44

MQ update

February 2003

In this issue

© Xephon plc 2003

MQ Update

Published by
Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

North American office
Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290

Disclaimer
Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Editor
Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues
A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.75) each including postage.

Contributions
When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

MQ Update on-line
Code from *MQ Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mq; you will need to supply a word from the printed issue.

© Xephon plc 2003. All rights reserved. None of the text in this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior permission of the copyright owner. Subscribers are free to copy any code reproduced in this publication for use in their own installations, but may not sell such code or incorporate it in any commercial product. No part of this publication may be used for any form of advertising, sales promotion, or publicity without the written permission of the publisher. Copying permits are available from Xephon in the form of pressure-sensitive labels, for application to individual copies. A pack of 240 labels costs $36 (£24), giving a cost per copy of 15 cents (10 pence). To order, contact Xephon at any of the addresses above.

Printed in England.

WMQ for z/OS V5.3: message expiry enhancement

INTRODUCTION

The announcement letter for WebSphere MQ (WMQ) for z/OS V5.3 at *http://www.ibmlink.ibm.com/usalets&parms=H_202-075* states under the heading *Queue manager performance, availability, and usability*, "More timely and efficient message expiry – currently, obsolete messages are only expired when they are encountered by an MQGET or MQGET browse. Obsolete messages can remain on a queue for a long time, especially if the queue is accessed by Msg-ID or Correl-ID. This item runs a periodic background task to scan all queues for expired messages".

This article explains this new "more timely and efficient message expiry support".

Let's look at the problem that required this solution and at some of the previous methods of dealing with it.

Consider an application that uses MQPUT to put a message on a remote queue and then uses MQGET with a wait interval for the message to arrive back on the reply-to queue. If the message does not arrive back within this interval the application may perform some other work. The reply message was created with an expiry interval on it and the application assumes the message will eventually be deleted.

However, if the message is not the target of a MQGET browse request (first introduced with MQSeries for OS/390 V5.2) or a destructive MQGET, it will remain on the reply-to queue forever.

The new facility can be used to delete these expired messages.

This problem has existed for a long time and several methods have previously been used to solve it. Let's look at some of them.

RIDDING QUEUES OF EXPIRED MESSAGES: PREVIOUS METHODS

I'm assuming you are familiar with this subject so I will not go into the full details of these methods.

Perform a destructive MQGET and specify a non-existent combination of Msg-ID and Correl-ID

Advantages

- There is only one call across the application/queue manager boundary.

Disadvantages

- It trashes the buffer pool.
- It doesn't work on indexed queues.
- The queue may not contain any expired messages.
- It has to be run on every applicable queue.
- Such a message might exist and hence be deleted!

Browse a queue using WMQ for OS/390 V5.2

Advantages

- This method would never remove any unexpired message from the queue.

Disadvantages

- It trashes the buffer pool.
- The queue might not contain any expired messages.
- It has to be run on every applicable queue.
- It needs to pass across the application/queue manager boundary for each message on the queue.

Both these schemes require workload applications to be written and maintained, distracting you from your principal business objectives. Now this work can be left to the queue manager itself or you can replace your own workload with a new queue manager command.

WHAT IS THE ‘NEW METHOD’?

This new expiry ‘scanning’ is an internal queue manager process, which knows the queues that contain messages that will expire and accesses the queues without the use of the MQGET API. It will cause message expiry reports to be created if required.

Advantages

- It does not involve repeated calls across the application/ queue manager boundary and so has a much reduced path length when compared to an MQGET request.
- It uses only one buffer for private queues and, therefore, does not trash the buffer pool.
- It is only performed when it believes that there is a worthwhile number of expired messages to be deleted.
- It will not prevent access to unexpired messages by your applications when it is running.
- It is run on every applicable queue.
- It is significantly more efficient than all previous methods.

WHAT NEW TOOLS ARE PROVIDED?

A new attribute, EXPRYINT, has been added to the QMGR object so that expiry scans can be performed automatically at your preferred intervals.

A new **REFRESH QMGR** command has been added so that you can perform an expiry scan when you choose.

Queue manager EXPRYINT keyword

The queue manager object has a new EXPRYINT attribute. The value will initially be OFF, as can be seen with a **DIS QMGR EXPRYINT** command after the first restart of your queue manager once V5.3 is installed. This value can also be determined with the MQINQ call, using a selector of MQIA_EXPIRY_INTERVAL and the value will be returned in seconds or as the constant MQEXPI_OFF.

You can change this attribute with an **ALTER MGR EXPRYINT(value)** command, where the value can be OFF or a number of seconds between one and 99,999,999.

How is this queue manager EXPRYINT keyword used?

When this value is not OFF all private queues (actually just their in-storage control blocks – no messages are read at this stage) are examined every time this interval ends. Those private queues that contain expired messages are considered for an internal scan. Private queues will be scanned when there is a reasonable chance of deleting 'many' expired messages. It would just be a waste of CPU time if there were only a chance of deleting one or two expired messages.

An internal queue manager task will perform the scan of the queue using a very efficient process. This does not interfere with the responsiveness of your applications when they are concurrently adding or removing messages to or from the queue being scanned, regardless of the depth of the queue.

For shared queues only one queue manager in a queue-sharing group will take control of the scanning work, and only that queue manager will run the queue-scanning task to search for and delete the expired messages. If this queue manager is stopped or its EXPRYINT value is set to OFF another queue manager in the queue-sharing group will take responsibility for running the expiry scans on all the shared queues. It therefore makes sense that all the queue managers in a queue-sharing group should have the same value in the QMGR object's EXPRYINT attribute, since you don't know which one will actually perform this work.

You can only observe that this new process is at work by seeing that the CURDEPTH of a queue has decreased. For shared queues you may see (unexpected) connections to structures in the Coupling Facility, caused by an expiry scan.

Note that this interval, when used, applies to all queues in your queue manager so you cannot use it to have some queues scanned hourly and others scanned every six hours.

If you require this type of support you can instead use the new **Refresh QMGR** command described below.

Naturally the pre-existing function is unchanged, so an expired message will continue to be deleted when it is the target of a destructive MQGET request or it is browsed using MQGET (first introduced in MQSeries for OS/390 V5.2).

What happens when I change the EXPRYINT value?

Changes to the EXPRYINT value take effect when the current interval expires unless the new interval is less than the unexpired portion of the current interval, in which case a scan will be scheduled immediately and the new interval value will take effect. For example, assume the EXPRYINT value was 86400 (86,400 seconds = 24 hours) and that it is changed to 7200 (2 hours):

- If the last scan was three hours ago the unexpired portion is 21 hours. Since two hours is less than 21 hours a scan is run now, and then every two hours.
- If the last scan was 23 hours ago the unexpired portion is one hour. Since two hours is not less than one hour a scan will take place in an hour, and future scans will then take place every two hours thereafter.

The REFRESH QMGR command

You can use the new **REFRESH QMGR** command to perform manually an expiry scan on a queue at a time that suits you.

This command can be used even if EXPRYINT is set to OFF.

You can use this command instead of or in addition to the QMGR EXPRYINT attribute, or as a direct replacement for any workload that you currently perform to achieve the same purpose.

The format of the new command is:

```
   REFRESH QMGR TYPE(EXPIRY) NAME(q-names*) CMDSCOPE()
```

Standard queue name matching is used if you specify a trailing asterisk (*) at the end of the NAME field.

It does not make sense to run the command concurrently in more than one queue manager in a queue sharing group. This applies to both private queues and shared queues, since the queue-scanning work need only be done by one task, so do not specify **CMDSCOPE** and the command will be run on the queue manager on which it was entered.

I said earlier that when you use EXPRYINT only one queue manager in a queue-sharing group takes control of the scanning work that will delete the expired messages; this new command can be run against shared queues from any queue manager in a queue sharing group.

Examples of the new command

- Example one: issue the command against a single queue:

  ```
  cpf REFRESH QMGR TYPE(EXPIRY) NAME(GEOFF.APPL)
  ```

  You will see on the console:

  ```
  CSQM173I cpf CSQMRMMS EXPIRED MESSAGE SCAN REQUESTED FOR 1 QUEUES
  CSQ9Ø22I cpf CSQMRMMS ' REFRESH QMGR' NORMAL COMPLETION
  ```

  The CSQ9022I message tells you the command has completed and *not* that the expiry scan itself is complete. The scan of the queue will then start and no further console messages are issued. You may display the CURDEPTH of the queue before and after the scan to see how many messages, if any, have been deleted.

- Example two: issue the command against a set of queues:

  ```
  cpf REFRESH QMGR TYPE(EXPIRY) NAME(GEOFF.*)
  ```

  You will see on the console:

  ```
  CSQM173I cpf CSQMRMMS EXPIRED MESSAGE SCAN REQUESTED FOR xx QUEUES
  CSQ9Ø22I cpf CSQMRMMS ' REFRESH QMGR' NORMAL COMPLETION
  ```

Note that, since the **CMDSCOPE** keyword was not used, the command runs on the system where it was entered.

SCANNING QUEUES AT SPECIFIC TIMES OF THE DAY/WEEK

After the first restart of your queue manager running V5.3, EXPRYINT will be set to OFF.

If you want an expiry scan to start daily at 8pm, say, wait until 8pm and then issue the command **ALTER QMGR EXPRYINT(86400)** on the relevant queue managers. Thereafter, expiry scans will start at 8pm every day without any further intervention.

If you have to stop and restart your queue manager, which most likely will not happen at 8pm:

- Ensure the command **ALTER QMGR EXPRYINT(OFF)** is in a data set in the CSQINP2 concatenation to prevent any queues being scanned before your 8pm target time.
- At 8pm issue the command **ALTER QMGR EXPRYINT(86400)**.

This procedure could also be managed by a job automation system. Your job automation system can also be used to submit the new command to be run against the selected subsets of your queues.

SUMMARY

Queue manager-initiated expiry processing solves a long-standing problem. Try it.

Geoffrey Belding
Senior Software Engineer
WMQ for z/OS Development
IBM Hursley (UK)

© IBM 2003

JavaMQMail: MQSeries as an e-mail infrastructure

We had a great application that used MQSeries as a key infrastructure tool that collected messages from various platforms and e-mailed them to the outside world. This application was written as a Lotus Notes agent. Our company is de-emphasizing Notes, so I converted the application to Java.

The old Notes agent, of course, ran on the Notes server. The new JavaMQMail application happens to reside on our queue manager server, but it could be on any LAN-connected server in your environment.

JavaMQMail picks up messages sent to it via MQSeries and uses the JavaMail classes to send out the resulting e-mails. The original requirement was to enable e-mails for programs running in CICS on our OS/390 mainframe, but JavaMQMail enables all platforms in our company to participate in a common, stable e-mail utility.

JavaMQMail also uses Sun's Java Network Directory Interface (JNDI) to issue LDAP calls that look up e-mail addresses on our Exchange server.

MQSeries, Java, and LDAP: quite a mix, but together they make for a powerful tool. This article also demonstrates a debugging technique that can be turned on and off. It also describes a queue browser utility that improves on *AMQSBCGC*. All of the Java programs are commented heavily so that you can customize them to suit your own installation.

INSTALLATION

Download the JNDI classes from Sun at *http://java.sun.com/jndi*.

To use JavaMail you also need the JavaBeans Activation Framework (JAF) classes. Download these from Sun at:

- *http://java.sun.com/products/javamail/index.html* for the JavaMail API.

© 2003. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

- *http://java.sun.com/products/javabeans/glasgow/jaf.html* for the JavaBeans Activation Framework.

Sun provides an excellent tutorial that shows you how to use JavaMail and explains how to download and install the components. I highly recommend you review this at *http://developer.java.sun.com/developer/onlineTraining/JavaMail*.

In the process of installing the JavaMail, JAF, and JNDI classes, you will need to add the following to your CLASSPATH:

- *activation.jar*.
- *mail.jar*.
- *jndi.jar*.
- *ldap.jar*.
- *providerutil.jar*.
- *ldapbp.jar*.

Then create an MQ queue to handle the messages themselves. In this example I called it *JAVA.MAIL.QUEUE*.

Notes provided a useful logging function but Notes is going away, so I replaced the Notes log with another MQ queue – *JAVA.MAIL.LOG.QUEUE*.

Don't forget to set up the processes on the test and production MQ servers. In our environment we have queue managers running on the mainframe and on one of the Windows NT servers. The relevant code is listed below.

SETTING UP THE PROCESSES

```
* Java-to-Mail Queue
* This side is for the mainframe.
* This also needs to be defined as a REMOTE queue on NTServer.
DEFINE QLOCAL('JAVA.MAIL.QUEUE') +
REPLACE +
LIKE('SYSTEM.DEFAULT.LOCAL.QUEUE') +
DESCR('Java-to-Mail Queue') +
INITQ('SYSTEM.DEFAULT.INITIATION.QUEUE') +
PROCESS('JAVA.MAIL.PROCESS') +
TRIGGER +
TRIGTYPE(FIRST)
DEFINE PROCESS('JAVA.MAIL.PROCESS') +
REPLACE +
DESCR('Process Definition for Java-to-Mail Queue')
+
* appltype 11 is WindowsNT
APPLTYPE(11) +
APPLICID('D:\MQM\JavaMQMail.CMD')
* Java-to-Mail Logging Queue
DEFINE QLOCAL( 'JAVA.MAIL.LOG.QUEUE' ) +
REPLACE +
* Common queue attributes
DESCR( 'Java-to-Mail Logging Queue' ) +
* Trigger attributes
NOTRIGGER
*
```

The *D:\MQM\JavaMQMail.CMD* is a local file on the NT server and contains the following single line:

```
   start c:\jdk1.3.1\bin\java.exe JavaMQMail NTQueueManagerName
```

This points out that the JDK needs to be installed locally on the Windows queue manager server. The target directory – where you move the classes to be run – is also on the NT server in *D:\MQM*. This location is not critical but you do need to know it.

To use the MQJava classes you will need the MA88 SupportPac from IBM. This is obtained from *http://www-3.ibm.com/software/ts/mqseries/txppacs/ma88.html*.

MA88 provides the MQ-to-Java classes and has its own quirks for installation, classpath, etc. Information on the MQJava classes has appeared previously in these pages. See Appendix A for a short summary of this information.

AMQSBCGC is a fine utility program supplied with MQSeries, which browses queues. However, it cannot handle the log messages produced by JavaMQMail in the *JAVA.MAIL.LOG.QUEUE*. I wrote a crude but effective replacement for *AMQSBCGC* and the source code is included here as *ReadJavaLog*. You invoke this class by hand by typing either:

```
   java ReadJavaLog QueueManagerName JAVA.MAIL.LOG.QUEUE
```

or:

```
   java ReadJavaLog QueueManagerName QueueName
```

ReadJavaLog would work very nicely with a front-end GUI. However, the GUI has not yet been implemented.

Mail will travel to all valid Internet addresses through the SMTP gateway server you specify. Failures (non-delivery reports) will go back to the sender from the Postmaster. That means that all sender and recipient addresses must be valid Internet-style addresses. In the Notes version of the agent the ‘From’ field was mostly decorative although the ‘To’ field had to match something in the Notes Address Book. Without the Notes server we don’t have the Notes address book so the e-mail addresses must stand on their own.

A valid Internet-style address includes an ‘@’ and a period (.) but no blanks. When the address provided is not a valid Internet-style e-mail address JavaMQMail will take whatever was sent and issue an LDAP lookup into a nearby Exchange directory. If Exchange recognizes the name it will return the correct SMTP e-mail address. JavaMQMail sends the address it does have to Exchange as a ‘UID’. UID in Exchange is the Alias and the ‘mail’ attribute is returned if anything is found. If Exchange finds something, great. Use the new address from there on. But if nothing is in Exchange that matches, JavaMQMail will attempt to use whatever the originally-supplied address was.

JavaMQMail contains lots of notifications back to the original senders of the messages. If bad addresses are sent down, the senders will soon find out about it.

Finally, I created *JavaMQMail.INI* and altered the program to read this INI file to obtain some parameters at start-up time. These are meant to temporarily override settings in the QueueManagerDEFS or in the *JavaMQMail.CMD* file. Parameters included are:

- *DEBUG=YES or NO*. Gather and report additional debugging trace information for each message. The default is to not collect additional information.

- *DEBUGDEST=Some.E-mail@destination.com.* Temporarily override the destination for debugging information.

  The default is *MailAgentFailureNotification@companye-mail.com.*

- *QMGR=QManagerName*. The program requires that you provide the queue manager name as the only parameter – see the **JavaMQMail.CMD** command above. If you want to override the setting in the *QueueManagerDEFS.TXT*, you can do it with this INI parameter.

So, in a nutshell:

- Add JavaMail, JAF, and JNDI to the MQ queue manager server.
- Add Process to SHOESS11, set *JAVA.MAIL.QUEUE* to trigger.
- Compile and move *JavaMQMail.class* to the queue manager server in *D:\MQM*, or wherever you want to put it.
- Define a trigger monitor for *SYSTEM.DEFAULT. INITIATION.QUEUE* on the queue manager server.

To get a better idea of the structure and operation of JavaMQMail I will describe the various methods and what they do.

public static void main(String[] args)

Every Java application needs a main() method. Because of the way I am sharing variables and values among the methods, I need to in essence create a new instance of the application from within itself. That's all that this main() method does.

public void mqMain()

The mqMain() method is the real start of the process. It reads the INI file and sets up the queue manager name and the debugging options and debug destination. Once it knows the queue manager it sets up the host name and channel name. It invokes other methods to read messages from the *JAVA.MAIL.QUEUE* and sends the messages to get processed in the processMessage()

method. At the end it writes a message to the Java console and to the logging queue, stating how many messages were processed.

public void processMessage()

processMessage() does the heavy lifting here. Three different kinds of message format are supported in this application. They all start with a two-digit code indicating what is to be done – the ‘doWhat’ code.

If ‘doWhat’ is ‘02’ the application will find a file on the local PC and attach it to an e-mail. After the two-digit ‘doWhat’ the next fields in the message must be the Send-To address for 48 characters, the From address for another 48 characters, the Subject field for 40 characters, then the FileName for 48 characters. It’s not important where the file is stored but it must match the name in the program.

If ‘doWhat’ is ‘03’ the application will take whatever text is supplied in the message and append it to the message. After the two-digit ‘doWhat’ the next fields must be the Send-To for 48, From for 48, and Subject for 40. Up to that point in fact, all the message formats are identical. But after the subject now comes a variable length text field; whatever is there will be mailed out as the message text.

If ‘doWhat’ is ‘06’ it is handled in almost the same way as ‘02’, where a file on the local PC is attached to an e-mail. But ‘06’ supports reporting back to the original sender via an error queue. The format for ‘06’ is the same as the others up through the subject; that is, it starts with the ‘doWhat’ code, the Send-To, From, and Subject. But after that it differs. The next field is the 48-character name of an error queue to which to report results. After that is an additional 52 characters of ‘key data’ to be returned to the original caller. After that is the 48-character file name to be attached to the message.

The original intent of ‘doWhat=06’ was for the mainframe program to send along a queue name and some identifying information. It would then listen for a message to appear back in that queue: this message could indicate either success or failure and would include the ‘key data’ to identify exactly where the problem or success occurred.

Messages are built using the JavaMail classes in multipart MIME format.

public void sendError(String errMsg)

sendError() formats any error messages and decides where and how to deliver them – via e-mail and/or via the supplied error queue back to the requestor.

public void sendGood()

sendGood() notifies the requestor that things worked okay by sending a success message to the indicated queue.

public void gatherDebug(String info)

There are many places in the program that generate debugging information. They all funnel through here. Most of the time we don't want all this information, we just want to know how many messages were handled and to whom they were addressed. But sometimes we really need as much additional information about the JavaMQMail process as we can get. gatherDebug() checks the debug switch and, if it's on, will accumulate the provided information. The log will get e-mailed or put onto a queue for later review and debugging.

public MQMessage getNextMessage(MQQueue localQueue)

This method simply grabs the next message from the indicated queue and handles any errors if there is nothing available.

public void attachFile()

As the name says, attachFile() is where any files are attached to the outgoing e-mail message.

public void sendMsgOut()

Here is where the e-mail message is actually mailed out.

public void logAction(String logMsg)

This method writes the supplied text into a message on the logging queue.

public String validateAddress(String addr)

Here is where we check the supplied addresses and verify that they are valid Internet-style addresses. If an address is not valid *validateAddress()* will consult with a nearby Exchange server via the *lookupLDAP()* method.

public String lookupLDAP(String addr)

Here is the LDAP code that first binds to LDAP then queries Exchange to get an SMTP e-mail address for the name that was sent to it. If nothing was found in Exchange it returns the original address.

public void sendFail(String myErrMsg)

If the e-mail send fails, this method informs the world.

INTERESTING DEVELOPMENTS

I first tried using the JRE instead of the full JDK. This gave me problems with 'UnsupportedEncodingException: Cp437' after the MQReadString() call. The errors went away when I installed the full JDK 1.3.1 instead.

Another problem: my agent would not trigger. After some basic debugging and suggestions from the MQSeries mailing list I finally realized that there was no trigger monitor running on the queue manager server. Well, of course. We'd never needed one before. To fix this, add the following line to the queue manager start-up commands:

```
   runmqtrm -m QueueManagerName -q SYSTEM.DEFAULT.INITIATION.QUEUE
```

The program code for *JavaMQMail.INI, READJAVALOG.JAVA*, and *JAVAMQMAIL.JAVA* can be found at www.xephon.com/ extras/JavaMQMail.txt.

APPENDIX A: INSTALLING THE MQ JAVA CLASSES FROM MA88

Obtain the MQ Java classes from SupportPac MA88. The Web site address is: *http://www-4.ibm.com/software/ts/mqseries/txppacs/ma88.html*.

- You must install Java – either the JDK or the JRE – on any server that will use the MQ Java classes and on any developer's PC.
- Unzip MA88 into a temporary directory. Use WinZip or some other utility with the ability to keep the names and directory structure intact.
- Run **setup.exe**. The classes will be installed into *c:\program files\IBM\MQSeries\Java*. I could not see a way to install them into a different directory.
- Update the CLASSPATH on the target PC to add the following:
  - *installdir\lib\com.ibm.mq.jar*
  - *installdir\lib\com.ibm.mqiiop.jar*
  - *installdir\lib\*
  - *installdir\samples\base\*

  Note that *installdir* is *c:\program files\IBM\MQSeries\Java*.
- Update the PATH to add the following:
  - *installdir\lib*
  - *installdir\bin*.
- Set the NT environment variable MQ_JAVA_INSTALL_PATH to the directory where JMS is installed.

MQ Java classes require JDK 1.2.2 or 1.3.1.

Joe Larson (USA) © Xephon 2003

Maximizing messaging availability: an update

In the article entitled *Maximizing message availability*, which was published in the September 2002 issue of *MQ Update, Table 1: Deciding which channel topology to use* (page 28), which summarized the use of generic channels, states that for messages greater than 63 kBytes the answer is 'no'!

However, there are circumstances when shared inbound channels can be used. The wording in the box should read 'No for outbound channels; maybe for inbound channels', with the following additional information.

- The size limitation stopping messages greater than 63KB being put onto a shared transmission queue makes the sending of outbound messages from shared channels an unavailable option.

  However, shared inbound channels can be used as long as no messages >63KB are destined for shared queues. Thus, if all messages are put to cloned private local queues, shared channels can be used.

- This option can enable a higher availability to be offered to a remote queue manager without any change to the remote queue manager.

  This could be of benefit if the remote queue manager belonged to another organization or if there were significant costs involved in making changes to the set-up on this.

- The corresponding increase in outbound availability would probably be achieved by clustering the outbound queue managers and using queue manager aliases to route the outbound messages through an available destination.

John B Jones, BSc, MSc, MIEE, CEng
IBM Hursley (UK)
© IBM 2003

Using MQAI to list queues

The MQSeries Administration Interface (MQAI) is an API that works in tandem with the standard MQSeries API to accomplish administrative functions, such as listing, creating, or destroying queues and other MQSeries objects. This article explores how to use this interface to generate a list of all queues on a queue manager. The programming language used in this article is Visual Basic 6.0, but the focus will be on the API and not the language so you should be able to generalize this technique to other platforms.

APPLICATIONS FOR MQAI

The standard MQSeries API is designed to let you send and receive messages but not to access processes, clusters, or channels. Neither does it support creating, deleting, or listing these objects. You can use tools such as MQExplorer to accomplish these tasks manually but you must use a special API such as MQAI to do these things programmatically. This article will walk you through the process of creating a list of queue names, which is a simple yet useful technique.

HOW MQAI WORKS

In a nutshell, MQSeries queue managers use a system queue named *SYSTEM.ADMIN.COMMAND.QUEUE* for administrative command input. The queue manager 'listens' to this queue and when it gets a message it will process the command and send a response back via a specified response queue.

Command messages must use Programmable Command Format (PCF) in order for the queue manager to understand and act upon them. PCF messages are organized into structures known as 'bags'. There may be several different types of bag within a message; the most important are administrative, response, and system bags. The administrative bag contains your command and supporting parameters. When the queue manager processes the command it returns a response bag containing a collection of system bags. Each system bag contains information about one of the MQ objects acted upon. For instance, if you request a list of all queues on the queue manager there will be a system bag returned for each queue and that bag will contain the queue's name.

SECURITY REQUIREMENTS

MQAI won't bypass MQSeries security so you must have the proper authorization for the objects you wish to access. You need:

- Queue manager (connect, inq, dsp).
- *SYSTEM.ADMIN.COMMAND.QUEUE* (put).
- *SYSTEM.DEFAULT.MODEL.QUEUE* (get, inq, dsp).
- Every object you want to list (inq, dsp).
- Every object you want to modify or delete (all).

STEPS TO USING MQAI AND PCF

MQAI works by sending request messages and receiving reply messages, but the API hides the underlying messaging from you. Use the steps detailed below to work with it.

- Connect to your queue manager.
- Create your bag objects by using the **mqCreateBag** command for the administrative and response bags. The system bags will be created for you by executing the query.
- Add your parameters and queries to the administrative bag by using the **mqAddString**, **mqAddInteger**, and **mqAddInquiry** commands. You can store multiple related queries in this bag.
- Execute the PCF command with **mqExecute**.
- Extract your data from the system bags embedded within the response bag with the **mqInquireBag**, **mqInquireString**, and **mqInquireInteger** commands.

Before adding the procedural code you should first declare the API commands, constants, structures, and variables. IBM provides a set of VB modules with the MQSeries client installation that defines all of this for you. You will want to add modules CMQB.BAS, CMQBB.BAS, CMQFB.BAS to your project. They were written in VB4 format and require a little tweaking in order for VB6 to use them. In the API declarations, change all parameters that look like 'ByVal QMgrName As String * 48' to 'ByVal QMgrName As String'. Finally, open your project properties, select the Make tab, and enter MqType = 2 in the conditional compilation arguments field. You are now ready to start coding.

The first thing to do is to connect to your queue manager. If you are more comfortable with the ActiveX API you can use that instead.

```
Dim intConnectionHandle As Long, intCC As Long, intRC As Long
' Connect to the queue manager.
MQCONN vData, intConnectionHandle, intCC, intRC
```

(Good programming requires you to check your completion code after each MQSeries call. However, I am omitting error logic from this sample for the sake of brevity.)

Next create your data bags. This involves declaring the bag variables, initializing them, and finally creating them with the **mqCreateBag** command. This command returns the standard MQSeries completion code and reason code variables.

```
Dim adminBag As Long    ' Admin bag handle.
Dim systemBag As Long   ' System bag handle.
Dim responseBag As Long ' Response bag handle.
' Initialize the bags.
adminBag = MQHB_UNUSABLE_HBAG
systemBag = MQHB_UNUSABLE_HBAG
responseBag = MQHB_UNUSABLE_HBAG
' Create an admin bag.
mqCreateBag MQCBO_ADMIN_BAG, adminBag, intCC, intRC
' Create a response bag.
mqCreateBag MQCBO_ADMIN_BAG, responseBag, intCC, intRC
```

Set up your query in the administrative bag. This involves specifying what type of object you want information on and what information you want about it. You can get several pieces of information at

© 2003. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

once but in this example we will start simply with just the queue name. If the data you want is in string format use the **mqAddString** command to request it. Otherwise, use **mqAddInteger** for numeric data. Some pieces of data require the **mqAddInquir**y command but we'll get to that one later.

```
' Put generic local queue name into admin bag.
mqAddString adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, "*", _
              intCC, intRC
```

Use **mqExecute** to execute the query. Notice that **mqExecute** uses the connection handle that you obtained from connecting to the queue manager. This example uses a default dynamic queue for the response messages upon which the response bag is built. To specify a static queue change the last MQHO_NONE parameter to a string containing the name of the response queue you wish to use.

```
  ' Send command to the queue manager.
  mqExecute intConnectionHandle, MQCMD_INQUIRE_Q, MQHB_NONE, adminBag,
_
              responseBag, MQHO_NONE, MQHO_NONE, intCC, intRC
```

Once you have executed the query you must access the data that has been returned to you in the response bag. First, find out how many system bags are in the response bag with the **mqCountItems** command. Next, you access each system bag with the **mqInquireBag** command. Finally, extract your information with the **mqInquireString** or **mqInquireInteger** commands.

```
' Count how many system bags are embedded in the response bag.
mqCountItems responseBag, MQHA_BAG_HANDLE, intQueueCount, _
             intCC, intRC
For i = Ø To intQueueCount - 1
    ' Get the system bag handle.
    mqInquireBag responseBag, MQHA_BAG_HANDLE, i, systemBag, _
                 intCC, intRC
    ' Get the queue name from the system bag.
    mqInquireString systemBag, MQCA_Q_NAME, Ø, MQ_Q_NAME_LENGTH,
      strQName, _
      intQNameLength, Ø, intCC, intRC
    ' Do something with the queue name.
    MsgBox "Queue Name: " & strQName
Next
```

Once you have finished working with MQAI delete your data bags with the **mqDeleteBag** command.

```
' Clean up
If adminBag > Ø Then _
    mqDeleteBag adminBag, intCC, intRC
If systemBag > Ø Then _
    mqDeleteBag systemBag, intCC, intRC
If responseBag > Ø Then _
    mqDeleteBag responseBag, intCC, intRC
' Disconnect the queue manager.
MQDISC intConnectionHandle, intCC, intRC
```

EMBELLISHING A THEME

While you are listing your queue names you may also want other pieces of data, for example queue type or queue depth, or you may wish to get only queue names of a certain type.

To add a filter for a specific queue type add this code, where variable *intTypeFilter* is a number representing the type of queue you want (see Table 1). This will limit the list to queues of the type you specify.

```
  ' Put local queue type into admin bag.
  mqAddInteger adminBag, MQIA_Q_TYPE, intTypeFilter, _
               mvarCompletionCode, mvarReasonCode
```

If you want to get the queue depth along with each queue name add this code when building the administrative bag:

```
' Add inquiry for current queue depths.
mqAddInquiry adminBag, MQIA_CURRENT_Q_DEPTH, mvarCompletionCode,
mvarReasonCode
```

Add this code to get the queue depth from the system bag. Note that the command will fail if you try to get the queue depth from an inappropriate queue type (ie remote).

```
' Get the queue depth from the system bag.
```

1 – Local	6 – Remote
2 – Model	7 – Cluster
3 – Alias	

Table 1: Number codes for queue type

```
  mqInquireInteger systemBag, MQIA_CURRENT_Q_DEPTH, MQIND_NONE,
  arDepth(i), _
                 mvarCompletionCode, mvarReasonCode
```

You can get the queue type along with your other data by adding this inquiry to the administrative bag.

```
  ' Add inquiry for queue types.
  mqAddInquiry adminBag, MQIA_Q_TYPE, mvarCompletionCode,
mvarReasonCode
```

To get it from the system bag, use the following command. It will give you a numeric code corresponding to the queue type (see Table 1).

```
  ' Get the queue type from the system bag.
  mqInquireInteger systemBag, MQIA_Q_TYPE, MQIND_NONE, QType, _
                 mvarCompletionCode, mvarReasonCode
```

ADDITIONAL RESOURCES

The following MQSeries manuals can provide additional help with the MQAI.

- *MQSeries Administration Interface Programming Guide and Reference*: see Chapter 6 for more samples.
- *MQSeries Programmable System Management*: this manual documents each PCF command.

CLSMQQUEUELIST.CLS

```
VERSION 1.Ø CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = Ø  'NotPersistable
  DataBindingBehavior = Ø  'vbNone
  DataSourceBehavior  = Ø  'vbNone
  MTSTransactionMode  = Ø  'NotAnMTSObject
END
Attribute VB_Name = "clsMQQueueList"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Attribute VB_Description = "This class provides access to the list of
queue names for a specified queue manager."
```

```
Attribute VB_Ext_KEY = "SavedWithClassBuilder6" ,"Yes"
Attribute VB_Ext_KEY = "Top_Level" ,"Yes"
Option Explicit
'local variable(s) to hold property value(s)
Private mvarConnectionHandle    As Long
Private mvarCompletionCode      As Long
Private mvarReasonCode          As Long
Private mvarErrorMessage        As String
Private mvarQueueCount          As Long
Private mvarTypeFilter          As enmQueueType
Private mvarQMgrName            As String * 48
Public Enum enmQueueType
    qtAll = Ø
    qtLocal = 1
    qtModel = 2
    qtAlias = 3
    qtRemote = 6
    qtCluster = 7
End Enum
' Private variables.
Private arQueue() As String
Private arType()  As String
Private arDepth() As Long
Private gbolDisconnect As Boolean
' MQSeries constants.
Const MQ_Q_MGR_NAME_LENGTH = 48
Const MQ_Q_NAME_LENGTH = 48
Const MQBL_NULL_TERMINATED = -1
Const MQCA_Q_NAME = 2Ø16
Const MQCA_REMOTE_Q_NAME = 2Ø18
Const MQCMD_INQUIRE_Q = 13
Const MQCBO_ADMIN_BAG = 1
Const MQCC_OK = Ø
Const MQCC_WARNING = 1
Const MQCC_FAILED = 2
Const MQHA_BAG_HANDLE = 4ØØ1
Const MQHB_NONE = -2
Const MQHB_UNUSABLE_HBAG = -1
Const MQHO_NONE = Ø
Const MQIA_Q_TYPE = 2Ø
Const MQIA_CURRENT_Q_DEPTH = 3
Const MQIND_NONE = -1
Const MQQT_LOCAL = 1
Const MQQT_REMOTE = 6
' API Declarations.
Private Declare Sub mqCountItems Lib "MQIC32.DLL" Alias
"mqCountItemsstd@2Ø" (ByVal Bag As Long, ByVal Selector As Long,
ItemCount As Long, CompCode As Long, Reason As Long)
Private Declare Sub mqCreateBag Lib "MQIC32.DLL" Alias
"mqCreateBagstd@16" (ByVal Options As Long, Bag As Long, CompCode As
Long, Reason As Long)
Private Declare Sub mqDeleteBag Lib "MQIC32.DLL" Alias
"mqDeleteBagstd@12" (Bag As Long, CompCode As Long, Reason As Long)
Private Declare Sub mqAddInquiry Lib "MQIC32.DLL" Alias
"mqAddInquirystd@16" (ByVal Bag As Long, ByVal Selector As Long,
CompCode As Long, Reason As Long)
Private Declare Sub mqAddInteger Lib "MQIC32.DLL" Alias
"mqAddIntegerstd@2Ø" (ByVal Bag As Long, ByVal Selector As Long, ByVal
ItemValue As Long, CompCode As Long, Reason As Long)
Private Declare Sub mqAddString Lib "MQIC32.DLL" Alias
"mqAddStringstd@24" (ByVal Bag As Long, ByVal Selector As Long, ByVal
BufferLength As Long, ByVal Buffer As String, CompCode As Long, Reason
As Long)
Private Declare Sub mqExecute Lib "MQIC32.DLL" Alias "mqExecutestd@36"
(ByVal Hconn As Long, ByVal Command As Long, ByVal OptionsBag As Long,
ByVal CommandBag As Long, ByVal responseBag As Long, ByVal CommandQ As
Long, ByVal ResponseQ As Long, CompCode As Long, Reason As Long)
Private Declare Sub mqInquireBag Lib "MQIC32.DLL" Alias
"mqInquireBagstd@24" (ByVal Bag As Long, ByVal Selector As Long, ByVal
ItemIndex As Long, ItemValue As Long, CompCode As Long, Reason As Long)
Private Declare Sub mqInquireInteger Lib "MQIC32.DLL" Alias
"mqInquireIntegerstd@24" (ByVal Bag As Long, ByVal Selector As Long,
ByVal ItemIndex As Long, ItemValue As Long, CompCode As Long, Reason As
Long)
Private Declare Sub mqInquireString Lib "MQIC32.DLL" Alias
"mqInquireStringstd@36" (ByVal Bag As Long, ByVal Selector As Long,
ByVal ItemIndex As Long, ByVal BufferLength As Long, ByVal Buffer As
String, StringLength As Long, CodedCharSetId As Long, CompCode As Long,
Reason As Long)
Private Declare Sub MQCONN Lib "MQIC32.DLL" Alias "MQCONNstd@16" (ByVal
QMgrName As String, Hconn As Long, CompCode As Long, Reason As Long)
Private Declare Sub MQDISC Lib "MQIC32.DLL" Alias "MQDISCstd@12" (Hconn
As Long, CompCode As Long, Reason As Long)
Public Sub LoadListControl(ByRef ListControl As Control)
    Dim i As Long
        If TypeOf ListControl Is ListBox _
    Or TypeOf ListControl Is ComboBox Then
        ListControl.Clear
        For i = Ø To QueueCount - 1
            ListControl.AddItem Queue(i)
        Next i
    End If
End Sub
Public Property Get QueueType(ByVal intIndex As Long) As String
    QueueType = arType(intIndex)
End Property
Public Property Let TypeFilter(ByVal vData As enmQueueType)
    mvarTypeFilter = vData
End Property
```

```
Public Property Get TypeFilter() As enmQueueType
    TypeFilter = mvarTypeFilter
End Property
Public Property Let QMgrName(ByVal vData As String)
    Dim intHandle As Long, intCC As Long, intRC As Long
    ' Connect to the queue manager.
    MQCONN vData, intHandle, intCC, intRC
    If intCC = MQCC_FAILED Then
        Err.Raise 10000, "clsMQQueueList", "Error " & intRC & "
connecting to queue manager " & vData
    End If
    gbolDisconnect = True
    mvarQMgrName = vData
    ConnectionHandle = intHandle
End Property
Public Property Get QMgrName() As String
    QMgrName = mvarQMgrName
End Property
Public Property Get Depth(ByVal intIndex As Long) As Long
    Depth = arDepth(intIndex)
End Property
Public Property Let ConnectionHandle(ByVal vData As Long)
    Dim adminBag As Long    ' Admin bag handle.
    Dim systemBag As Long   ' System bag handle.
    Dim responseBag As Long ' Response bag handle.
    Dim intCompCode As Long, intReason As Long
    Dim i As Long, QType As enmQueueType
    Dim strQName As String * MQ_Q_NAME_LENGTH, intQNameLength As Long
    ' Update the property value.
    mvarConnectionHandle = vData
        ' Get the queue name list.
    adminBag = MQHB_UNUSABLE_HBAG
    systemBag = MQHB_UNUSABLE_HBAG
    responseBag = MQHB_UNUSABLE_HBAG
    ' Create an admin bag.
    mqCreateBag MQCBO_ADMIN_BAG, adminBag, mvarCompletionCode,
mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqCreateBag failure on admin bag."
        Exit Property
    End If
    ' Create a response bag.
    mqCreateBag MQCBO_ADMIN_BAG, responseBag, mvarCompletionCode,
mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqCreateBag failure on response bag."
        Exit Property
    End If
    ' Put generic local queue name into admin bag.
    mqAddString adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, "*",
mvarCompletionCode, mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqAddString failure."
        Exit Property
    End If
    If mvarTypeFilter <> qtAll Then
        ' Put local queue type into admin bag.
        mqAddInteger adminBag, MQIA_Q_TYPE, mvarTypeFilter,
mvarCompletionCode, mvarReasonCode
        If mvarCompletionCode = MQCC_FAILED Then
            mvarErrorMessage = "mqAddInteger failure."
            Exit Property
        End If
    End If
    ' Add inquiry for current queue depths.
    mqAddInquiry adminBag, MQIA_CURRENT_Q_DEPTH, mvarCompletionCode,
mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqAddInquiry failure for queue depth."
        Exit Property
    End If
    ' Add inquiry for queue types.
    mqAddInquiry adminBag, MQIA_Q_TYPE, mvarCompletionCode,
mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        MsgBox "mqAddInquiry failure for queue type.  RC=" &
mvarReasonCode
        End
    End If
    ' Send command to the queue manager.
    mqExecute mvarConnectionHandle, MQCMD_INQUIRE_Q, MQHB_NONE,
adminBag, responseBag, MQHO_NONE, MQHO_NONE, _
              mvarCompletionCode, mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqExecute failure."
        Exit Property
    End If
    ' Count number of system bags are embedded in the response bag.
    mqCountItems responseBag, MQHA_BAG_HANDLE, mvarQueueCount,
mvarCompletionCode, mvarReasonCode
    If mvarCompletionCode = MQCC_FAILED Then
        mvarErrorMessage = "mqCountItems failure."
        Exit Property
    End If
        ReDim arQueue(mvarQueueCount)
    ReDim arType(mvarQueueCount)
    ReDim arDepth(mvarQueueCount)
    For i = Ø To mvarQueueCount - 1
        ' Get the system bag handle.
        mqInquireBag responseBag, MQHA_BAG_HANDLE, i, systemBag,
mvarCompletionCode, mvarReasonCode
        If mvarCompletionCode = MQCC_FAILED Then
            mvarErrorMessage = "mqInquireBag failure."
            Exit Property
        End If
        ' Get the queue name from the system bag.
        mqInquireString systemBag, MQCA_Q_NAME, Ø, MQ_Q_NAME_LENGTH,
strQName, _
                        intQNameLength, Ø, mvarCompletionCode,
mvarReasonCode
        If mvarCompletionCode = MQCC_FAILED Then
            mvarErrorMessage = "mqInquireString failure."
            Exit Property
        End If
                ' Get the queue type from the system bag.
        mqInquireInteger systemBag, MQIA_Q_TYPE, MQIND_NONE, QType, _
                         mvarCompletionCode, mvarReasonCode
        If mvarCompletionCode = MQCC_FAILED Then
            MsgBox "mqInquireInteger failure on queue type.  RC=" &
mvarReasonCode
            End
        End If
                Select Case QType
            Case qtLocal:   arType(i) = "Local"
            Case qtModel:   arType(i) = "Model"
            Case qtAlias:   arType(i) = "Alias"
            Case qtRemote:  arType(i) = "Remote"
            Case qtCluster: arType(i) = "Cluster"
        End Select
                If QType = qtLocal Then
            ' Get the queue depth from the system bag.
            mqInquireInteger systemBag, MQIA_CURRENT_Q_DEPTH,
MQIND_NONE, arDepth(i), _
                             mvarCompletionCode, mvarReasonCode
            If mvarCompletionCode = MQCC_FAILED Then
                mvarErrorMessage = "mqInquireInteger failure."
                Exit Property
            End If
        Else
            arDepth(i) = Ø
        End If
        arQueue(i) = Trim(strQName)
    Next i
    ' Clean up
    If adminBag > Ø Then _
        mqDeleteBag adminBag, mvarCompletionCode, mvarReasonCode      '
Delete the admin bag.
    If systemBag > Ø Then _
        mqDeleteBag systemBag, mvarCompletionCode, mvarReasonCode     '
Delete the system bag.
```

```
    If responseBag > Ø Then _
        mqDeleteBag responseBag, mvarCompletionCode, mvarReasonCode   '
Delete the response bag.
End Property
Public Property Get ConnectionHandle() As Long
    ConnectionHandle = mvarConnectionHandle
End Property
Public Property Get Queue(ByVal intIndex As Long) As String
    Queue = arQueue(intIndex)
End Property
Public Property Get QueueCount() As Long
    QueueCount = mvarQueueCount
End Property
Public Property Get ErrorMessage() As String
    ErrorMessage = mvarErrorMessage
End Property
Public Property Get ReasonCode() As Long
    ReasonCode = mvarReasonCode
End Property
Public Property Get CompletionCode() As Long
    CompletionCode = mvarCompletionCode
End Property
Private Sub Class_Initialize()
End Sub
Private Sub Class_Terminate()
    Dim intCC As Long, intRC As Long
    If gbolDisconnect Then MQDISC ConnectionHandle, intCC, intRC
End Sub
```

Mills Perry
IT Consultant/Instructor
ZyQuest (USA)

© Xephon 2003

Data grouping in WebSphere MQIntegrator using dynamic field resolution

INTRODUCTION

When manipulating data in WebSphere MQIntegrator (WMQI), simple transformation and augmentation of a message is not always sufficient.

A single message may contain multiple records and it may be necessary to reorder this data, summarize it, or perform grouping operations on it.

WMQI presents a powerful model of a message to a user via the SELECT statement: an incoming message can be treated as if it were a relational database and the message can be transformed arbitrarily by judicious use of SELECTs.

However, the WMQI implementation of SELECT does not include the GROUP BY clause or allow the order of the resulting rows to be controlled.

In this article we describe a typical scenario where this behaviour may be required and show how the dynamic field resolution syntax introduced in WMQI V2.1 can be used to solve the problem.

SAMPLE SCENARIO

Consider the common scenario outlined below.

An incoming message consists of several line items. Each line item could represent a bank transfer, for example. Each bank transfer item includes bank account number, branch-ID (sort code), and payment amount.

The requirement is that multiple messages should be produced – one for each sort code (ie there would be *n* output messages if there were *n* unique sort codes in the input message).

Record structure

A sample input message would be:

```
20-00-00 12345678 00100.00
15-13-11 11112222 00543.23
12-34-56 87654321 00011.00
15-13-11 44445555 00000.99
20-00-00 33334444 10000.00
```

The three output messages would be:

```
20-00-00 12345678 00100.00
```

```
20-00-00 33334444 10000.00

15-13-11 11112222 00543.23
15-13-11 44445555 00000.99

12-34-56 87654321 00011.00
```

Variations on this requirement would include 'summary' operations (perhaps the total or average value for each group must be calculated). It may also be necessary to place some order on the output data (perhaps only one message should be produced but the line items should be grouped by sort code and ordered within these groups by account number).

DYNAMIC FIELD ADDRESSING

The solution outlined in this article relies on the dynamic field addressing feature in WMQI V2.1. The feature is described briefly in the *ESQL Reference Manual*.

Consider the sample tree illustrated in Figure 1. One could reference the current IBM stock price through:

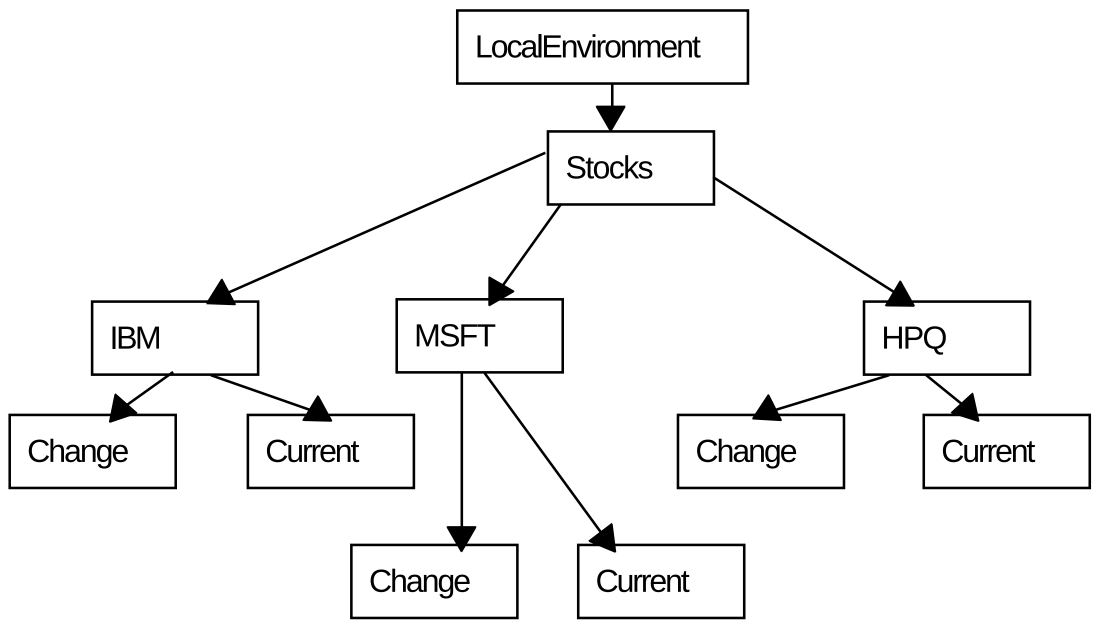

Figure 1: Dynamic field addressing – example tree

```
LocalEnvironment.Stocks.IBM.Current
```

However, if one wants to choose the ticker symbol at runtime the dynamic field addressing syntax is very useful.

Consider the following:

```
DECLARE symbol CHAR;
SET symbol='MSFT';
```

We can now use a reference of the form:

```
LocalEnvironment.Stocks.{symbol}.Current
```

This portion of the path will be resolved at runtime. (Note that there is no '$' symbol – don't confuse this syntax with Perl! The key point here is that the value of the 'symbol' field could be obtained from an incoming message or a database table, or computed in some other way.

Furthermore, we can use this syntax to create tree elements. A statement of the following form will cause the relevant tree segments to be created.

```
DECLARE symbol CHAR;
SET symbol = 'FON';
SET LocalEnvironment.Stocks.{ symbol }.Current=
  InputRoot.XML.Stocks.FON.Current;
```

Aside on Field Reference Variables

The Field Reference Variable feature in WMQI allows syntax of the following form to be written.

```
DECLARE treewalker REFERENCE TO LocalEnvironment.Stocks;
MOVE treewalker FIRSTCHILD NAME 'IBM';
```

Combined with the CREATE FIELD syntax we can also use this technique to create and navigate trees dynamically. However, this syntax is not ideal for dynamic generation of new trees since it is a little cumbersome.

SAMPLE SOLUTION

Presented here is a partial solution to the problem of producing multiple output messages (one per sort code) based on one incoming message. It is not intended to be an optimal solution. In particular, the repeated use of the CARDINALITY function is not recommended. A better solution would keep track of the size of the arrays to avoid the repeated recalculations.

```
-- We assume the line items are in the Environment.Data.Body[] array
-- Obtain reference to the first line item
-- Note that we perform the operation in two stages:
-- we declare the reference to an element which may have no
-- children and then attempt to move to the first child.
-- Had we declared the reference to the child directly,
-- then failure would have been harder to detect.
DECLARE treewalker REFERENCE TO Environment.Data;
MOVE treewalker FIRSTCHILD NAME 'Body';
DECLARE sortcode CHAR;
-- This variable will be used to track how many
-- instances of this sort code we have already
-- seen
DECLARE card INTEGER;
-- The indexcount field is used to track how many
-- unique sort codes we have seen
DECLARE indexcount INTEGER;
SET indexcount=1;
WHILE(LASTMOVE(treewalker)) DO
  SET sortcode=treewalker.SortCode;
  -- How many instances of this sort
  -- sort code have we already seen?
  SET card = CARDINALITY(Environment.Group.{sortcode}[]);
  IF(card=Ø) THEN
    -- If this is the first instance, store the
    -- sort code in a separate array
    -- This allows us to sort on this field more easily
    -- should we choose to, for example
    SET Environment.Index[indexcount]=sortcode;
    SET indexcount=indexcount+1;
  END IF;
  -- Copy this line item into the next position in
  -- this sort-code's portion of the tree
  SET Environment.Group.{sortcode}[card+1] = treewalker;
  MOVE treewalker NEXTSIBLING;
END WHILE;
-- We have now grouped all the line items by sort code
-- We now generate a new output message for each sort
-- code
DECLARE I INTEGER;
-- This is redundant I guess :-)
SET indexcount = CARDINALITY(Environment.YKB.Index[]);
-- record count will store how many line items
-- there are for the current sort code
```

```
DECLARE recordcount INTEGER;
-- recorditeration tracks which item we are currently
-- working on
DECLARE recorditeration INTEGER;
-- Now create output messages
-- indexiter tracks which sort code (the first, second
-- or third, etc) we are dealing with
DECLARE indexiter INTEGER;
SET indexiter = 1;
WHILE(indexiter <= indexcount) DO
  -- Create output messages
  -- copy message header
  SET I = 1;
  WHILE I < CARDINALITY(InputRoot.*[]) DO
     SET OutputRoot.*[I] = InputRoot.*[I];
   SET I=I+1;
  END WHILE;
  -- we would specify the message set, etc., here
  SET sortcode=Environment.Index[indexiter];
  DECLARE bodywalker REFERENCE To Environment. Group.{sortcode}[1];
  -- We have now navigated to the portion
  -- of the tree which stores the records for
  -- this sort code
    SET recordcount = CARDINALITY(Environment.YKB.Group.{sortcode}[]);
  SET recorditeration=1;
  -- We now iterate over each line item for this branch id

  WHILE(recorditeration<=recordcount) DO
    SET "OutputRoot"."MRM"."DETAILS_ELEMENT"[recorditeration].SortCode
= bodywalker.SortCode;
    -- set the other fields here
    -- move to the next line item
    MOVE bodywalker NEXTSIBLING;
    SET recorditeration=recorditeration+1;
  END WHILE;
    SET indexiter=indexiter+1;
  PROPAGATE;
END WHILE;
-- PROPAGATE has been used to propagate each message
-- The default behaviour of the compute node is to
-- propagate the current tree in the OutputRoot
-- tree. Returning FALSE inhibits this behaviour
RETURN FALSE;
```

Richard G Brown
Technical Sales
IBM Hursley (UK) ©IBM 2003

Channel Event Queue Viewer for MQ for OS/390

INSTALLATION

To install the Channel Event Queue Viewer follow the steps given below:

- Send the REXX code to your mainframe (ASCII mode in FTP or ASCII and CR/LF in Personal Communications file transfer).
- Store it as a library member (RECFM=FB, LRECL=80) with the name *MQEVNCHL*.

Now you are ready to use it! To do so:

- Read this article.
- Use the supplied code (after tailoring to your site's requirements) to run the Channel Event Queue Viewer against your *SYSTEM.ADMIN.CHANNEL.EVENT* queue.

USE

General information – MQSeries channel events

MQSeries has the facility to gather events that relate to its work. There are three types of event (called 'instrumentation events') and each category has its own event queue:

- Queue manager events (SYSTEM.ADMIN.QMGR.EVENT).
- Channel events (SYSTEM.ADMIN.CHANNEL.EVENT).
- Performance events (SYSTEM.ADMIN.PERFM.EVENT).

If a relevant queue is not available events are ignored.

I will describe channel events only briefly – I suggest you consult the MQSeries manuals if you require more detailed information.

Channel events are basically notifications about the condition of a channel during its operation, eg ‘channel stopped’, ‘conversion error’, etc.

Channel events are generated:

- When a channel instance starts or stops.
- When a channel receives a conversion error warning when getting a message.
- When an attempt is made to create a channel automatically; the event is generated whether the attempt succeeds or fails.

Remember that client connections to MQSeries for OS/390 5.2 do not cause ‘Channel Started’ or ‘Channel Stopped’ events.

There are several channel events (event descriptions are taken from the manual on event monitoring):

- Channel activated – this condition is detected when a channel that has been waiting to become active and for which a ‘Channel Not Activated’ event has been generated is now able to become active because an active slot has been released by another channel. This event is not generated for a channel that is able to become active without waiting for an active slot to be released.
- Channel auto-definition error – this condition is detected when the automatic definition of a channel fails. Additional information indicating the reason for the failure is returned in the event message.

  This does not apply to MQSeries for OS/390 V2.1 and earlier.

  Channel Event Viewer does not support this event.
- Channel auto-definition OK – this condition is detected when the automatic definition of a channel is successful.

  This does not apply to MQSeries for OS/390 V2.1 and earlier.

  Channel Event Viewer does not support this event.
- Channel conversion error – this condition is detected when a channel is unable to carry out data conversion and the

MQGET call to get a message from the transmission queue has resulted in a data conversion terror. Additional information indicating the reason for the failure is returned in the event message.

- Channel not activated – this condition is detected when a channel is required to become active either because it is starting or because it is about to make another attempt to establish connection with its partner. However, it is unable to do so because the limit on the number of active channels has been reached.
- Channel started – this condition is detected when initial data negotiation is complete and resynchronization has been performed where necessary, such that message transfer can proceed.
- Channel stopped – this condition is detected when a channel has been stopped. Additional information indicating the reason for the failure is returned in the event message.

Most channel events are enabled by default and there is no command to disable them. The exceptions are the two automatic channel definition events.

All messages that are put to event queues have a special format, namely an additional header that is appended in the front of the application data. The format of a message sent to an event queue is set to MQFMT_EVENT.

RUNNING THE CHANNEL EVENT QUEUE VIEWER

The Channel Event Queue Viewer can be invoked either from the supplied job or run from a TSO or ISPF line command. The last two options are not recommended, since, depending on the number of messages on the channel event queue, the Channel Event Queue Viewer can produce a significant output.

The job to run the Channel Event Queue Viewer is shown and explained below with line numbers in brackets for reference.

© 2003. Reproduction prohibited. Please inform Xephon of any infringement.

```
//MQENVCHL JOB NOTIFY=&SYSUID                                      (1)
//*
//* parameters in SYSTSIN:
//*  QMgrName
//*  ChannelEventQueueName
//*
//GETENV    EXEC PGM=IKJEFT01,
//          PARM='%MQENVCHL'
//STEPLIB   DD DSN=MA18.LOAD,DISP=SHR                               (2)
//          DD DSN=MQM.SCSQAUTH,DISP=SHR                            (3)
//SYSEXEC   DD DSN=YOUR.DSN,DISP=SHR                                (4)
//SYSTSPRT  DD SYSOUT=*                                             (5)
//SYSTSIN   DD *
QmgrName                                                            (6a)
ChannelEventQueueName                                               (6b)
/*
```

Tailoring

- Line 1: supply a valid job card.
- Line 2: MA18 load library.
- Line 3: MQSeries hlq.SCSQAUTH library.
- Line 4: library containing the MQENVCHL source.
- Line 5: a class for sysout (the Channel Event Queue Viewer reports will be printed there).
- Line 6: parameters for the Channel Event Queue Viewer:
  - line 6 (a): queue manager name (required)
  - line 6 (b): channel event queue Name (optional).

Parameters for the Channel Event Queue Viewer must come in the order shown the last one may be omitted (leave blank line instead) but the queue manager name is required as it has no default.

Default values for the Channel Event Queue Viewer parameters

- QMgr – none.
- ChannelEventQueueName – none; if blank it defaults to SYSTEM.ADMIN.CHANNEL.EVENT.

OUTPUT

The Channel Event Queue Viewer will produce a report for every message it finds on the channel event queue. The following information will be given:

- The message descriptor, MQMD. I will not describe here the full message descriptor (please refer to *MQSeries Application Programming Reference Manual*), but a couple of fields need a comment since they are subject to change when the message is put to the dead letter queue.
  - format – always set to MQFMT_EVENT
  - type – always set to MQMT_DATAGRAM
  - PutTime and PutDate – time and date when the message was put to the channel event queue.
- The contents of the event header. The fields listed below will always be present:
  - type – always set to EVENT
  - reason – reason of the event; it is one of the following:
    - MQRC_CHANNEL_ACTIVATED
    - MQRC_CHANNEL_AUTO_DEF_ERROR
    - MQRC_CHANNEL_AUTO_DEF_OK
    - MQRC_CHANNEL_CONV_ERROR
    - MQRC_CHANNEL_NOT_ACTIVATED
    - MQRC_CHANNEL_STARTED
    - MQRC_CHANNEL_STOPPED
    - queue manager name – name of the queue manager
    - channel name – name of the channel for which the event was generated.

  There are also additional fields depending upon the event and/or channel types:

- Connection Name – if the channel has successfully established a TCP connection this is the Internet address. Otherwise it is the contents of the CONNAME field in the channel definition.
- Transmission Queue Name – present for sender, server, cluster-sender, and cluster-receiver channels only.
- Conversion Reason Code – reason for which the conversion has failed; present in conversion error events only.
- Format – format of the message for which the conversion has failed; present in conversion error events only.
- Reason Qualifier – present for channel-stopped events only.
- Error Identifier – present for channel-stopped events only. The complete error identifier and/or message will be presented in the following fields:
  - Aux Error Data Int 1
  - Aux Error Data Int 2
  - Aux Error Data Str 1
  - Aux Error Data Str 2
  - Aux Error Data Str 3.

TAKE NOTE!

I originally wrote the Channel Event Viewer for MQSeries for OS/390 V2.1. In this version the channel auto definitions events were not supported. Although I now have V5.2 I still do not support those two events. They will be read from the queue and almost all information will be shown except for event-specific fields.

After reading all messages from the Channel event queue the Channel Event Queue Viewer will print the total number and quit.

© 2003. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

PROGRAMMING REMARKS

I have written the Channel Event Queue Viewer in REXX. Since the standard MQSeries API is not available for REXX I used SupportPac MA18.

MQEVNCHL.REX

```
/* REXX                                              by Marcin Grabinski
*/

MaxMsgLen = 500                    /* Channel event message lengths
*/
                                   /* differ, 500 is enough for all
*/
PARSE EXTERNAL QMgr
PARSE EXTERNAL EventQ

IF EventQ = '' THEN
  EventQ = 'SYSTEM.ADMIN.CHANNEL.EVENT'

SAY
SAY 'Channel Event Queue Viewer written by Marcin Grabinski'
SAY
/* Initialize the interface */
RXMQVTRACE = ''
rcc= RXMQV('INIT')
SAY rcc
/* Connect to Queue Manager */
RXMQVTRACE = ''
rcc = RXMQV('CONN', QMgr)
SAY rcc
IF WORD(rcc, 5) <> 'FAILED' THEN /* Connect OK */
DO
  /* Open Queue for Input */
  RXMQVTRACE = ''
  oo = MQOO_INPUT_AS_Q_DEF + MQOO_INQUIRE
  rcc = RXMQV('OPEN', EventQ, oo , 'h2', 'ood.' )
  SAY  rcc
  IF WORD(rcc, 5) <> 'FAILED' THEN /* Open OK */
  DO
    /* Get Current Queue Depth */
    RXMQVTRACE = ''
    rcc = RXMQV('INQ', h2, MQIA_CURRENT_Q_DEPTH, 'depth')
    SAY  rcc
    /* Read messages                    */
    RXMQVTRACE = ''
    DO i = 1 TO depth
```

```
g.0      = MaxMsgLen
g.1      = ''
igmo.opt = MQGMO_ACCEPT_TRUNCATED_MSG
rcc = RXMQV('GET', h2,'g.','igmd.','ogmd.','igmo.','ogmo.')
SAY  rcc
IF ( WORD(rcc,1) = 2033 ) THEN LEAVE
SAY
SAY 'Event message #'i':'
SAY
SAY '  The event header is 'g.0' bytes long'
SAY '  Message Descriptor: '
SAY '     MsgId:            'ogmd.msgid
SAY '     CorelId:          'ogmd.cid
SAY '     Report:           'ogmd.rep
SAY '     MsgType:          'ogmd.msg
SAY '     Expiry:           'ogmd.exp
SAY '     Feedback:         'ogmd.fbk
SAY '     Encoding:         'ogmd.enc
SAY '     CodedCharSetId:   'ogmd.ccsi
SAY '     Format:           'ogmd.form
SAY '     Priority:         'ogmd.pri
SAY '     Persistence:      'ogmd.per
SAY '     BackoutCount:     'ogmd.bc
SAY '     ReplyToQ:         'ogmd.rtoq
SAY '     ReplyToQMgr:      'ogmd.rtoqm
SAY '     UserId:           'ogmd.uid
SAY '     AccountingToken:  'ogmd.at
SAY '     ApplIdentityData: 'ogmd.aid
SAY '     PutApplType:      'ogmd.pat
SAY '     PutApplName:      'ogmd.pan
SAY '     PutDate:          'ogmd.pd
SAY '     PutTime:          'ogmd.pt
SAY '     ApplOriginData:   'ogmd.aod
SAY
/* Extract the event header  */
rcc = RXMQV('EVENT', 'g.', 'x.')
SAY  rcc
IF WORD(rcc, 1) = 0 THEN /* Event header ok */
DO
  INTERPRET 'rtext = RXMQV.RCMAP.'x.REA
  SAY '  Event header:'
  SAY '     Type:                 'x.TYPE
  SAY '     Reason:               'x.REA' ('rtext')'
  /* Fields common for all channel events */
  SAY '     Queue Manager Name:   'x.QM
  SAY '     Channel Name:         'x.CN
  /* Fields dependant on channel type */
  IF x.CONN <> 'X.CONN' THEN
    SAY '     Connection Name:       'x.CONN
  IF x.XQN <> 'X.XQN' THEN
```

```
          SAY '     Transmission Queue:    'x.XQN
        /* Channel stop and conversion error events have some more */
        IF rtext = 'MQRC_CHANNEL_CONV_ERROR' THEN
        DO
          SAY '     Conversion Reason Code:'x.CONVRC
          SAY '     Format:                'x.FORMAT
        END
        IF rtext = 'MQRC_CHANNEL_STOPPED' THEN
        DO
          SAY '     Reason Qualifier:      'x.RQUAL
          SAY '     Error Identifier:      'x.EID
          SAY '     Aux Error Data Int 1:  'x.AED1
          SAY '     Aux Error Data Int 2:  'x.AED2
          SAY '     Aux Error Data Str 1:  'x.AEDS1
          SAY '     Aux Error Data Str 2:  'x.AEDS2
          SAY '     Aux Error Data Str 3:  'x.AEDS3
          SAY
        END
        SAY
     END /* Event header OK */
   END /* DO i=1 TO depth*/
   SAY 'There were 'depth' messages on the event queue 'EventQ
   SAY
   /* Stop access to a Queue */
   RXMQVTRACE = ''
   rcc = RXMQV('CLOSE', h2, mqco_none)
   SAY  rcc
 END /* Open OK *
 /* Disconnect from the QM */
 RXMQVTRACE = ''
 rcc = RXMQV('DISC', )
 SAY  rcc
END /* Connect OK */
/* Remove the Interface functions from the Rexx Workspace ... */
RXMQVTRACE = 'TERM'
rcc = RXMQV('TERM', )
SAY  rcc
RETURN
```

Marcin Grabinski
Systems Engineer
SPIN (Poland)

© SPIN 2003

MQ news

Cape Clear Software is now shipping Cape Clear 4, the latest version of its Web Services development, integration, deployment, and management product suite.

Cape Clear 4 includes three integrated products: Studio, Server, and Manager. Studio is said to provide an environment for designing and developing Web Services. Server apparently provides the technology to integrate and deploy large-scale Web Services with diverse back-end systems and technologies such as Java, J2EE, CORBA, Microsoft .NET, IBM WebSphere MQ, as well as packaged applications such as SAP. Manager is claimed to provide tools to control the configuration, deployment, and security of large-scale Web Services deployments.

The company says this latest version delivers new features that include performance and security enhancements, support for JMS and MQ-based Web Services, and new Web Services management capabilities.

For more information contact:
Cape Clear Software, 2929 Campus Drive, Suite 400, San Mateo, CA 94403, USA.
Tel: (888) 227 3439.
Fax: (413) 622 4295.
Web: http://www.capeclear.com

Cape Clear Software, 61 Fitzwilliam Lane, Dublin 2, Ireland.
Tel: +353 1 241 9900.
Fax: +353 1 241 9901.

* * *

Candle Corporation has recently introduced PathWAI, a modular suite of solutions that, claims the company, empowers businesses rapidly to architect, develop, deploy and manage their WebSphere infrastructures.

The PathWAI suite includes architecture, development, and deployment modules, including application performance testing and tuning tools for WebSphere Application Server or WebSphere MQ Integrator (WMQI); and services and training to meet required business performance metrics. The management solutions for testing and monitoring WMQI and WebSphere Application Server are available on Windows NT, Sun, AIX, and OS/390 platforms.

The PathWAI Dashboard module for WebSphere MQ (WMQ) is claimed to provide a single-pane-of-glass view of WMQ systems management activity, including enterprisewide configuration for WMQ. The offering apparently allows users to map the effect of WMQ performance and availability on specific business objectives.

For more information contact:
Candle, 100 N Sepulveda Blvd, El Segundo, CA, 90245, USA.
Tel: (310) 535 3600.
Fax: (310) 727 4287.
Web: http://www.candle.com

Candle, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey, GU16 7ER, UK.
Tel: +44 1276 414 700.
Fax: +44 1276 414 777.

* * *