your own document to the link database:
  Open the database
  Select 'Create'
  Select 'External-Call Parameters'
  Complete the form (there is no validation performed against the data you enter)
  Select File - Save
- If you want to define your own MQSeries message formats, you must write the corresponding exit routine to perform data conversion for the destination (server) queue manager if you require it.
- If you run the MQLSX link sample application, you may need to change the Agent settings. In particular:

  ```
  AMgr_DocUpdateAgentMinInterval=1
          (default is 30)

  AMgr_DocUpdateEventDelay=1
                       (default is 5)
  ```

  Setting these to 1 will cause the Agent to run as soon as it can. However, you should set them according to the workload on your Agent. See the section "About the NOTES.INI file" in the *Lotus Notes Administration Guide* for more information.

Running the MQLSX link sample application

1. Check that all the appropriate set-up work has been completed. See “Before you run the MQLSX link sample” on page 150 for details.
2. Start MQSeries program, amqslnk0, running in your MQSeries environment. From a command line, enter

   ```
   amqslnk0 (-q InputQName) (QMgrName)
   ```

   where the default InputQName is SYSTEM.SAMPLE.NOTES.INQUEUE on the default queue manager.
3. Single click on the MQLSX Client icon, this selects the database (or open the database by double clicking).
4. Select Create from the tool bar.
5. Select MQLSX Client Request to run the client application.
6. Use the tab function to move from one field to another. Change the Agent server and database names if you need to use different ones from those displayed.

   To change the default names displayed by the client application:
   - Select Design - Forms
   - Double click on the MQLSX Client Request form name
   - Select name of the field you want to change against Define

- Select 'Default Value' against Event
- Enter new default value

7. Change the link database document name if you want to use a different one from the default.
8. Enter the data you want to send to the MQSeries application.
9. Click OK.

   If the link database document you have chosen does not include the name of a reply queue, 'Reply Message is not expected ' is displayed in a message box. If the link database document does include the name of a reply queue, the sample will display the reply data and the time. It waits approximately 10 seconds for the MQSeries reply message to appear on the queue. When this time is exceeded, you are given the option of retrying (select OK) or cancelling. If you retry, it has the same 10 second threshold, after which the process is repeated.
   If you select cancel, you must save the document before exiting the client application if you want to get the reply at a later time.

To request the reply to a request document you sent earlier

1. Double click on the MQLSX Client icon to open the database.
2. A list of documents you have saved are displayed in the sample view. Select the one you are interested in (double click).
3. Click OK.

What happens when you run the MQLSX link sample

When you run this application, using the unmodified MQLSX Agent database and link database document, it composes a document that includes a request for reply information from the MQSeries program (amqslnk0 or an application of your own), and concludes by updating the original Notes document with the requested information.

Components of the MQLSX link sample application (request for reply data)

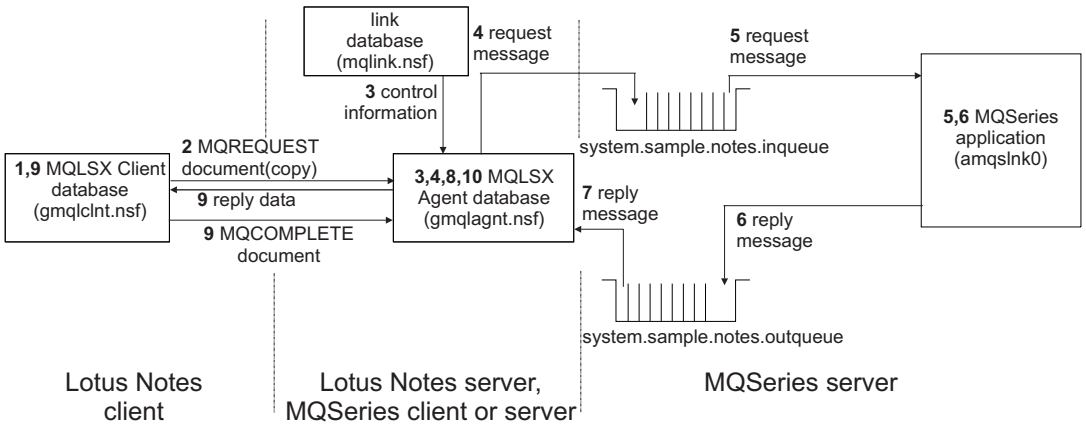

Figure 1. Components of the MQLSX link sample application

Using the MQLSX Client request form in the Client database, you enter the data you want to send and Click OK.

1. Using the sample input form on the Notes client, you enter the data you want to send. The input form also allows you to specify the link database document name, and the names of the agent server and database.
2. Click OK. A LotusScript program associated with the OK button composes a document using the MQREQUEST form. This MQREQUEST document is created on the client database (gmqlclnt.nsf) and copied to the Agent database (gmqlagnt.nsf).
3. The agent, which is 'change activated', runs when it detects the new document and reads it. Using the information in the Agent Parameters document, the agent:
   - Extracts the name of the link database document
   - Reads the link database document
   - Connects to the queue manager
   - Opens the input queue (for example, SYSTEM.SAMPLE.NOTES.INQUEUE)
   - Constructs an MQSeries message, using the information in the link database document as well as the MQREQUEST document
   - Puts the MQSeries message on the queue
   - Creates an MQREPLYCONTEXT document if a reply message is expected
   - Looks for replies for any MQREPLYCONTEXT documents
4. The MQSeries sample server application, amqslnk0, gets the message from the queue (it uses the MQGMO_WAIT option) and processes it. If the data in the message is character, the words in the message are reversed, for example "Hello World!" becomes "World! Hello".

   If a reply has been requested:

5. The program amqslnk0 creates the MQSeries reply message, to include the time as well as the data, placing the message on the reply queue.
6. The Agent gets the MQSeries message from the reply queue specified in the MQREPLYCONTEXT document (for example, SYSTEM.SAMPLE.NOTES.OUTQUEUE).
7. The Agent:
   - Uses the MQREPLYCONTEXT document to match the reply up with the MQREQUEST document it has on its database
   - Updates the status of the MQREQUEST document to '2' to indicate that the message is now available
   - Deletes the MQREPLYCONTEXT document
8. The LotusScript client application associated with the input form:
   - Polls the agent database to check for a change in status of the MQREQUEST document to '2'. It does this by sending null StartAgent documents to the Agent database, each time OK is clicked, to regularly activate the Agent to perform reply message processing
   - If the reply is not available and 10 seconds have passed, the sample displays a message box giving you the option to retry, or cancel the request

- If you chose to cancel the request, save your document if you want to collect the reply at a later time
- If you retry, the sample continues to poll for a change in status of the MQREQUEST document on the agent database for a further 10 second period
- When the message is available, the sample updates the MQREQUEST document on the MQLSX Client database (gmqlclnt.nsf) and displays the reply data and time
- Sends an MQCOMPLETE document to the agent

9. The agent:
   - Runs when the MQCOMPLETE document arrives
   - Deletes the MQREQUEST document that the MQCOMPLETE document refers to

   When you request a reply to an MQREQUEST document at a later point in time:
   - The sample sends a MQSTARTAGENT document to the agent
   - The agent runs:
     – Checks to see if there are any MQREPLYCONTEXT documents
     – Checks to see if there are any messages on the reply queues identified by the MQREPLYCONTEXT documents
     – Processes the messages as normal, using the MQREPLYCONTEXT and MQREQUEST documents it has in its database

Error handling in the MQLSX link sample application

Errors checking takes place throughout the MQLSX sample application components. In addition to Notes and LotusScript errors that can occur, there are also MQLSX error situations that can cause error messages specific to the MQLSX to be displayed.

MQLSX Client Error messages

Errors detected by the MQLSX Client are displayed using a message box.

The following error, warning, and informational messages may be displayed by the MQLSX client:

```
ERROR: Error saving document.
```

An attempt to save a document in the current database was unsuccessful.

```
ERROR: Request failed.
```

A detailed error message is displayed in the current document (error_field_msg field).

```
ERROR: The agent document required by this request is not
available and may have been deleted.
```

The MQREQUEST document in the agent database could not be copied to the client and may have been deleted.

```
ERROR: Open Agent database failed.
```

The agent database on the Notes server could not be opened.

```
WARNING: Request is already complete. Resend?

INFO: Reply message is not expected.

INFO: Reply message not available. Retry?
```

MQLSX Agent error messages

Errors detected by the MQLSX Agent are printed to the Notes server console. An error can be detected by:

- Lotus Notes
- MQLSX
- The link sample

Where an error is detected by Notes or the MQLSX, the link sample adds further information to the error message before printing it to the console.

The format of a message therefore is different to what you would traditionally expect:

Errors detected by Lotus Notes

The format of these messages output by the link sample is:

MQLSX link Agent Notes Error: Agent_Insert_String Notes_Error_Message (Notes error = Notes_Error_Number Line nnnn)

where:

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank

Notes_Error_Message
is the LotusScript error message

nnnn
is the line number in the LotusScript source code where the error occurred

Errors detected by the MQLSX

The format of these messages output by the link sample is:

MQLSX link Agent Error: Agent_Error_Number Agent_Error_Message Agent_Insert_String MQLSX_Error_Message (Notes error = Notes_Error_Number Line nnnn)

where:

Agent_Error_Number
is an agent error number that may be omitted for MQLSX detected errors if zero

Agent_Error_Message
is an agent error message

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank

MQLSX_Error_Message
is the LotusScript MQLSX error message and will typically provide the MQSeries or MQLSX reason code

nnnn
is the line number in the LotusScript source code where the error occurred

Errors detected by the link sample Agent
The format of these messages output by the link sample is:

MQLSX link Agent Error: Agent_Error_Number Agent_Error_Message Agent_Insert_String

where:

Agent_Error_Number
is an agent error number that may be omitted for MQLSX detected errors if zero

Agent_Error_Message
is an agent error message

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank.

The error messages you can encounter that are detected by the Agent component of the link sample are:

`67001 View not found.`

The 'Agent Parameters' View was not found in the agent database.

`67002 Parameter document not found.`

The agent parameters document was not found in the agent database.

`67003 link Entry Error: start or end position value not valid.`

An error was detected in a link database entry as indicated.

`67004 Link Entry Error: one or more field items are missing.`

An error was detected in a link database entry as indicated.

`67005 Link Entry Error: data type not valid.`

An error was detected in a link database entry as indicated.

`67006 Link Entry Error: data length not valid for NUM.`

An error was detected in a link database entry as indicated.

The data length specified for fields of type NUM, INTEL-BINARY or S390-BINARY must be 2 or 4.

`67007 Link Entry Error: field length value too big.`

An error was detected in a link database entry as indicated.

The field length for fields of type CHAR may not exceed 32000

```
67008 link database could not be opened.
```

The specified link database could not be opened.

```
67009 Reply processing error.
```

A reply processing error occurred. More information, that may include a reason code, is provided in the accompanying insert string.

```
67010 Backout threshold of failing message exceeded; message
will be discarded.
```

A failing message that was previously backed out was discarded. Information about the exact cause of the failure is provided in earlier error messages.

```
67011 Numeric value is not valid or causes overflow.
```

An non-numeric or out-of-range numeric value was entered by the user or returned in an MQSeries reply message.

Two-byte NUM, INTEL-BINARY, and S390-BINARY values numbers are unsigned numbers in the range 0 to 65535.

Four-byte NUM, INTEL-BINARY, and S390-BINARY values are signed numbers in the range -2 147 483 648 to 2 147 483 647.

The Agent_Insert_String may be blank or one of the following:

```
Link Entry document not found(used with Reply processing error)

Request document not found(used with Reply processing error)

Error accessing queue manager(used with MQLSX detected error)
```

Note: If the error

```
"Cannot forward declare CLASS or TYPE"
```

occurs, this is an intermittent problem that may be cured by forcing a recompile of the LotusScript code.

MQLSX Client Request error messages

The following messages may be displayed in the MQLSX Client Request document (in the error_field_message field):

```
Reply message failed link entry error condition checks.
```

The reply message failed the specified link entry error condition checks.

```
Link Entry Document Error.
```

The specified link entry document cannot be opened.

```
Numeric value is not valid or causes overflow.
```

An non-numeric or out-of-range numeric value was entered by the user or returned in an MQSeries reply message.

Two-byte NUM, INTEL-BINARY, and S390-BINARY values numbers are unsigned numbers in the range 0 to 65535.

Four-byte NUM, INTEL-BINARY, and S390-BINARY values are signed numbers in the range -2147483648 to 2147483647.

```
Backout threshold of failing message exceeded; message will be
discarded.
```

A failing message that was previously backed out was discarded. Information about the exact cause of the failure is provided in earlier error messages.

```
An error was detected by the agent.
Consult the agent messages for more information.
```

This provides indication to the user that the agent detected an error details of which are provided in the agent messages.

Customizing the MQSeries link sample application

There are several ways in which you can change this sample, yet still get the benefit of using it without having to do any further programming.

You change the MQLSX Client database (gmqlclnt.nsf):

- If you want to change the length of time that the program waits before checking to see if there is a reply message
- If you want to send or receive anything other than the fields defined in the sample

You change the MQLSX Agent database (gmqlagnt.nsf):

- If you want to send or receive anything other than the fields defined in the sample
- If you want to change the name of the Notes server the link database runs on

  Note: The name of the Notes server can only be different from that of the agent if the agent is run manually.

Changing the wait time

Unless changed, the sample waits for 10 seconds before it checks to see if a reply message is available. To change this:

- Open the MQLSX Client database (gmqlclnt.nsf)
- Select Design - Forms - MQLSX Client Request
- Click on the Define OK button
- Select the Event Declarations option
- Change the value WAIT_TIME from 10 to the new value (in seconds)
- Save the form

Changing the name of the Notes Server and Queue Manager for mqlink.n

To change this:

- Open the MQLSX Agent database (gmqlagnt.nsf)
- Under Folders & Views in the navigator, Select Agent Parameters
- Double click on mqlink.nsf
- Select Action - Edit Document
- Change the necessary fields and save the document

Changing the send and reply fields

Unless changed, the sample sends two fields of data (known as Func and SData) to the MQSeries application, and receives two fields of data (known as RData and Time) back in the reply message.

The fields sent are:

1. A fixed text string of three characters 'Msg'. The field name on the form is Function, in the code this is known as Func.
2. A field that contains the data you enter when you run the sample. This may or may not be padded out with spaces, it depends on what you select in the Agent Parameters document. This field is known as SData.

Changes are necessary to both the MQLSX Client database, the MQLSX Agent database and the link database:

MQLSX Client database

In this database you need to make changes to the:

- MQLSX Client Request form
- MQREQUEST form (to be copied to the MQLSX Agent database)

Looking at the MQLSX Client Request, you will find the code for the classes under '(Declarations)' and the code that calls the classes under the Event 'Click'.

MQLSX Agent database

In this database you need to make changes to the:

- MQLSX Agent
- MQREQUEST form (may be copied from the MQLSX Client database)
- Agent Parameters form

Looking at the MQLSX Agent, you will find the code for the classes under '(Declarations)' and the code that calls the classes under the Event 'Initialize'.

Link database

In this database you need to make changes to the:

- External Call Parameters document (or create a new one)

Note: If the agent is running on a server, the link database must reside on the same server as the agent. The link database can only be on a different server from the agent if the agent is run manually.

Steps to take

The following lists the areas that you need to modify if you want to send or receive a different number of fields of data to that supplied within the sample:

- **MQREQUEST form:**
  - Open the MQLSX Client database (gmqlclnt.nsf)
  - Select Design - Forms - MQREQUEST (double-click)
  - Modify the form to include the fields you want and save it
  - Copy the form to the MQLSX Agent database (or make sure the form on the MQLSX Agent database has the same changes made to it)
- Class/method code of the MQLSX Client Request form:
  - Open the MQLSX Client database (gmqlclnt.nsf)
  - Select Design - Forms - MQLSX Client Request (double-click)

  If you want to change the form:
  - Modify the form to include the field you want and save it

  If you want to change information relating to the field on the form, such as the Agent server name, the default values of one or more fields, or Define OK button.

  To change the Define OK button:
  - Reveal the lower pane to display the definitions and script
  - Select Define OK(button) and Event '(Declarations)'
  - Add/modify the Property Set statements in the TempDocument class to cover each piece of data you want in the send and reply messages
  - Change the constant declaration MQ_MSG_FUNCTION, to alter the field 'Function' (the sample puts a text string 'Msg' in this filed) sent, or delete it if it is not required by your MQSeries application
  - Add a Property Set statement to the CurrentDocument class for each piece of data you want in the reply message and change the ReInitialize method to initialize these properties
  - Change the CopyMqMsgItems method in the CurrentDocument class to deal with the properties you now have in the send and reply messages
  - Modify the UpdateWithReply method in the RequestDocInfo class to process the properties you now have in your reply message
- **Class/method code of the MQLSX Agent:**
  - Open the MQLSX Agent database (gmqlagnt.nsf)
  - Select Agents and double click on MQLSX Agent
  - Reveal the lower pane to display the 'What should this agent do?'
  - Select Event ('Declarations)'
  - Change the Get Data and Set ReceiveData properties in the RequestDocument class to include all the properties you now have in the send and receive messages

  - Change the New method in the MQReplyMessage class to process all the properties you now have in the reply message

- **Link database:**
  - If you change an existing document, you modify the Request Offsets and Reply Offsets
  - Alternatively you can create a new document from the External Parameters Call form

Data conversion

If your MQSeries application and your Notes Server are running on platforms that have different character sets or different integer encodings, the data from your Notes application has to be converted. This can be performed by the MQLSX, the application, or by MQSeries on behalf of the application.

These are the steps that take place:

1. The agent builds the MQSeries request message using the information in Request Offsets in the link database entry. The format options are:

   CHAR
   The field is copied across unchanged and so appears in the codepage used by the Notes Server.

   NUM
   The field is encoded as a 2-byte or 4-byte, signed or unsigned, integer in Intel or S/390-format according to the platform on which the agent is running. Use this format if conversion is performed at the receiver by MQSeries or the application.

   INTEL-BINARY
   The field is encoded as a 2-byte or 4-byte Intel format signed or unsigned integer.

   S/390-BINARY
   The field is encoded as a 2-byte or 4-byte S390-format signed or unsigned integer. For messages received by the agent, both INTEL_BINARY and S/390_BINARY formats are treated the same as NUM, with data conversion provided by the MQLSX.

   Two-byte NUM, INTEL_BINARY, and S/390-BINARY value numbers are unsigned numbers in the range 0 to 65535

   Four-byte NUM, INTEL_BINARY, and S/390-BINARY value numbers are signed numbers in the range -2 147 483 648 to 2 147 483 647.

2. When the agent puts the message on the request queue, it sets the message descriptor format field to the value given in the message format field of the link database entry. The possible values of the message format are:

   Blank
   Use this option when your application performs any necessary data conversion. Your application must check against the encoding and CodedCharSet fields in the message descriptor.

   MQSTR
   Use this option when all the fields in the message are of type CHAR. This indicates to MQSeries that the application data is all characters and MQSeries does any necessary data conversion. This is not recommended if you are

converting between single-byte and double-byte character sets, as individual fields may expand or contract.

User-defined format
Use this option when a user exit program is provided at the receiver to enable MQSeries to perform the necessary conversion.

For more information see “Data conversion” on page 18

Designing your own applications using mqlink.nsf

When designing your application to incorporate Notes, there are some configuration factors that you need to consider:

- The MQLSX Client database can be installed on a Notes client or Notes server.
- The MQLSX Agent database must be run on a Notes server (unless the agent is to be run manually), with either an MQSeries client or MQSeries server installed.
- The link database (mqlink.nsf) must be on the same Notes server as the agent (unless the agent is to be run manually).
- Your MQSeries application must run on an MQSeries platform that supports your Agent environment.
- Messages put by the Agent are persistent.
- The Agent is not designed to be triggered by an MQSeries message.

Appendix B. MQLSX link extra agent sample application

This appendix describes the MQLSX link extra agent sample application. This application is designed to show how, from an enterprise application, you can run a program that results in an update to a Notes database.

The Notes agent in this sample can be run manually or as a scheduled agent, but it can also run using a trigger monitor. The MQSeries trigger monitor for Lotus Notes agents can be used with both MQLSX and MQEI applications.

For more information see the *MQSeries Trigger Monitor for Lotus Notes agents User Guide* (MA7E). You can download this support pac from the MQSeries Web site

`http://www.software.ibm.com/ts/mqseries`

This appendix describes:

- The components of the application, what they contain and the role of each individual part
- What you must do before you run the application
- How to run the application (manually, as a scheduled agent, or using the trigger monitor)
- What happens when you run the application
- How you can change the application (customization)

Look at Figure 1 on page 152 which shows how the components fit together and their relationship with Notes and MQSeries.

Introduction to the MQLSX link extra agent sample application

The MQLSX link extra agent sample provides enterprise applications with a means of sending data to a Lotus Notes environment via MQSeries. This data can be used to update or add one or more documents to a Notes database. Such updates may span more than one database. You specify what action to take using a combination of key fields and rules in a separate database (link database). An exit is provided for special error handling.

The MQSeries link extra agent runs as a Notes agent on either your Domino server or on your Notes client.

The function provided in the MQSeries Trigger Monitor for Lotus Notes Agents allows an application on any of the MQSeries supported platforms to initiate the sending of data to the Lotus Notes Server or workstation. The MQLSX link extra agent sample allows data from a non-Notes environment to be used to update Lotus Notes databases. A link database provides mapping between the system application and fields defined in the Lotus Notes databases.

Components

The MQLSX link extra agent sample consists of the following components:

- Agent database (gmqlxtra.nsf)
- The link database (mqlinkx.nsf)
- Lotus Notes sample applications databases (test1.nsf & test2.nsf)
  - test1.nsf contains salary information
  - test2.nsf contains address information
- Test application files (mqlxdemo.*)

Lotus Notes agent database (gmqlxtra.nsf)

As well as the agent program, this database contains a Notes document providing the default parameters needed if the agent is not started by the trigger monitor, or the default parameters used by the trigger monitor.

The defaults provided are:

Table 1. Lotus Notes Agent Database default parameters

QName:	NOTE
QMgrName:	Blank (Default queue manager)
Envdata:	mqlinkx.nsf

Lotus Notes link extra database (mqlinkx.nsf)

This is a Lotus Notes database that you need to populate with entries to describe how your MQSeries message data maps to fields in a given Lotus Notes document. The Lotus Notes link extra database is an extension of the MQSeries link for Lotus Notes database and can be used to define entries for both MQSeries link and MQSeries link extra by using different forms.

The information held in a document on the database includes:

- The entry name from the link database document. This name matches that of the MQSeries queue where messages to be processed by MQSeries link extra agent are retrieved.
- The name of an optional view to be used to search for a Notes document (a faster alternative to searching the entire database).
  If this option is used, the first column of the view must contain the data for the first key field specified for that entry.
- Logical processing rules (For example, If found Insert, Insert always, and so on,) for each message retrieved.

  Note: If you are already using the MQSeries link extra SupportPac, you can use exactly the same database with this sample.

Lotus Notes sample applications databases (test1.nsf & test2.nsf)

Two sample Lotus Notes demonstration databases are provided with MQSeries link extra agent sample:

- test1.nsf contains salary information
- test2.nsf contains address information

The Link database has entries to enable documents contained in the demonstration databases to be updated. If you create your own applications, suitable Link database entries must be provided.

Note: The demonstration example does not automatically use the document search by view function.

MQSeries sample application (mqlxdemo files)

An MQSeries demonstration application is provided to help you get started. This demonstration application is a C program designed to read messages from standard input. A data file is provided for use with the demonstration application. When the data file is piped to the demonstration application, messages are put that the MQLSX link extra agent uses to update the Lotus Notes demonstration databases. The demonstration application can use triggering to start the MQLSX link extra agent.

Comparison with the MQSeries link extra for Lotus Notes SupportPac

This sample uses exactly the same

- Set of rules
- mqlinkx.nsf database

The differences are:

- The mqlinkx executable file is replaced by a Notes agent database (gmqlxtra.nsf).
- The MQLSX link extra agent sample can be run using a trigger monitor (the alternative is to run it manually or as a scheduled agent).
- When a message cannot be processed and the retry limit is exceeded, the MQLSX link extra agent sample commits the message in order to remove it from the queue. If there is a backout requeue queue or a dead-letter queue defined, the failing message is put on it.
- The 'Additional Selection Formula', an optional parameter in the mqlinkx.nsf database, must conform to the LotusScript FTSearch rules (whereas previously this conformed to the Notes formula rules).
- The 'Additional Selection Formula' parameter may be used in addition to the 'View to Use for Searching' (whereas previously it could not).

  Caution: If you specify numeric KEY fields in the link database, you **must** create a full text index for the target database.

Restrictions

- The agent cannot be used with the MQSeries Trigger Monitor for Lotus Notes agents on UNIX platforms with versions of Notes 4.5 and earlier.
- User messages are only provided in US English.

Recommendation

Triggered Notes agents should not be long-running; they should allow themselves to be retriggered.

Design of the MQLSX link extra agent sample

The MQLSX link extra agent sample provides the same function as MQSeries link extra for Lotus Notes, with the addition of application type (APPLTYPE) 22 to support the triggering of Notes agents.

The add-in task (mqlinkx.exe) provided by MQSeries link extra for Lotus Notes is replaced by a Notes Agent (gmqlxtra.nsf).

The trigger monitor uses the MQSeries MQTMC2 structure. The only fields in this structure used by the Notes agent are QName, QMgrName, and EnvData. The EnvData field is used to hold both the name of the link database and the link database server name.

Note: The name of the Notes database may not included spaces, however the link database server may have a name that includes spaces. A space must be added in EnvData after the name of the database, before the name of the link database server.

The trigger message document (using the MQTMC2 structure) may originate from:

- A trigger document passed from the trigger monitor
- An equivalent document in the Agent database (when you run the Notes agent manually)
- A combination, with the queue and queue name parameters supplied within the document from the trigger monitor and the link database and server names provided by the document in the Agent database

If neither document is found, the default values are used. If the value of a parameter is unspecified, the default value is used.

The QName field in the trigger message document, as with any document using the link database, is compared to the 'Entry' field in the link database. When a match is found, the trigger document is processed using the conditions and rules in the corresponding document in the link database. After successfully applying the rules of the link document, the link database is searched to find a further matching 'Entry'. Processing of the trigger document ends when no further matching entries are found.

Error processing

If the agent is unable to process a message, and the retry limit is exceeded, the Notes agent:

1. Attempts to queue the message on the backout requeue queue (the name of which is held by the application queue) after which it commits the message

   If this fails:

2. Puts the message onto the dead-letter queue (if one is defined to the queue manager) after which it commits the message

   If this fails:

3. Commits the message, causing it to be removed from the application queue and discarded.

For information on triggering, see the *MQSeries Application Programming Reference Manual.*

Notes agent user exit

The Notes agent provides you with the option of using a LotusScript user (error) exit. The sample user exit provided is invoked when an error is detected. It outputs a message for information (the value of the MSG_USEREXITINVOKED string constant), and returns "Stop and Exit", at which it will stop the agent. An alternative return value SCERR_OK_TO_CONTINUE can be used to indicate that the exit has successfully handled the error and it is OK for the agent to continue.

```
'*********************************************************
'*  Sample LotusScript UserExit function
'*
'*  UserExit input parameters:
'*  Rule    As String
'*  Rule (one of: "0","1","2","3","4","5","6","7","8","9" )
'*  RuleDoc As NotesDocument  rule document
'*  Msg     As MQMessage      MQMessage
'*  QMgr    As MQQueueManager MQQueueManager
'*  UserDb  As NotesDataBase  User database
'*  UserExit As Long returns one of:
'*  SCERR_OK_TO_CONTINUE:
'*      the message was processed successfully and should
'*      be committed
'*  SCERR_STOP_AND_EXIT:
'*      the message was not processed successfully and should
'*      be rolled back.
The agent should terminate.
'*********************************************************

Function UserExit( Rule As String,  RuleDoc As NotesDocument,
Msg As MQMessage,
               QMgr As MQQueueManager, UserDb As NotesDataBase)
As Long
     Messagebox MSG_USEREXITINVOKED
     Print MSG_USEREXITINVOKED

     UserExit = SCERR_STOP_AND_EXIT
'    UserExit = SCERR_OK_TO_CONTINUE
End Function
```

Setting up the MQLSX link extra agent sample

If you are installing the MQLSX link extra agent sample for the first time or you already have it, follow the instructions under “Upgrading from the MQSeries link extra for Lotus Notes SupportPac”

Prerequisites

MQSeries link extra agent requires Lotus Domino Release 4.5 or later, and the following levels of MQSeries:

- MQSeries client on HP-UX
- MQSeries client on OS/2
- MQSeries client on Sun Solaris
- MQSeries client on Windows 3.1
- MQSeries client on Windows NT
- MQSeries for Windows NT Version 5.1 (for the server)

If you choose to use one of the MQSeries client environments, connect it to an MQSeries server that supports it. This can be any MQSeries server that supports the MQSeries client, and does not have to be a server capable of running Notes.

Installing the MQLSX link extra agent sample for the first time

1. Copy the four Lotus Notes database files that are provided in the MQSeries link extra agent sample for Lotus Notes package into a suitable directory on your Notes workstation:
   - mqlinkx.nsf
   - gmqlxtra.nsf
   - test1.nsf (Salary Information database)
   - test2.nsf (Address Information database)

   Note: These databases (test1.nsf and test2.nsf) do not use the document search by view function.
2. Add icons for these databases to your work space.
3. Install the databases on your server if required using File - Database - New Copy and add the icons to your work space.

Upgrading from the MQSeries link extra for Lotus Notes SupportPac

Install only gmqlxtra.nsf, test1.nsf, and test2.nsf as described when installing for the first time. The other databases are already available as part of the MQSeries link extra for Lotus Notes SupportPac.

Setting up MQSeries to run the MQLSX link extra agent sample

1. Start the MQSeries queue manager.
2. Edit the MQSeries MQSC file command, gmqstnm0.tst (MQSeries Trigger Monitor for Lotus Notes), to match your environment. Replace the existing versions of test1.nsf and test2.nsf with the new versions, to take advantage of

the performance benefits, as these databases specify a view name for searching.

The user exit requires a queue to put error message details. The error queue name suffix must be the same as the name of the application queue name you are putting messages to. If you want to run the sample program mqlxdemo.exe, the MQSeries definition is already set up to use "NOTE" as the application queue name and error message queue name suffix. If you want to create other application queues, you must modify the gmqstnm0.tst file to maintain this relationship.

3. Use runmqsc to create the MQSeries link extra agent queue manager objects. Issue the command:

   ```
   runmqsc QMgrName < gmqstnm0.tst > mqlx.out
   ```

   Where:

   - QMgrName is the name of the queue manager; if you don't specify a value the default queue manager is used.
   - gmqstnm0.tst is the name of your MQSC command file.
   - mqlx.out is the name of the file that contains the runmqsc output results.

4. Check the output of runmqsc. The following queue manager objects should now exist:

   Queues:

   ```
   NOTE           (The MQSeries link extra agent application queue*)

   MQLX.HOLD.NOTE (The user exit error message queue)

   SYSTEM.SAMPLE.NOTES.AGENT.INITQ   (The initiation queue)
   ```

   Process:

   ```
   SYSTEM.SAMPLE.NOTES.AGENT.PROCESS (Process definition to

                  start MQSeries link extra agent process
   mqlinkx.exe)
   ```

   Note: The value of the ENTRY field used to search the MQSeries link database will be equal to the queue name.

Before you run the MQLSX link extra agent sample

There are a number of steps you must complete before you can successfully run the MQLSX link extra agent sample.

You can either run the MQSeries link extra agent sample manually, as a scheduled agent, or by using triggering using the MQSeries trigger monitor for Lotus Notes Agents. If you choose to run the MQSeries link extra agent sample using triggering, you should verify that MQSeries link extra agent can initiate updates to Lotus Notes.

If you want to start it by using triggering with the MQSeries trigger monitor for Lotus Notes Agents, you must:

1. Start the MQSeries queue manager
2. Create an MQSeries application program queue

3. Create an MQSeries agent initiation queue
4. Create an MQSeries agent process definition
5. Start the Trigger Monitor program

Start the MQSeries queue manager

Before you can create the queues you need to run this sample, you must start your MQSeries queue manager. This can be the default queue manager, or a user-defined queue manager.

Create an MQSeries application program queue

You can create an MQSeries application program queue by using the supplied MQSC sample script (gmqstnm0.tst) or by using the runmqsc command shown below:

```
DEFINE QLOCAL('NOTE') REPLACE +

DESCR('Sample Notes Agent Application Program Queue') +

INITQ('SYSTEM.SAMPLE.NOTES.AGENT.INITQ') +

PROCESS('SYSTEM.SAMPLE.NOTES.AGENT.PROCESS') +

TRIGGER +

TRIGTYPE(FIRST) +
```

Table 2. runmqsc command definitions for creating an application program queue

Definition	Purpose
DESCR	Specifies a descriptive plain-text comment.
INITQ	Specifies the name of the initiation queue.
PROCESS	Specifies the name of the agent process definition.
TRIGGER	Specifies that triggering is active for this queue.
TRIGTYPE(FIRST)	Specifies that a trigger message should be written when the first message of suitable priority arrives on this queue.
TRIGDATA	Specifies information required by the agent to process messages from this queue. For the IBM MQSeries link LotusScript Extension (MQLSX), this optional.

Note: TRIGDATA is typically not used and can be specified as blank or omitted if not required.

Create an MQSeries agent initiation queue

You can create an MQSeries agent initiation queue by using the supplied MQSC sample script or by using the runmqsc command shown below:

```
DEFINE QLOCAL('SYSTEM.SAMPLE.NOTES.AGENT.INITQ') REPLACE +

DESCR('Sample Triggered Notes Agent Initiation Queue')
```

DESCR specifies a descriptive plain-text comment.

Create an MQSeries agent process definition

You can create an MQSeries agent process definition by using the supplied MQSC sample script or by using the runmqsc command shown below:

```
DEFINE PROCESS('SYSTEM.SAMPLE.NOTES.AGENT.PROCESS') REPLACE +

DESCR('Sample Triggered Notes Agent Process') +

APPLTYPE(22) +

APPLICID('gmqlxtra.nsf MQLSX link extra Agent') +

USERDATA('User data used by MQLSX link extra Agent')
```

Table 3. runmqsc command definition for creating an agent intitiation queue

Definition	Purpose
DESCR	Specifies a descriptive plain-text comment.
APPLTYPE	APPLTYPE 22 specifies the application type as a Notes agent.
APPLICID	Specifies the name of the agent database (*.nsf) file followed by the agent name. Note that the agent name may contain spaces while the agent database file name may not. You must specify the fully qualified path and filename of the agent database if it is not the default Notes directory.
USERDATA	Specifies user data required by the agent. You can specify this as a blank or just omit it if you don't require it.

Start the Trigger Monitor program

You may want to start the Trigger Monitor program before you can run the MQLSX link extra agent.

On MQSeries for Windows NT, the command is the same:

```
runmqtnm $-m QManagerName' $-q InitiationQueueName'
```

If you are using an MQSeries client, the command is:

```
runmqtnc $-m QManagerName' $-q InitiationQueueName'
```

where QManagerName is the name of your MQSeries queue manager and InitiationQueueName is the name of your MQSeries initiation queue.

Verifying MQSeries link extra agent can initiate updates to Lotus Notes

If you have chosen to start the MQSeries link extra agent by triggering, you should verify that MQSeries link extra agent can initiate updates to Lotus Notes. Two sample demonstration Lotus Notes databases are provided. Running the sample demonstrates how separate documents in two different Lotus Notes databases can be updated by putting messages on a single queue.

1. Start Lotus Notes.

   You should have the MQSeries link extra agent work space icons representing the link database and two sample application databases. If you do not, add the icons for the three Notes databases to your work space."Start the MQSeries queue manager.

2. Use runmqsc to check to see if the following queue manager objects are defined. If not, go back and perform the MQSeries Setup procedure.

   Queues:

   ```
   NOTE           (The MQSeries link extra agent application queue*)

   MQLX.HOLD.NOTE (The user exit error message queue)

   SYSTEM.SAMPLE.NOTES.AGENT.INITQ   (The initiation queue)
   ```

   Process:

   ```
   SYSTEM.SAMPLE.NOTES.AGENT.PROCESS (Process definition to start MQSeries
                                      link extra agent process mqlinkx.exe)
   ```

   Note: * The value of the ENTRY field used to search the MQSeries link database will be equal to the queue name.

3. Start the trigger monitor by issuing the runmqtrm command.

   ```
   C:> runmqtnm -m [QMgrName] -q SYSTEM.SAMPLE.NOTES.AGENT.INITQ

    or

       runmqtnc -m [QMgrName] -q SYSTEM.SAMPLE.NOTES.AGENT.INITQ

                                 (trigger monitor for client)
   ```

   Where [QMgrName] is the name of your MQSeries queue manager and SYSTEM.SAMPLE.NOTES.AGENT.INITQ is the name of the initiation queue the trigger monitor will monitor.

4. Change to the directory where your mqlxdemo program file is stored.

5. Run the mqlxdemo program to put sample data on a queue called NOTE.

   For example:

   ```
   mqlxdemo NOTE [QMgrName] < mqlxdemo.dat
   ```

   Where [QMgrName] is the name of the queue manager. If you do not specify a value here, the system default queue manager is used.

   The application mqlxdemo reads the sample data in the mqlxdemo.dat file and writes the MQSeries messages to the NOTE queue. The NOTE queue being a triggered queue will cause the process defined by SYSTEM.SAMPLE.NOTES.AGENT.PROCESS to run the agent.

   The agent,for each message retrieved, reads the link database to determine the processing rule, and the Lotus Notes document and fields to update. It uses the queue name as the name of the ENTRY identifier in the link database.

6. Open a view in the Salary (test1.nsf) or Address (test2.nsf) database. You should now see new document entries reflecting the data processed.

   You can also run the agent program manually or as a scheduled agent (for example On Schedule Hourly) to perform these updates instead of using triggers.

Running the MQLSX link extra agent sample

Before you run the sample, make sure that you have completed all the steps described in “Introduction to the MQLSX link extra agent sample application” on page 163.

The MQLSX link extra agent sample may be started via triggering, as a scheduled agent, or run manually. The name of the queue used must match the name of the link database entry.

It is necessary for the appropriate MQSeries subsystem to be installed wherever the agent triggered task will execute. This could be either the MQSeries server or MQSeries client.

To run the MQLSX link extra agent sample manually

To run the sample manually you need:

- gmqlxtra.nsf
- mqlinkx.nsf
- test1.nsf
- test2.nsf
- mqlsxdemo
- mqlsxdemo.dat

Having checked that these are installed on your system:

1. Start MQSeries by entering the command: strmqm QMgrName
2. Modify the MQSeries agent parameters document in the agent database for your environment, specify:
   - Queue name
   - Link database name (if different from mqlinkx.nsf)
   - Queue manager name (if you are not using the default queue manager)
3. Select Agents - MQLSX Link Extra Agent
4. Select Actions - Run

Using full text indices and views

If a target database has a full text index, existing documents are located more quickly but because the index is updated by the agent each time it creates a new document, document creation takes longer.

Whether an application performs better with or without a full text index therefore depends on the characteristics of the particular application.

Note: If you are using numeric key fields to locate an existing document, the target database **must** have a full text index.

On a server, Notes periodically updates a full text index automatically. If the frequency of this update is adequate for your application, you can improve the performance of document creation in a database with a full text index, by removing the agent update statement from the LotusScript declarations event section.

This can be accomplished by commenting out the line by inserting a single quotation mark (') at the start of the line:

```
' Call oUserDb.UpdateFTIndex( False )
```

If a database is updated manually (for example by deleting or adding documents) you must ensure the full text index is deleted and recreated before running the agent.

Better performance is generally obtained if a View is specified to locate existing documents, particularly if the database has no full text index.

Using a Notes view to search for a document

If you have a large number of documents in the database that are to be updated with information from your host system, you are recommended to use a view to search for documents. However, you must adhere to the conditions attached to using views.

- A view must have at least one column sorted in ascending order.
- The first column of the view must contain the data for the first key field specified for that entry.

How the MQLSX link extra agent sample works

The MQLSX link extra agent sample demonstrates how you can use the MQSeries Trigger Monitor for Lotus Notes agents to trigger a Lotus Notes Agent called "MQSeries link extra agent" that is contained in the Notes database, gmqlxtra.nsf.

The MQSeries link extra agent makes use of the MQLSX. You should have the appropriate MQSeries subsystem installed wherever the agent will run.

When triggering is required, an application queue with triggering turned on is needed with an associated process definition. The triggered queue has a defined process that starts the MQSeries link extra agent. The MQSeries link extra agent processes messages placed on an input queue and consults the Lotus Notes link database for the processing rule and the Notes document and fields to update. The MQSeries link extra agent program may call a customized LotusScript user exit.

The agent gets an MQSeries message from the specified queue and, for each link entry rule document with an Entry field matching the queue name, applies the update defined by the rule to the user database specified by that rule document.

Note: A single message may match more than one rule document and a single rule document may select more than one user document for update.

Using the MQSeries link extra agent sample

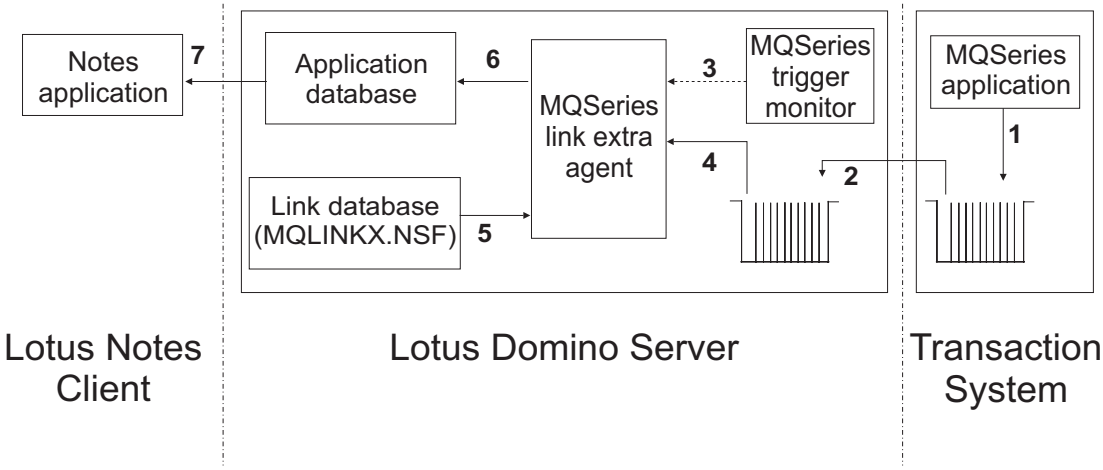

Figure 2. Using the MQSeries link extra agent sample

1. An MQSeries application running on your enterprise system generates a message containing the data it wants to pass to the Notes environment, and puts it on a queue, defined as a transmission queue.
2. MQSeries, running on the Domino server extracts the message and puts it on a local application queue. This local queue has been defined as a triggered queue.
3. The trigger monitor program recognizes a message has arrived on the queue, carries out the normal triggering checks, and if all the criteria are met, issues a call to run the Notes agent.
4. The MQSeries link extra agent gets the message from the application queue.
5. It reads the document in the link database, the queue name as the key (to match against the Entry field), for each message and determines the processing rules.
6. The appropriate Notes document is updated or created, as determined by the processing rules.
7. The Lotus client application opens the database and views the data. There is no dynamic display of any changed data, as it is the enterprise application initiating the change.

Link extra database contents (mqlinkx.nsf)

The link extra database contains the information needed by the Notes agent to determine what processing it needs to do for an incoming message:

- Entry
  The name of the MQSeries queue from which the Notes agent gets messages.
- Description
  Text that is useful to you. For example, you may want include the type of information held in the messages, or the name of the transmission queues that supply the messages.
- Database Information

User Database Name: The name of the Notes database to be updated with the data in the MQSeries message.
View to Use for Searching: The name of a Notes view known to your Notes application that identifies the document to be updated. This must be shared view.

- Rule
  There are a set of predetermined rules from which you must select one. The rule you select (there is no default rule) determines how the Notes agent processes the MQSeries message, based on documents that already exist in the Notes database.

  In summary:

  Insert:
  A new document is created

  Update:
  The located document is updated

  Ignore:
  No action is taken

  Fail:
  A call is made to the user exit

  The rules are:

Table 4. Link extra agent contents rules

Rule no.	Condition	Action taken
0	If found insert, else fail	When no documents are found, call the user exit.
1	If found update, else fail	When no documents are found, call the user exit.
2	If found fail, else insert	When one or more documents are found, call the user exit.
3	If found update, else insert	When a matching document is not found, a new one is created, the user exit is not called.
4	If found update, else ignore	When no matching document is found, the message is ignored or discarded and the user exit is not called.
5	Insert always	A new document is always created regardless of whether a matching document is found, the user exit is not called.
6	If found ignore, else fail	When no matching documents are found, call the user exit.
7	If found insert, else ignore	When no matching documents are found, the message is ignored or discarded and the user exit is not called.
8	If found fail, else ignore	When one or more matching documents are found, call the user exit.
9	If found ignore, else insert	When a matching document is found, the message is ignored or discarded and the user exit is not called.

- Output Offsets
  Describes the layout of the data in the MQSeries message:
  Syntax:
  FieldName <space> Start<space> End<space>CHAR | NUM |
  INTEL-BINARY<space>KEY<comparison>

Table 5. Output offset definitions

Syntax	Meaning
Fieldname	The name of the field in the Notes form to be updated with data in this part of the MQSeries message. A Fieldname value of "FORM" can be used to specify the form name of the target document.
Start	The position in the MQSeries message at which the data in this field starts.
End	The position in the MQSeries message at which the data in this field ends.
CHAR | NUM | INTEL-BINARY	The datatype of the field. Use CHAR for character data, and NUM for (Intel or S/390 format) numeric data. Note that for compatibility, INTEL-BINARY can also be specified but is treated exactly the same as NUM.
KEY	Optional. Use the word KEY against a field to indicate that this field is of the selection criteria when searching for a document in a Notes database. Caution If you specify numeric KEY fields in the link database, you MUST create a full text index for the target database.
<comparison>	Optional and only used with a KEY field. A token that you set to <, >, +, =, CASE or NOCASE. When selected, is used to compare the field in the Notes document with the field in the MQSeries message. The default value is NOCASE. **Note:** Use the word CASE to indicate that the search must be case sensitive. Use the word NOCASE to indicate that the search is not case sensitive.

- Additional Selection Formula
  Optional. Used to specify search criteria when searching for documents in a Notes database. All conditions must conform to the LotusScript FTSearch verb.

Note: When KEY fields are not specified and the Additional Selection Formula is blank, all documents in the database are selected.

Error handling and status reporting in the MQLSX link extra agent sample

Messages are output by the sample both to report errors and provide status information. All messages are output by the trigger monitor to the Notes status bar or server console.

Status messages

The status messages that you may see in the Notes status bar when running the MQLSX link extra sample are:

Table 6. Status messages

Message	What it means...
Starting MQLSX link extra agent Version n.n	The Agent program is starting.
Using default parameters	The Agent could not find an agent parameter document is using the default parameters.
Using default queue manager	The Agent is using the default MQSeries queue manager.
Using default queue name	The Agent is using the default queue name.
Using default link database	The Agent is using the default link database.
Using parameters from agent parameter document	The agent could not find a trigger message document and is using the agent parameter document.
Using parameters from trigger message document	The Agent is using parameters from the trigger message document.
User exit invoked	The user exit was invoked.
End of messages	
Agent ended abnormally	
Agent ended normally	

Error Messages

Error messages output by the sample are detected by LotusScript, the MQLSX, or the agent. The format in each case is different.

Errors detected by Lotus Notes

The format of these messages output by the link extra agent sample is:

MQLSX link extra Agent Notes Error: Agent_Insert_String Notes_Error_Message (Notes error = Notes_Error_Number Line nnnn)

where:

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank

Notes_Error_Message
is the LotusScript error message

nnnn
is the line number in the LotusScript source code where the error occurred

Errors detected by the MQLSX

The format of these messages output by the link extra agent sample is:

MQLSX link extra agent Error: Agent_Error_Number Agent_Error_Message Agent_Insert_String Notes_Error_Message (Notes error = Notes_Error_Number Line nnnn)

where:

Agent_Error_Number
is an agent error number that may be omitted for MQLSX detected errors if zero

Agent_Error_Message
is an agent error message

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank.

MQLSX_Error_Message
is the LotusScript MQLSX error message and will typically provide the MQSeries or MQLSX reason code

Notes_Error_Message
is the LotusScript error message

nnnn
is the line number in the LotusScript source code where the error occurred

Errors detected by the link extra agent sample

The format of these messages output by the link extra agent sample is:

MQLSX link extra agent Error: Agent_Error_Number Agent_Error_Message Agent_Insert_String

where:

Agent_Error_Number
is an agent error number that may be omitted for MQLSX detected errors if zero

Agent_Error_Message
is an agent error message

Agent_Insert_String
is any additional useful information about the error that the agent can provide and may be omitted if blank.

The error messages you can encounter that are detected by the Agent component of the Link sample are:

Table 7 (Page 1 of 3). Link extra agent sample error definitions

Message	What it means...	Action
Link database could not be opened. (66001)	An error occurred opening the Link database.	Check that the Link database is available and its specification is correct.
User database could not be opened. (66002)	An error occurred opening the target user database.	Check that the target user database is available and its specification is correct.
Link Entry Error: start or end position value not valid. (66003)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. The start offset must be greater than zero and the end offset must be equal to or greater than the start offset.
Link Entry Error: one or more field items are missing. (66004)	The error was detected in the Link database field definition shown.	Correct the missing value (or values) and retry the operation.
Link Entry Error: data type not valid. (66005)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. Valid data types are CHAR, NUM, (and INTEL_BINARY).
Link Entry Error: data length not valid for NUM. (66006)	The error was detected in the Link database field definition shown.	Valid data lengths for NUM are 2, 4 and 8. Correct the incorrect value and retry the operation.
Link Entry Error: KEY position value not valid. (66007)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. Values other than "KEY" are not valid in the 'key' position.
Link Entry Error: comparison position value not valid. (66008)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. Values other than "CASE" or "NOCASE" are not valid in the 'comparison' position.
Link Entry Error: form specification not valid. (66009)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. 'Key' and 'comparison' keywords and datatypes other than CHAR cannot be specified for a field used to specify the form name (that is, with field name 'FORM').
Link Entry Error: field length value too big. (66010)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. Field lengths greater than 32000 are not supported.

Table 7 (Page 2 of 3). Link extra agent sample error definitions

Message	What it means...	Action
Link Entry Error: key field length value is too big. (66011)	The error was detected in the Link database field definition shown.	Correct the incorrect value (or values) and retry the operation. Key field lengths greater than 64 are not supported.
Error saving Notes document. (66012)	An unexpected error occurred when attempting to save a document in the target Notes database.	Check that your target Notes databases can be successfully accessed and check that there is sufficient system resources (for example, disk space).
A number in an MQSeries message had an invalid exponent. (66013)	The exponent value of an 8-byte NUM (double precision) number is not valid. Refer the problem to the provider of your server application.	
A rule document with the specified name was not found. (66014)		Check that a rule document exists for application queue name shown.
The view specified in the Link Entry rule document was not found. (66015).		Check that a view exists in the target database for the view name shown.
A Invalid rule was detected. (66016)		Replace the rule shown in the Link entry document with a supported value.
Backout threshold of failing message exceeded; message could not be requeued and was discarded. (66017)	A failing message was discarded because, after backing out and retrying the required number of times, it could not be successfully requeued on either the backout requeue queue or the dead letter queue.	Refer to the previous error messages displayed whilst running the link extra agent sample to determine the cause of the problem. Create a backout requeue queue or a dead-letter queue if required.
Backout threshold of failing message exceeded; message was requeued to: (66018)	A failing message was requeued to the queue indicated because, after backing out and retrying the required number of times, the message could not be processed successfully.	Refer to the previous error messages displayed whilst running the link extra agent sample to determine the cause of the problem and retry the operation if required. Note If the message was requeued to the dead letter queue the message data will be prefixed by a dead letter header.

Table 7 (Page 3 of 3). Link extra agent sample error definitions		
Message	**What it means...**	**Action**
User exit returned "Stop and exit". (66019)	A user exit call specified by the processing Rule returned "Stop and Exit". Processing is ended.	Investigate why the user exit was invoked and returned the completion code shown if this was not the intended action.
Numeric key fields are not valid with a target database that is not full text indexed. (66020)		Create a full text index in the target database or change the key field to use type CHAR.
The link database view shown was not found. (66021)	The view required to access information in the link database was not found.	Create the required view in the link database.

Agent_Insert_String

The Agent_Insert_String may be blank or one of the following:

```
MQLSX error getting message from application queue
```

Consult your *MQSeries Application Programming Reference* or “Reason codes” on page 41.

```
MQLSX message DataOffset error.
```

Check that your field definition values in the link database are correct for the incoming messages. See the *MQSeries Application Programming Reference* or “Reason codes” on page 41.

```
MQLSX message ReadString error.
```

Check that your field definition values in the link database are correct for the incoming messages. See the *MQSeries Application Programming Reference* or “Reason codes” on page 41.

```
MQLSX message ReadShort error.
```

An MQLSX error occurred when attempting to read a short integer from an MQMessage. See the *MQSeries Application Programming Reference* or “Reason codes” on page 41.

`MQLSX message ReadLong error.`

An MQLSX error occurred when attempting to read a long integer from an MQMessage. Check that your field offset values in the link database are correct for the incoming messages. See the *MQSeries Application Programming Reference* or “Reason codes” on page 41.

`Error accessing queue manager`

An MQLSX error occurred when attempting to access the queue manager. Check your target queue name is correct and the queue manager is started. See the *MQSeries Application Programming Reference* or “Reason codes” on page 41.

Appendix C. The MQLSX Distribution List sample application

This appendix describes the MQLSX Distribution List sample application. This sample is designed to illustrate in a graphical way how MQSeries Distribution Lists work in a Lotus Notes database using LotusScript.

The sample consists of a single Notes database called gmqldist.nsf. To add the Distribution List sample application to the Notes desktop Select File / Database / Open and then select 'Distribution List Sample' from the list (or if using the browse option select the file called gmqldist.nsf) and press the 'Add Icon' button. The sample database icon will appear on your Lotus Notes desktop.

Note: The following information may also be found in the Distribution List sample database from the 'About this database' help menu option.

Before running the sample

- Create and start a default queue manager (can be any name). If you already have one make sure it is the default and that it is running (strmqm -i QueueManageName).
- Create 5 new queues called Queue1, Queue2, Queue3, Queue4, & Queue5. (use runmqsc.exe and type define qlocal('Queue1') etc).

Note: The sample will NOT work without these queues defined.

Running the sample

Open (or select) the database, then from the Create menu select 'Distribution List Sample'. This will run the sample (make sure the default queue manager is running, with the appropriate queues defined as described above).

A form will appear that enables you to explore the operation of Distribution Lists (if an error is encountered at this point, a message box will appear reporting the problem with the appropriate MQLSX reason code). Note: the most common error encountered usually occurs when the default queue manager has not been started or the necessary queues have not been defined.

When the samples form first appears, you will be presented with an empty Distribution List (that is, just the table headings) and a series of buttons.

These buttons let you explore the operation of the Distribution List:

- You can add new Distribution List Items to the Distribution List (maximum 5 entries) using the 'Add' button.
- You can traverse the list with the 'Next' and 'Previous' buttons.
- The Distribution List can be opened and closed using the 'Open' and 'Close' buttons.

- An entry field lets you type in a text message that will be stored in an MQMessage and put to the Distribution List (when the 'Put' button is pressed).
- When the 'Put' button is pressed, the MQ Message will be put on all the queues associated with the items in the Distribution List.
- The 'Get' button can then be used to read the MQMessage back from the queues and display the text received in the message in the table on the form.

About the sample code

To view the design and underlying code of the Distribution List sample, open the database click on 'Design' in the navigator panel followed by 'Forms'. The words 'Distribution List Sample' will appear highlighted to the right of the navigator panel. Double click on the highlighted text to open the design form. You will be presented with a split screen showing the form design on the top half and the code window on the bottom half. Follow the instructions in the code window to guide you through the code contained in the sample.

Note: See *MQSeries Application Programming Guide* for details on MQSeries Distribution Lists.

Appendix D. Notices

This information was developed for products and services offered in the United Kingdom. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM documentation or non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those documents or Web sites. The materials for those documents or Web sites are not part of the materials for this IBM product and use of those documents or Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
IBM	IMS	MQSeries
OS/2	OS/390	VSE/ESA

Lotus, and LotusScript are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, or service names, may be the trademarks or service marks of others.

Glossary of terms and abbreviations

This glossary defines MQSeries terms and abbreviations used in this book. If you do not find the term you are looking for, see the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

administrator commands. MQSeries commands used to manage MQSeries objects, such as queues, processes, and namelists.

alias queue object. An MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

application queue. A queue used by an application.

asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

attribute. One of a set of properties that defines the characteristics of an MQSeries object.

C

CDF. Channel definition file.

channel. See *message channel*.

channel definition file (CDF). In MQSeries, a file containing communication channel definitions that associate transmission queues with communication links.

channel event. An event indicating that a channel instance has become available or unavailable. Channel events are generated on the queue managers at both ends of the channel.

client. A run-time component that provides access to queuing services on a server for local user applications. The queues used by the applications reside on the server. See also *MQSeries client*.

client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

client connection channel type. The type of MQI channel definition associated with an MQSeries client. See also *server connection channel type*.

command. In MQSeries, an administration instruction that can be carried out by the queue manager.

completion code. A return code indicating how an MQI call has ended.

connect. To provide a queue manager connection handle, which an application uses on subsequent MQI calls. The connection is made either by the MQCONN call, or automatically by the MQOPEN call.

connection handle. The identifier or token by which a program accesses the queue manager to which it is connected.

context. Information about the origin of a message.

context security. In MQSeries, a method of allowing security to be handled such that messages are obliged to carry details of their origins in the message descriptor.

D

dead-letter queue (DLQ). A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

dead-letter queue handler. An MQSeries-supplied utility that monitors a dead-letter queue (DLQ) and processes messages on the queue in accordance with a user-written rules table.

DLQ. Dead-letter queue.

E

event. See *channel event*, *instrumentation event*, *performance event*, and *queue manager event*.

event data. In an event message, the part of the message data that contains information about the event (such as the queue manager name, and the application that gave rise to the event). See also *event header*.

event header. In an event message, the part of the message data that identifies the event type of the reason code for the event.

event message. Contains information (such as the category of event, the name of the application that caused the event, and queue manager statistics) relating to the origin of an instrumentation event in a network of MQSeries systems.

event queue. The queue onto which the queue manager puts an event message after it detects an event. Each category of event (queue manager, performance, or channel event) has its own event queue.

Event Viewer. A tool provided by Windows NT to examine and manage log files.

F

FFST. First Failure Support Technology.

First Failure Support Technology (FFST). Used by MQSeries on UNIX systems, MQSeries for OS/2 Warp, MQSeries for Windows NT, and MQSeries for AS/400 to detect and report software problems.

G

get. In message queuing, to use the MQGET call to remove a message from a queue.

H

handle. See *connection handle* and *object handle*.

hardened message. A message that is written to auxiliary (disk) storage so that the message will not be lost in the event of a system failure. See also *persistent message*.

I

instrumentation event. A facility that can be used to monitor the operation of queue managers in a network of MQSeries systems. MQSeries provides instrumentation events for monitoring queue manager resource definitions, performance conditions, and channel conditions. Instrumentation events can be used by a user-written reporting mechanism in an administration application that displays the events to a system operator. They also allow applications acting as agents for other administration networks to monitor reports and create the appropriate alerts.

L

local definition of a remote queue. An MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

local queue. A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager. The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called *remote queue managers*, even if they are running on the same system as the program.

log. In MQSeries, a file recording the work done by queue managers while they receive, transmit, and deliver messages, to enable them to recover in the event of failure.

log file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file in which all significant changes to the data controlled by a queue manager are recorded. If the primary log files become full, MQSeries allocates secondary log files.

M

message. (1) In message queuing applications, a communication sent between programs. See also *persistent message* and *nonpersistent message*. (2) In system programming, information intended for the terminal operator or system administrator.

message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a receiver at the other end) and a communication link. Contrast with *MQI channel*.

message descriptor. Control information describing the message format and presentation that is carried as part of an MQSeries message. The format of the message descriptor is defined by the MQMD structure.

message queue. Synonym for *queue*.

message queue interface (MQI). The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

messaging. See *synchronous messaging* and *asynchronous messaging*.

MQAI. MQSeries Administration Interface.

MQI. Message queue interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system, and transfers only MQI calls and responses in a bidirectional manner. Contrast with *message channel*.

MQSC. MQSeries commands.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries Administration Interface (MQAI). A programming interface to MQSeries.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

N

namelist. An MQSeries object that contains a list of names, for example, queue names.

nonpersistent message. A message that does not survive a restart of the queue manager. Contrast with *persistent message*.

O

OAM. Object authority manager.

object. In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

object authority manager (OAM). In MQSeries on UNIX systems and MQSeries for Windows NT, the default authorization service for command and object management. The OAM can be replaced by, or run in combination with, a customer-supplied security service.

object descriptor. A data structure that identifies a particular MQSeries object. Included in the descriptor are the name of the object and the object type.

object handle. The identifier or token by which a program accesses the MQSeries object with which it is working.

P

PCF. Programmable command format.

PCF command. See *programmable command format*.

performance event. A category of event indicating that a limit condition has occurred.

persistent message. A message that survives a restart of the queue manager. Contrast with *nonpersistent message*.

ping. In distributed queuing, a diagnostic aid that uses the exchange of a test message to confirm that a message channel or a TCP/IP connection is functioning.

programmable command format (PCF). A type of MQSeries message used by:

- User administration applications, to put PCF commands onto the system command input queue of a specified queue manager
- User administration applications, to get the results of a PCF command from a specified queue manager
- A queue manager, as a notification that an event has occurred

Contrast with *MQSC*.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages—they point to other queues, or can be used as models for dynamic queues.

queue manager. (1) A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. (2) An MQSeries object that defines the attributes of a particular queue manager.

queue manager event. An event that indicates:

- An error condition has occurred in relation to the resources used by a queue manager. For example, a queue is unavailable.
- A significant change has occurred in the queue manager. For example, a queue manager has stopped or started.

queuing. See *message queuing*.

R

reason code. A return code that describes the reason for the failure or partial success of an MQI call.

receiver channel. In message queuing, a channel that responds to a sender channel, takes messages from a communication link, and puts them on a local queue.

Registry. In Windows NT, a secure database that provides a single source for system and application configuration data.

Registry Editor. In Windows NT, the program item that allows the user to edit the Registry.

Registry Hive. In Windows NT, the structure of the data stored in the Registry.

remote queue. A queue belonging to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. To a program, a queue manager that is not the one to which the program is connected.

remote queue object. See *local definition of a remote queue*.

remote queuing. In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

reply message. A type of message used for replies to request messages. Contrast with *request message* and *report message*.

reply-to queue. The name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

report message. A type of message that gives information about another message. A report message can indicate that a message has been delivered, has arrived at its destination, has expired, or could not be processed for some reason. Contrast with *reply message* and *request message*.

requester channel. In message queuing, a channel that may be started remotely by a sender channel. The requester channel accepts messages from the sender channel over a communication link and puts the messages on the local queue designated in the message. See also *server channel*.

request message. A type of message used to request a reply from another program. Contrast with *reply message* and *report message*.

return codes. The collective name for completion codes and reason codes.

S

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With

synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

T

time-independent messaging. See *asynchronous messaging*.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF).

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

U

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Sending your comments to IBM

MQSeries®

LotusScript® Extension

SC34-5404-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, use the Readers' Comment Form
- By fax:
  - From outside the U.K., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries®

LotusScript® Extension

SC34-5404-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Telephone	Email

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States
Belgium	France	Israel	New Zealand	Spain	of America
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____

Company or Organization _____

Address _____

_____

EMAIL _____

Telephone _____

1 Cut along this line

4 Fasten here with adhesive tape

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5404-00