78

MQ update

December 2005

In this issue

A itc PUBLICATION

Published by

Xephon Inc
PO Box 550547
Dallas, Texas 75355
USA

Phone: 214-340-5690
Fax: 214-341-7081

Editor

Trevor Eddolls
E-mail: trevore@xephon.com

Publisher

Colin Smith
E-mail: info@xephon.com

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.

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs $380.00 in the USA and Canada; £255.00 in the UK; £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 2001 issue, are available separately to subscribers for $33.75 (£22.50) each including postage.

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of $160 (£100 outside North America) per 1000 words and $80 (£50) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of $32 (£20) per 100 lines. To find out more about contributing an article, without any obligation, please 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.

Printed in England.

Sample dead-letter queue handler message flow

The following project illustrates a quick and easy way to develop a sample dead-letter queue handler type of program using WMQI message flow.

First let's look at some background and the message format on the dead-letter queue.

SYSTEM.DEAD.LETTER.QUEUE

From the MQ manual:

Each queue manager should have a local queue to be used as a dead-letter queue so that messages that cannot be delivered to their correct destination can be stored for later retrieval. You must explicitly tell the queue manager about the dead-letter queue. You can do this by specifying a dead-letter queue on the ***crtmqm*** *command or you can use the ALTER QMGR command to specify one later. You must also define the dead-letter queue before it can be used.*

A sample dead-letter queue called SYSTEM.DEAD.LETTER.QUEUE is supplied with the product. This queue is automatically created when you run the sample. You can modify this definition, if required. There is no need to rename it.

"A dead-letter queue has no special requirements except that

- *It must be a local queue.*
- *Its MAXMSGL (maximum message length) attribute must enable the queue to accommodate the largest messages that the queue manager has to handle plus the size of the dead-letter header (MQDLH).*

MQSeries provides a dead-letter queue handler that allows you to specify how messages found on a dead-letter queue are to be processed or removed.

MESSAGE FORMAT ON THE SYSTEM.DEAD.LETTER.QUEUE

Messages can be put on the DLQ by queue managers, by message channel agents (MCAs), and by applications. All messages on the DLQ should be prefixed with a dead-letter header structure, MQDLH. Messages put on the DLQ by a queue manager or by a message channel agent always have an MQDLH; applications putting messages on the DLQ are strongly recommended to supply an MQDLH. The Reason field of the MQDLH structure contains a reason code that identifies why the message is on the DLQ.

The MQDLH structure

In order to process a message in the dead-letter queue, we must know what the structure of the MQDLH header is like. A detailed description of the MQDLH can be found in Chapter 6

Field name	Name of constant	Value of constant
StrucId	MQDLH_STRUC_ID	'DLHb'
Version	MQDLH_VERSION_1	1
Reason	MQRC_NONE	0
DestQName	None	Null string or blanks
DestQMgrName	None	Null string or blanks
Encoding	None	0
CodedCharSetId	MQCCSI_UNDEFINED	0
Format	MQFMT_NONE	Blanks
PutApplType	None	0
PutApplName	None	Null string or blanks
PutDate	None	Null string or blanks
PutTime	None	Null string or blanks

Figure 1: MQDLH description

of the *MQ Application Programming Reference* – see Figure 1.

The DLQ_Handler message flow

The processing logic of the message flow is based on the settings in the MQDLH header, particularly the *Reason* field. This field tells us why the message got sent to the dead-letter queue in the first place. By interrogating this *Reason* field, we can determine the action on the message. In this example, we will resend all the messages in the dead-letter queue whose *Reason* code is mqrc=2503, assuming that the queue-full problem had been fixed.

The message coming into the message flow may have the following structure:

```
InputRoot - Properties
            MQMD
            MQDLH
            MQHRF2 or InputBody
```

Because we don't exactly know whether the message will contain headers other than the MQDLH, we need to parse the message as a BLOB – see Figure 2.

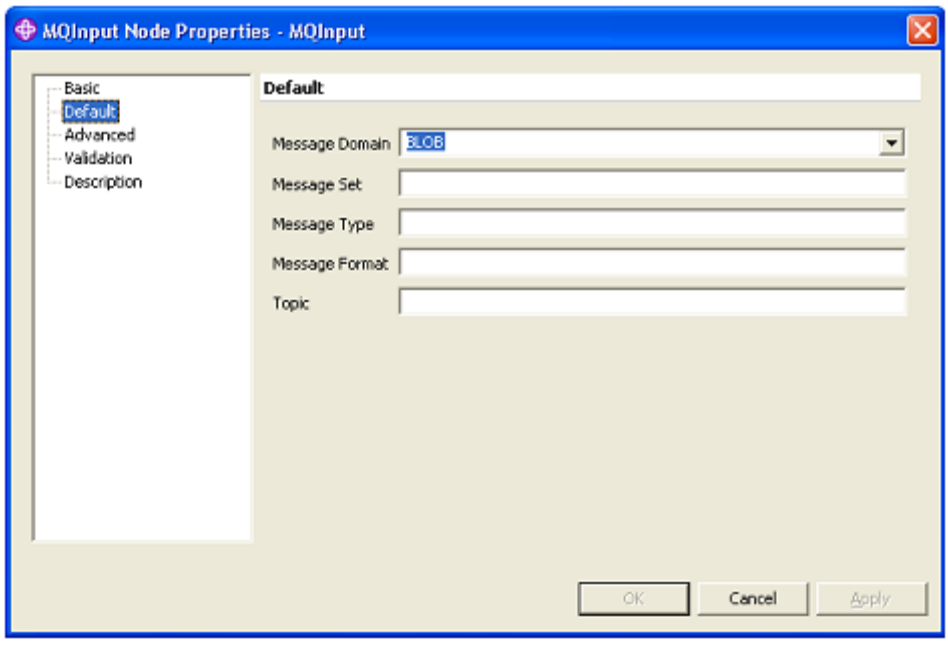

Figure 2: Setting the default to BLOB

In the next compute node, we will first copy the message header to the output, and then we will parse the MQDLH header into the environment variables for access:

```
CREATE COMPUTE MODULE DLQ_handler_Compute
      CREATE FUNCTION Main() RETURNS BOOLEAN
      BEGIN
      CALL CopyMessageHeaders();
SET wholeMsgBLOB = InputRoot.BLOB.BLOB;

      -- the first 172 characters which is MQDLH
      -- Parse the MQDLH structure in the Environement Variable

      SET tempBLOB = substring(wholeMsgBLOB from 1 for 172);
Create LASTCHILD of Environment.Variables.MQDLHOut DOMAIN('MQDLH')
Parse(tempBLOB, InputRoot.MQMD.Encoding, InputRoot.MQMD.CodedCharSetId);
```

Then we will check for the Reason code that is MQRC_Q_FULL. If the Reason code is Q_FULL, we will write the message back to the queue specified by the *DestQName* and *DestQMgrName*. Otherwise, we will write the message out to a local application dead-letter queue ADLQ.OUT for further action.

```
-- check for the Q Full condition
If Environment.Variables.MQDLHOut.MQDLH.Reason = MQRC_Q_FULL then
-- Process Q_FULL

Else
CALL CopyEntireMessage();
      Set
OutputLocalEnvironment.Destination.MQ.DestinationData.queueName =
'ADLQ_OUT';
End If; -- Done Sample DEAD LETTER HANDLER
```

Here is the processing logic for Q_FULL:

```
SET beginMsg = 173;

-- check what is following the MQDLH
-- is it the MQHRF2 (for jms message)
If Environment.Variables.MQDLHOut.MQDLH.Format = 'MQHRF2  ' then
-- restore the MQHRF2 header from the BLOB
-- first find out the total length of MQHRF2 header
SET bitHolder = X'00000000';
-- The length field is at the 8th bytes of the MQHRF2 header
Set BigEn = bitHolder || SUBSTRING(wholeMsgBLOB FROM 181 FOR 4);
Set MQHRF2HeaderLength = CAST(BigEn AS INT CCSID I
nputRoot.MQMD.CodedCharSetId);
-- If the platform is Windows, we need to convert the significant bytes
```

```
for the actual length
      If Environment.Variables.MQDLHOut.MQDLH.Encoding = 546 then
            -- Windows
Set BigEn = SUBSTRING(wholeMsgBLOB FROM 181 FOR 4) || bitHolder;
Set MQHRF2HeaderLength = CAST(ConvertEndian(BigEn) AS INT CCSID
InputRoot.MQMD.CodedCharSetId);
      End If;
-- with this length, we can parse the MQHRF2 header to the environment
tree
Set tempBLOB = SUBSTRING(wholeMsgBLOB from beginMsg FOR
MQHRF2HeaderLength);

CREATE LASTCHILD OF Environment.Variables.MQRFH2Out DOMAIN('MQRFH2')
PARSE(tempBLOB,  Environment.Variables.MQDLHOut.MQDLH.Encoding,
Environment.Variables.MQDLHOut.MQDLH.CodedCharSetId);

      -- Override the MQMD.Format field first
      Set OutputRoot.MQMD.Format = 'MQHRF2  ';
      -- write the MQHRF2 header
      Set OutputRoot.MQRFH2 = Environment.Variables.MQRFH2Out.MQRFH2;
                        -- write the rest of the message as BLOB out
      Set beginMsg = beginMsg + MQHRF2HeaderLength;
      Set tempBLOB = Substring(wholeMsgBLOB from beginMsg);
      Set OutputRoot.BLOB.BLOB = tempBLOB;
      -- set the Destination queue and queue manager accordingly
Set OutputLocalEnvironment.Destination.MQ.DestinationData.queueName =
Environment.Variables.MQDLHOut.MQDLH.DestQName;
Set OutputLocalEnvironment.Destination.MQ.DestinationData.queueMgrName =
Environment.Variables.MQDLHOut.MQDLH.DestQMgrName;

Else
      -- no MQHRf2 header
      -- just write message out as BLOB
      Set OutputRoot.MQMD.Format = 'MQSTR   ';
      -- set the Destination queue and queue manager accordingly
SET OutputRoot.BLOB.BLOB = substring(InputRoot.BLOB.BLOB from 173);
Set OutputLocalEnvironment.Destination.MQ.DestinationData.queueName =
Environment.Variables.MQDLHOut.MQDLH.DestQName;
Set
OutputLocalEnvironment.Destination.MQ.DestinationData.queueManagerName =
Environment.Variables.MQDLHOut.MQDLH.DestQMgrName;

End If;
```

The procedure to convert Little and Big Endian is as follows:

```
CREATE FUNCTION ConvertEndian (Value BLOB)
            RETURNS BLOB BEGIN
            DECLARE LittleEn BLOB;
            DECLARE FieldLength INT;
```

```
            SET FieldLength = LENGTH(Value);
            SET LittleEn = substring(Value from FieldLength for 1);
            SET FieldLength = FieldLength - 1;
            WHILE FieldLength >= 1  DO
SET LittleEn = LittleEn || substring(Value from FieldLength for 1);
                  SET FieldLength = FieldLength - 1;
            END WHILE;
            SET Value = LittleEn;
            RETURN Value;
      END;
```

Somehow, when we restore the message, the mcd>MSD is not set correctly, we need to add more filter logic to reset the message domains – see Figure 3.

```
If upper(Environment.Variables.MQRFH2Out.MQRFH2.mcd.Msd) = 'JMS_TEXT'
then
      RETURN TRUE;
Else
RETURN FALSE;
End If;
```

Figure 4 is a snapshot of the sample message flow.

Downloadable from the Xephon Web site are a sample dlq

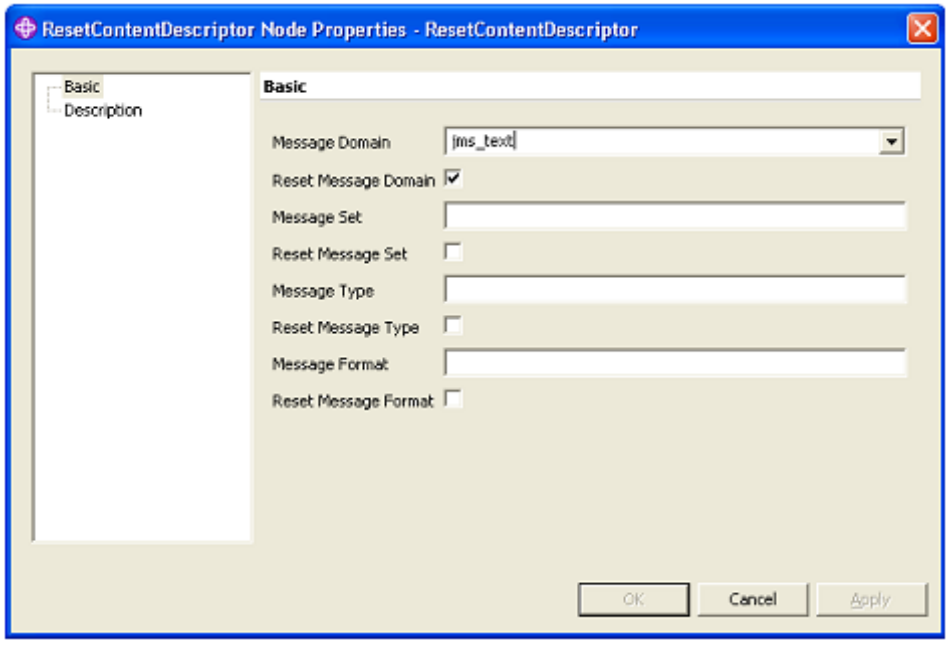

Figure 3: Message domain

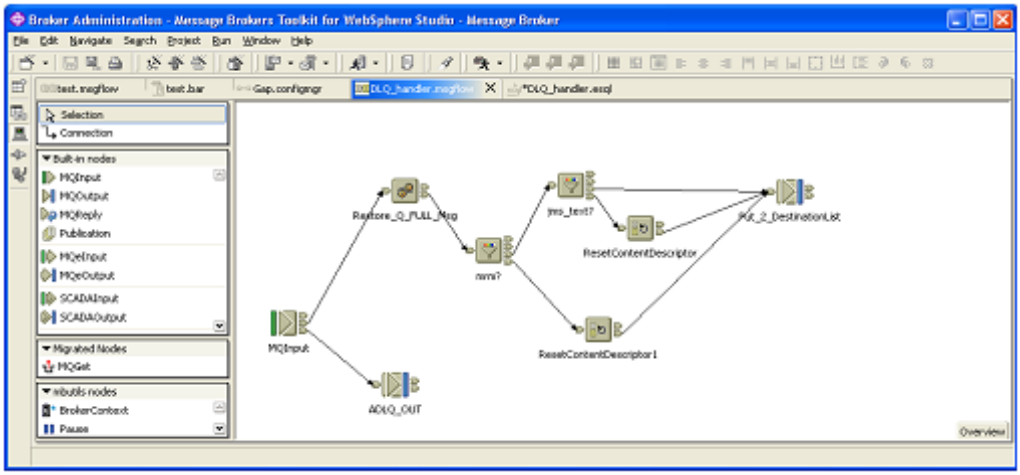

Figure 4: Message flow

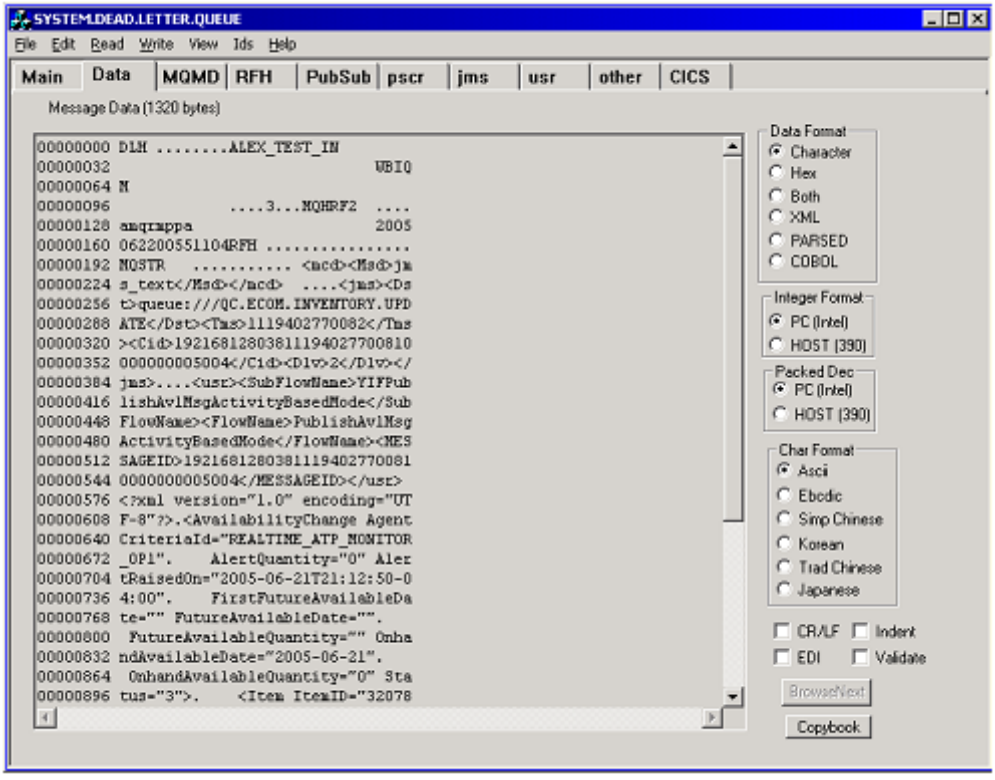

Figure 5: Snapshot of dlqmsg1 message

message, the message flow project, and the bar file. The files to download are www.xephon.com/extras/0512Dlqmsg1, www.xephon.com/extras/0512DLQ_Handler.zip, and www.xephon.com/extras/0512DLQ_Handler.bar.

To run the sample, download the files, create a local queue ADLQ.OUT and ALEX_TEST_IN. Put the sample dlq message (dlqmsg1) in the SYSTEM.DEAD.LETTER.QUEUE with **rfhutil** and deploy the bar file.

Figure 5 is a snapshot of the dlqmsg1 message.

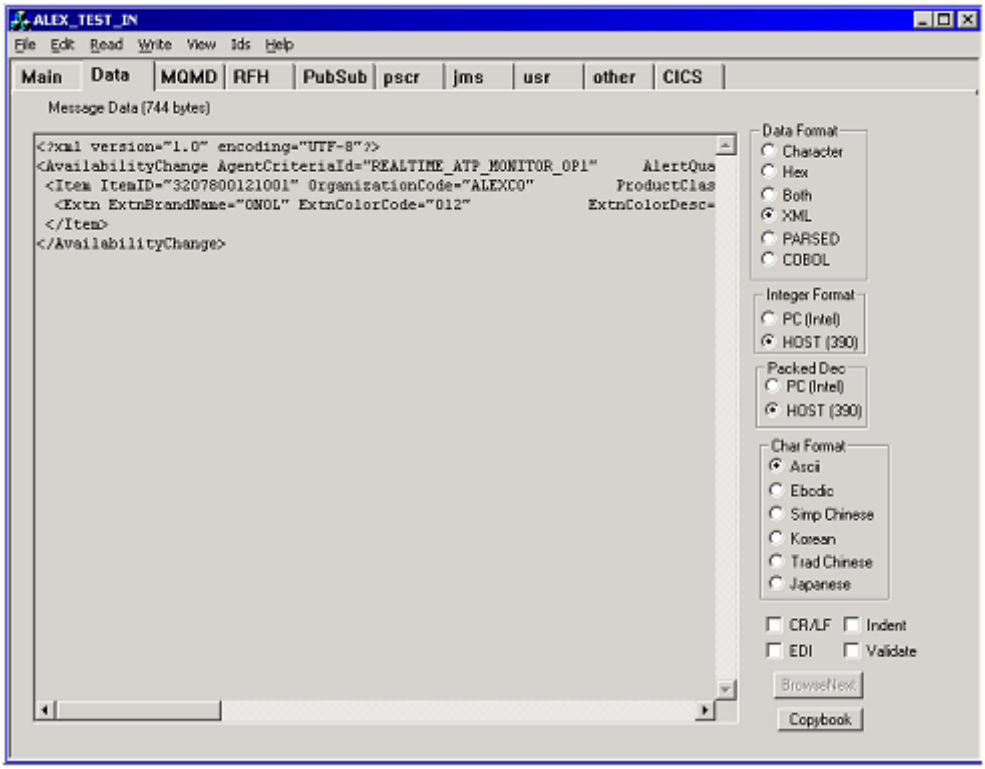

Figure 6: Restored/re-sent message

The restored/re-sent message is shown in Figure 6.

Have fun.

Alex Au
IT Architect
IBM Global Services (USA)

Communication with legacy applications

INTRODUCTION

The aim of this article is to show how to communicate with legacy applications from applications on other platforms. The example used is an implementation of intercommunication between a .NET application and a COBOL application using MQSeries and CICS DPL bridge.

OVERVIEW

A lot of organizations have invested years of effort in homegrown applications on IBM mainframes. Business rules and processes are defined on them, staff have been trained on them, the systems are valuable, and they work. The natural wish of such organizations is to extend the useful life of existing legacy applications. This can be done by exposing the functionality of these applications to new applications on diverse platforms, including the Web and mobile devices. And, what is more important, this task should be done without redevelopment of the existing applications.

Although many companies provide sophisticated solutions to this problem, no-one can suggest a universal one. Thus, the solution implemented in this article is no panacea; it just shows a simple approach to exposing some legacy functionality to .NET platforms without using additional middle-level software like SNA, HIS, or BizTalk Servers, etc.

WHY MQSERIES AND CICS?

Many legacy applications were developed using transaction monitoring and control systems such as IBM's CICS. CICS gives developers a single entry point to execute an atomic transaction that will be managed by the host, but can participate in a broader transaction initiated by a controlling host.

The MQSeries-CICS bridge is the component of MQSeries for OS/390 that allows direct access from MQSeries applications to applications on a CICS system.

So, the answer is obvious – this is the simplest way to communicate with legacy applications.

HOW DOES THIS WORK?

The test .NET application starts a legacy application by sending a structured message via MQSeries to the CICS bridge request queue and retrieves a response from a reply queue.

Here is the sequence of events that happen when you put such a structured message on the request queue:

1 A message with a request is transmitted to the request queue on a mainframe.

2 The CICS bridge monitor task, which is constantly browsing the queue, recognizes it.

3 After some authentication checks, the CICS DPL bridge task is started with the appropriate authority.

4 This task removes the message from the request queue.

5 It builds a COMMAREA from the data in the message and issues an **EXEC CICS LINK** for the program requested in the message.

6 The program returns the response in the COMMAREA used by the request.

7 The CICS DPL bridge task reads the COMMAREA, creates a response message (whether it is a normal response from an application or a system error), and puts it on the reply-to queue specified in the request message.

8 The CICS DPL bridge task ends.

Please note several important things:

- The bridge monitor task recognizes requests by their CorrelationId – it should be equal to AMQ!NEW_SESSION_CORRELID in the content coding of the message.
- It's possible to provide a user name and password for the CICS DPL bridge task from a client application through the request MQ message and control header.
- The *ReplayToQueue* and *ReplayToQueueManager* fields of the request MQ message should contain the correct queue and queue manager names.
- The requested COMMAREA should be big enough to contain any requests or responses.

THE MESSAGE STRUCTURE

The structure of the request message differs slightly depending on the legacy application. There are four types of request message structure:

1 An application running a single DPL program uses the default processing options, and does not send or receive COMMAREA data. A message with this structure contains only the legacy program name (eight bytes in length).

2 An application running a single DPL program uses the default processing options, and sends and receives COMMAREA data. A message with this structure has a program name followed by the user data (COMMAREA).

3 An application that runs one or more DPL programs within a unit of work, or needs specific authorization to run the program, but does not send or receive COMMAREA data. A message with this structure contains a control header (a system header, which is described below) and the program name.

4 An application that invokes one or more DPL programs within a unit of work, or needs specific authorization to run the program, and sends and receives COMMAREA data.

A message with this structure contains the control header, the program name, and the user data (COMMAREA).

The response message structure can be one of the following types:

- The normal response structure, which contains the control header, the program name, and, optionally, the output COMMAREA data.
- If a bridge task running a DPL program ends abnormally, the response is returned to the reply queue. The structure of such a message is the control header followed by an error message indicating the error type.

Pay attention to the following details:

- When you want to send only a program name, and no COMMAREA data, the program name must be eight characters long. It must not be padded to the right with spaces; if it is, the bridge will report a COMMAREA negative length error.
- When you want to send COMMAREA data, you must then pad the program name with spaces to the right to give a total length of eight characters.

THE CONTROL HEADER

The messages for CICS DPL bridge can contain a standard control header, MQCIH. This header contains additional information for CICS bridge that helps the bridge to start a legacy application properly. If MQCIH header is present in the message, the format of the MQ message should be MQFMT_CICS. Take a look at the MQCIH header structure:

Offset:	Type:	Length:	Name:
0000	CHARACTER	4	MQCIH-STRUCID
0004	FULLWORD	4	MQCIH-VERSION
0008	FULLWORD	4	MQCIH-STRUCLENGTH
000C	BINARY	8	reserved
0014	CHARACTER	8	MQCIH-FORMAT
001C	BINARY	4	reserved

```
0020            BINARY          4           MQCIH-RETURNCODE
0024            BINARY          4           MQCIH-COMPCODE
0028            BINARY          4           MQCIH-REASON
002C            BINARY          4           MQCIH-UOWCONTROL
0030            FULLWORD        4           MQCIH-GETWAITINTERVAL
0034            BINARY          4           MQCIH-LINKTYPE
0038            BINARY          4           MQCIH-OUTPUTDATALENGTH
003C            FULLWORD        4           MQCIH-FACILITYKEEPTIME
0040            FULLWORD        4           MQCIH-ADSDESCRIPTOR
0044            FULLWORD        4           MQCIH-CONVERSATIONALTASK
0048            FULLWORD        4           MQCIH-TASKENDSTATUS
004C            CHARACTER       8           MQCIH-FACILITY
0054            CHARACTER       4           MQCIH-FUNCTION
0058            CHARACTER       4           MQCIH-ABENDCODE
005C            CHARACTER       8           MQCIH-AUTHENTICATOR
0064            CHARACTER       8           reserved
006C            CHARACTER       8           MQCIH-REPLYTOFORMAT
0074            CHARACTER       4           MQCIH-REMOTESYSID
0078            CHARACTER       4           MQCIH-REMOTETRANSID
007C            CHARACTER       4           MQCIH-TRANSACTIONID
0080            CHARACTER       4           MQCIH-FACILITYLIKE
0084            CHARACTER       4           MQCIH-ATTENTIONID
0088            CHARACTER       4           MQCIH-STARTCODE
008C            CHARACTER       4           MQCIH-CANCELCODE
0090            CHARACTER       4           MQCIH-NEXTTRANSACTIONID
0094            CHARACTER      16           reserved
00A4            FULLWORD        4           MQCIH-CURSORPOSITION
00A8            FULLWORD        4           MQCIH-ERROROFFSET
00AC            FULLWORD        4           MQCIH-INPUTITEM
00B0            BINARY          4           reserved
```

Actually, in order to start a program via CICS DPL bridge, it's enough to use just a few fields of this header, ie MQCIH-FORMAT, MQCIH-LINKTYPE, MQCIH-AUTHENTICATOR, MQCIH-OUTPUTDATALENGTH, MQCIH-RETURNCODE, MQCIH-COMPCODE, MQCIH-REASON, and, optionally, MQCIH-UOWCONTROL and MQCIH-TRANSACTIONID.

THE SOLUTION

Now, let's overview the solution. First of all, it is implemented as a Microsoft Visual Studio .NET 2003 solution, and it consists of four projects. One of them is a demo project, which demonstrates how to use the bridge component through MQSeries from a .NET console applications. So, let's look at the solution's file inventory, with the file's location followed by a description:

- */Demo* – the demo project.
- *./COBOL/CALC.cob* – defines the COBOL copybook, which is used by the legacy application.
- *./App.config* – the demo configuration file. It contains MQ settings, content encoding, etc.
- *./AssemblyInfo.cs* – the project information file.
- *./Calc.cs* – defines class, which can be serialized and deserialized to/from the *CALC.cob* copybook.
- *./CicsDplBridgeDemo.cs* – defines the main entry point for the demo.
- */NesterovskyBros.Common* – this library contains the common interface for serializable classes and auxiliary classes.
- *./AssemblyInfo.cs* – the library information file.
- *./HexEncoder.cs* – defines a byte array to a hex string, and a hex string to a byte array converter.
- *./ICobolSerializableData.cs* – defines a common interface for each class that can be serialized/deserialized to/from COBOL records.
- *./MemoryDump.cs* – defines an auxiliary class for printing memory dumps.
- */NesterovskyBros.MQ.CicsDplBridge* – this library contains a class that implements communication to CICS DPL bridge through MQSeries.
- *./AssemblyInfo.cs* – the library information file.
- *./Bridge.cs* – defines a class that implements a common communication pattern with CICS DPL bridge through MQSeries.
- *NesterovskyBros.MQ.Connector* – this library contains classes for general MQSeries communication.

- *./AssemblyInfo.cs* – the library information file.
- *./IBM MQSeries Library for .NET/amqmdnet.dll* – the IBM .NET library for communication with MQSeries.
- *./ConnectorProperties.cs* – defines MQ connector properties.
- *./Installer.cs* – defines the installer for COM+ components.
- *./Mqcih.cs* – defines the control header class (MQCIH). The class implements the ICobolSerializableData interface, and it is an example of how to serialize instances to COBOL records.
- *./MQConnector.cs* – defines the MQ connector class. It's implemented as a COM+ component in order to allow it to run under a different user identity. This class implements request-response, one-way send and receive communication patterns with MQSeries.
- *./MQConstants.cs* – some MQ-related constants that were not included in the IBM.WMQ.MQC class.
- *./SerializableMQMessage.cs* – defines a serializable wrapper for a standard MQ message.

The MQ connector class is implemented as a COM+ server-activated component in order to allow the MQSeries library to be used under a specified user identity. Such an approach allows the use of MQSeries not only from stand-alone applications, but also from ASP.NET applications (Web applications and Web services). Note that SerializableMQMessage wraps an IBM MQMessage class; this is done because the original MQMessage class cannot be serialized to stream. There is another tricky thing – use of the ConnectorProperties class in order to set MQ properties (MQ queue manager name, input and output queues, etc) to MQ connectors. Since each method call is a round-trip to a server component, this approach allows an improvement in performance of the COM+ component.

You can get more detailed information about the solution implementation from the source files attached to this article.

THE DEMO

The demo was conceived as a simple client for a real (also simple) COBOL application, which is working on mainframe under CICS' control. The legacy application accepts as input a COMMAREA, which is defined in the */COBOL/CALC.cob* copybook. In this copybook, CALC-REQ is an input structure, and CALC-RESP is an output one. In a few words the legacy application does the following:

1. It receives a COMMAREA and performs one of four arithmetical operations (add, subtract, divide, or multiply) followed by two decimal arguments.
2. When the operation finishes successfully, the legacy application returns CALC-RESP structure with RESP-CODE equal to 0, otherwise it returns 1.
   - When RESP-CODE is 0, the RESULT field contains the answer.
   - When RESP-CODE is 1, the ERR-CODE and ERR-MSG fields contain an error code and an appropriate error message.

The *App.config* (*Demo.exe.config* after compilation) file contains settings for the MQ connector component, a legacy application name, content encoding, etc.

In order to execute the demo against the real legacy application, do the following:

1. Open the solution in Microsoft Visual Studio .NET 2003.
2. Modify the settings in the *App.config* file to fit your environment.
3. Do **Rebuild Solution**.
4. Start *Demo.exe*.

Note: by default the input and output XML file names will be taken from the configuration file.

The first time, when the input XML file with the serialized CALC-REQ structure doesn't exist, it will be created automatically. The next time you can modify and use the existing file.

The *CicsDplBridgeDemo.cs* file is the main class of the demo project. Let's look at its Run() method:

```
private static int Run(string[] args)
{
      /* some content was skipped */
      try
      {
        // set MQ settings
        ConnectorProperties properties = new ConnectorProperties();
        /* some content was skipped */
        // create an CicsDplBridge instance
        Bridge bridge = new Bridge(properties, encoding,
userName, password, useTransaction);
        // read XML input file and deserialize it in CalcReq instance.
        XmlSerializer serializer = new XmlSerializer(typeof(CalcReq));
        CalcReq request = null;
        using (Stream stream = new FileStream(inputFile, FileMode.Open,
FileAccess.Read, FileShare.Read))
        {
          request = serializer.Deserialize(stream) as CalcReq;
        }
        // create an empty response instance
        ICobolSerializableData response = new CalcResp();
        // make a call of a legacy application
        bridge.Invoke(applicationId, request, ref response);
        if (response != null)
        {
          // serialize CALC-RESP structure to XML file
          serializer = new XmlSerializer(typeof(CalcResp));
          using (Stream stream = new FileStream(outputFile,
FileMode.CreateNew))
          {
            serializer.Serialize(stream, response);
          }
    /* some content was skipped */
        }
        else
        {
          Console.WriteLine(
"There is no response from legacy application \"{0}\".",
```

```
applicationId);
        }
      }
      catch (Exception e)
      {
        Error(e, "Error: ");
        return 1;
      }
      return 0;
}
```

This method shows how to create a ConnectorProperties instance and fill it in, create a bridge instance, read from the XML file CALC-REQ request structure, invoke a bridge method (to make a call to the legacy application), and write a CALC-RESP response to an XML file.

CONCLUSION

This article demonstrates how easy it is to use a legacy application from a .NET console application. Even the creation of a Web service that, in turn, calls a legacy application, will take about half an hour to implement using the components from this solution. So, we see how contemporary technologies extend the life of a legacy application without making any changes to it.

All the files mentioned in this article are available to download in one zip file from www.xephon.com/extras/0512sources.zip.

Arthur Nesterovsky
Software Developer (Israel)

A Java toolkit for WebSphere MQ

If you are a WebSphere MQSeries (WMQ) systems administrator and reading this article, I have to assume it is because you are one of the following:

- An incurable insomniac and have been driven to desperate measures.

- Need to learn more about WMQ and/or improve your understanding of one of IBM's hottest technologies.
- Tired of trying to prove to your application development client community that their inability to send messages across your network is the result of their application problems and not your infrastructure design/ implementation.
- Sadly sadomasochistic with nothing better to do than try to comprehend the incomprehensible.

Regardless of why you are reading this, the fact is that you are about to embark on a journey that would make the most hardened and cynical adventurer think twice. Most of us have become WMQ systems administrators solely to avoid the requirement to deal with programming languages, and the client communities that try to control anyone who attempts to write code for a living.

THE OBLIGATORY HISTORICAL REMINISCENCES

If you have been in this industry as long as I have (I can still remember watching my mother hardwire the front panel of the Iliac IV at the University of Illinois), then you probably have not escaped the requirement to spend some time as a programmer. My first paying job as a programmer was with the University of Kentucky Department of Psychology. I started coding FORTRAN, and then rapidly moved into the procedural languages like GPSS and SPS. This unfortunate upbringing culminated in my writing Basic Assembler Language programs for IBM in Poughkeepsie.

However, I rapidly moved from programming into database administration, and then progressed through the years to the day when I became an independent consultant, working with WebSphere Business Integration (WBI). Long gone were the days when I would spend hours writing and debugging program code.

As a WBI systems administrator, I have progressed from

creating WMQ objects that others have designed to generating middleware infrastructure designs of my own. Implementing hub and spoke architectures, using multiple queue managers on multiple platforms, as well as publish-and-subscribe environments using WBI Message Broker have enabled me to create some fairly sophisticated environments. Those environments are active today in some of the world's leading financial, technical, and industrial organizations.

However, as I have progressed along the path of my technical career, I discovered a need to be able to do more than design and implement the infrastructures themselves. I have found a need to be a user of those infrastructure components from time to time.

THE PREMISE BEHIND THIS SERIES OF ARTICLES

As WBI environments become more complex, it becomes necessary to enter messages into the front, and watch them fall out of the back of the pipes that we are building. At first, I used the standard WMQ utilities like **amqsput** to introduce messages into queues and pass them along paths, proving that I have correctly defined paths from A to B.

Then, users began to ask more pointed questions like, "What about moving multiple messages in a batch from A to B?" and "I need to be able to read my messages from B, but my application is not ready yet to process those messages".

As the requirements of my consumers became more sophisticated, I progressed to using freeware tools and the IBM Support Packs like MA01 (The Q program) and RFHUTIL, as well as tools from CapitalWare and others. These tools enabled me to move messages more easily, and in a greater variety of configurations, but my clients wanted ever more facilities. Still, I was able to meet most requests with one of the tools that became a standard part of my kit bag.

Then, as I moved more into the complex design and application integration roles, which pay the larger stipends, my clients began to make comments like:

- “My application can’t pass messages into WMQ.”
- “I write my messages, but they don’t end up where they should be.”
- “My message flows are not triggering when messages arrive on the queue.”
- “My message sets don’t transform as I expected them to.”
- and so on…

Not surprisingly, in each case the problem is always with WBI and never with the application programs themselves. Even as you would prove that messages, created and written with the utilities, moved as expected, the applications teams would always insist, “Well, you aren’t using programs, you are using the WMQ utilities and they are not as sophisticated as my application…” or words to that effect.

So, I went back to basics, and figured that I would learn one of the ‘new’ programming languages and show those applications guys that it really wasn’t my infrastructure that had the problem.

They say that, ‘the road to Hell is paved with good intentions’. Well, I didn’t really intend to become a source of utility programs for the benefit of my customers. All I really wanted to do was be able to prove that connectivity problems were not my fault. However, I have started to find that every client site that I work at has the same basic set of problems, and that a kit bag of basic WBI programs can drastically reduce the time to resolution for a communications problem.

CAVEAT(S)

Starting in this issue and, hopefully, continuing in future articles, I will be publishing some of the basic Java utility applications that I have created for use in a WBI environment. These applications have been created to be freely distributed, and are available on my corporate Web site, distributed as a Java Archive (JAR) file. Xephon, in cooperation with 3-INGs

Limited, will be incorporating the Java programs into a JAR file that will be accessible to their client base at www.xephon.com/extras/0512xephon.jar.

Let me state, categorically, I am not the world's best Java programmer. Some of you will look at my code and wonder who in their right mind would create programs like these. However, I can say that the programs themselves have all been extensively used in real client environments, and work as designed.

The programs that I will be publishing in this series are what I call 'integration' programs. They do not use a GUI (Graphical User Interface), but are instead 'command line' programs that accomplish a specific purpose. Since the programs are designed to work with WMQ, their content can be taken and moved into other applications and reused in that fashion, but I personally wouldn't know an IDE or a widget if they came and slapped me in the face.

If anyone would like to take the time to recommend improvements to these programs, I will be happy to hear your comments and suggestions. You can contact me at Aaron.Cain@3ings.com, with your thoughts and ideas, and I will see what can be incorporated into both the 3-INGs and Xephon program libraries.

LET'S GET STARTED

In order to run the Java applications in this series, you will need to ensure that you have a working Java environment on the workstation or server that you wish to use. Since WBI runs on a variety of platforms, your particular requirements may be unique. However, most environments can be broken down into the following categories:

- Mainframes
- Windows environments
- Unix environments.

I will not be addressing the mainframe environments (z/OS or iSeries) in the context of these articles, though I don't rule out the possibility in the future.

Windows

Windows environments are usually fairly consistent, but I do make some assumptions:

- You are using Windows 32-bit environments.
- You are running Windows 2000 or above.

I am not saying that the programs won't run in Windows environments other than these, but I have not tested them extensively in environments that do not match these basics.

Unix

Unix environments are fairly consistent in the way in which they execute Java programs, but can vary widely depending on the flavour you are running. The programs have been used extensively in Linux (RedHat and Suse), AIX (4.*x* and above), and Solaris (2.6 and above).

As above, the programs will probably run in other environment, but you may need to make some adjustments.

Java

The last piece that has to be in place for these programs to work is Java. The programs have been developed using the Java Software Development Kit 1.4.2, and have been run successfully using Java Runtime Environment 1.3.1 and above. Again, if you have a version of Java pre-dating these environments, you may find some program inconsistencies.

SETTING UP THE SERVER ENVIRONMENT

Running a Java program requires you to identify two components to your server:

- The location of the Java Runtime Environment
- The location of the Java program.

To do this, you need to have a way of changing the environment variables on the server you will be working with.

The Java program that is listed below is contained in the 0512xephon.jar file (which can be downloaded from www.xephon.com/extras) or the 3INGs.jar file (which can be downloaded from www.3ings.com/3INGs_Mentoring.htm).

Windows

In order to use the Java program on a Windows server, you will need to copy the JAR file to a directory. For all our development and testing in our laboratory environment, we use:

```
C:\Java
```

However, you can choose any location you want.

To run the programs on a Windows workstation or server, you will need to start a command window (the old DOS prompt). To do this, you can execute 'cmd' from *Run* in the *Start* menu. This will open a command prompt dialogue box. To test whether you have Java installed on the Windows server, run the command:

```
java -version
```

You should see output similar to:

```
C:\Documents and Settings\Aaron Cain>java -version
java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_02-b09)
Java HotSpot(TM) Client VM (build 1.5.0_02-b09, mixed mode, sharing)
```

If Java has not been installed on your workstation, you can download it from *http://java.sun.com/j2se/1.5.0/download.jsp*. The programs that will be included in this series have been developed using the Java Development Kit 1.4.2, but they have been used commercially on JDK Versions from 1.3.1 to 1.5. Other versions may be compatible, but it will be up to you to verify the compatibility of your version of Java.

You can list the server environment variables, by running the Windows command **set**:

```
C:\Java>set
.   .   .
.   .   .
.   .   .
CLASSPATH=C:\Program Files\IBM\WebSphere
MQ\Java\lib\providerutil.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\com.ibm.mqjms.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\ldap.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\jta.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\jndi.jar;C:\Program Files\IBM\WebSphere MQ\
Java\lib\jms.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\connector.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\fscontext.jar;C:\Program Files\IBM\WebSphere
MQ\Java\lib\com.ibm.mq.jar;.;C:\PROGRA~1\IBM\SQLLIB\java\db2java.zip;C:\PROGRA
~1\IBM\SQLLIB\java\db2jcc.jar;C:\PROGRA~1\IBM\SQLLIB\java\sqlj.zip;C:\PROGRA~1\I
BM\SQLLIB\java\db2jcc_license_cisuz.jar;C:\PROGRA~1\IBM\SQLLIB\java\db2jcc_licen
se_cu.jar;C:\PROGRA~1\IBM\SQLLIB\bin;C:\PROGRA~1\IBM\SQLLIB\tools\db2XTrigger.ja
r;C:\PROGRA~1\IBM\SQLLIB\java\common.jar
.   .   .
.   .   .
.   .   .
```

The environment variable that you need to be concerned with is CLASSPATH. The CLASSPATH has to include the Java program(s) along with the WMQ Java support libraries.

To set the CLASSPATH variable, you can concatenate the directory paths to the various WMQ JAR files manually, or you can use the following Windows *JSetLt.bat* file after ensuring that the directory path names for each of the components in the script are corrected for your server environment:

```
C:\Java>type JSetLT.bat
@ECHO OFF
SET JAVA_HOME=C:\Program Files\IBM\WebSphere MQ\Java
SET PATH=%PATH%;C:\j2sdk1.4.2_04\bin
SET PATH=%PATH%;C:\Program Files\IBM\WebSphere MQ\Java\lib
SET LD_LIBRARY_PATH=C:\j2sdk1.4.2_04\lib;%JAVA_HOME%\lib
SET CLASSPATH=%CLASSPATH%;.;C:\j2sdk1.4.2_04\lib;C:\Java
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\dt.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\htmlconverter.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\com.ibm.mq.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\com.ibm.mqbind.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\com.ibm.mqjms.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\com.ibm.mq.pcf.jar
```

```
SET  CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\connector.jar
SET  CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\jta.jar
SET  CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
SET  CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib
SET CLASSPATH=%CLASSPATH%;C:\Program Files\IBM\WAS\jmxc.jar
SET  CLASSPATH=%CLASSPATH%;C:\Program Files\IBM\WAS\admin.jar
SET  CLASSPATH=%CLASSPATH%;C:\Program Files\IBM\WAS\wsexception.jar
SET  CLASSPATH=%CLASSPATH%;C:\Java\3INGs.jar
```

Note: the *JSetLT.bat* file is also included in the Xephon.jar and 3INGs.jar files.

Once you have executed the *JSetLT.bat* file from the command line you will be ready to run the programs in the JAR file from the command window.

When you close the command window, you will lose the customization to the CLASSPATH variable that was made by the *JSetLT.bat* file. Each time you open a command window to run Java programs from, you must run the batch file again to make the changes to the environment variables. Alternatively, you can customize the server environment variables with the libraries, and these will then be available to all users of the environment.

Unix

In order to use the Java program on a Unix server, you will need to copy the JAR file to a directory. In our laboratory environment, for all of our development and testing, we use a directory off the user's home directory named *WMQ_tools*:

```
caina:/home/caina/WMQ_tools> ls -al
total 1232
drwxr-xr-x   3 caina    staff          4096 06 Oct 16:26 .
drwxr-xr-x  11 caina    staff          4096 10 Oct 10:53 ..
-rw-r--r--   1 caina    staff        112805 29 Sep 17:48 3INGs.jar
-rwxr-xr-x   1 caina    staff        376387 22 Sep 09:08 q
-rwxr-xr-x   1 caina    staff         86256 01 Oct 2004  saveqmgr.aix
```

However, you can specify any location that you require. The programs run from the command line in the Unix environment.

To test whether you have Java installed on the Unix server, run the command:

```
java -version
```

You should see output similar to:

```
caina:/home/caina/WMQ_tools> java -version java version "1.4.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1) Classic
VM (build 1.4.1, J2RE 1.4.1 IBM AIX build ca1411ifx-20040810
(141SR3) (JIT enabled: jitc))
```

If Java has not been installed on your workstation, as outlined above, you can download it from *http://java.sun.com/j2se/1.5.0/download.jsp*. You can list the server environment variables by running the Unix command **env**:

```
caina:/home/caina/WMQ_tools> env
_=/usr/bin/env
LANG=en_GB
.  .  .
.  .  .
.  .  .
CLASSPATH=/usr/mqm/java/lib/com.ibm.mq.jar:/usr/mqm/java/lib/
com.ibm.mqbind.jar:/usr/mqm/java/lib/connector.jar:.:/home/caina/
WMQ_tools/3INGs.jar
.  .  .
.  .  .
.  .  .
```

The environment variable that you need to be concerned with is CLASSPATH. The CLASSPATH has to include the Java program(s) along with the WMQ Java support libraries.

To set the CLASSPATH variable for a Unix user, you can concatenate the directory paths to the various WMQ JAR files manually, or you can use the following Unix **export CLASSPATH** commands after ensuring that the directory path names for each of the components in the commands are corrected for your server environment:

```
export  CLASSPATH=$CLASSPATH:/home/caina/WMQ_tools/3INGs.jar:.
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/com.ibm.mqbind.jar
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/com.ibm.mq.jar
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/com.ibm.mqjms.jar
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/connector.jar
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/fscontext.jar
export  CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/rmm.jar
```

If you update the user's login profile with the **export** commands, then each time you log back onto the server, the userid will be

ready to run the programs. If you execute these commands from a user session, when you log out, the changes will be lost, and must be re-entered the next time that you want to use the programs.

TESTING THE ENVIRONMENT

Regardless of whether you have downloaded the Xephon.jar or the 3INGs.jar, a Java program has been included that will display the version of the JAR file and ensure that you have completed your environment customization correctly. When you have executed the necessary customization steps and are ready to test your environment, you can check to see whether your installation is correct by running the Hello World program from the JAR file.

Windows

From the command window where you have executed the *JSetLT.bat* file, issue one of the following commands as appropriate:

```
C:\Java> java org.Xephon.HW          (Xephon.jar)
C:\Java> java com.THREEINGs.HW       (3INGs.jar)
```

If you have downloaded the Xephon.jar file, you should see output that is similar to the following:

```
C:\Java>java org.Xephon.HW
Hello world from the Xephon - WMQ Java Library revision 10 October 2005
```

If you have downloaded the 3INGs.jar file, you should see output that is similar to the following:

```
C:\Java> java com.THREEINGs.HW
Hello world from the 3-INGs - WMQ Java Library revision 08 October 2005
```

Unix

From the command line using the userid where you have set the CLASSPATH environment variable, issue one of the following commands as appropriate:

```
caina:/home/caina/WMQ_tools> java org.Xephon.HW          (Xephon.jar)
caina:/home/caina/WMQ_tools> java com.THREEINGs.HW       (3INGs.jar)
```

If you have downloaded the Xephon.jar file, you should see output that is similar to the following:

```
caina:/home/caina/WMQ_tools> java org.Xephon.HW
Hello world from the Xephon - WMQ Java Library revision 10 October 2005
```

If you have downloaded the 3INGs.jar file, you should see output that is similar to the following:

```
caina:/home/caina/WMQ_tools> java com.THREEINGs.HW
Hello world from the 3-INGs - WMQ Java Library revision 08 October 2005
```

PROGRAM NUMBER 1 – GETQDEPTH.JAVA

The first program that we will be reviewing is *getQDepth.java*. This program checks the queue depth for a specific local queue on a queue manager. Our first program does two things:

1 Connects to a WMQ queue manager

2 Accesses a local queue.

Given that about half the phone calls I get are related to programs having problems accessing queues, this simple test is very effective.

The properties file for getQDepth.java

Each of the Java programs in this series will use a properties file to find the queue manager, queue, and other objects that are required for execution. The properties file is contained in the JAR file, along with the source code and Java class for the program.

To extract the properties file, choose the command below that matches the JAR file you have downloaded:

```
jar xvf 3INGs.jar com/THREEINGs/WMQ/getQDepth.properties
jar xvf Xephon.jar org/Xephon/getQDepth.properties
```

The commands are exactly the same on both Windows and

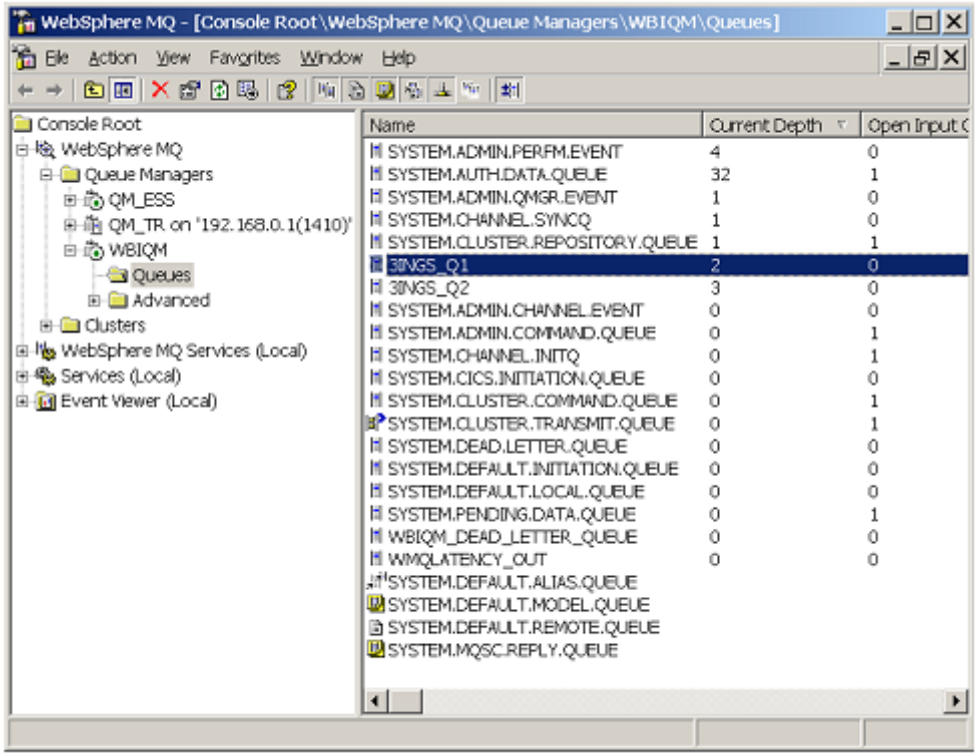

Figure 1: Output showing two queues

Unix. The properties file will be extracted, and put into local subdirectories starting with 'com' on your server. So, for example, if you execute the command using the Xephon JAR file from C:\Java, the *getQDepth.properties* file will be found in the directory *C:\Java\org\Xephon\getQDepth.properties*.

Move the *getQDepth.properties* file to your local directory, and edit it as indicated in the documentation within the file. The file will be similar to the following:

```
################################################################################
#       getQDepth.properties
#      This file contains the general properties entries for the
#      3-INGs WebSphere MQSeries Library get WMQ queue depth program
#      which is available from www.3ings.com as a Java Exemplar
#      and as part of the 3-INGs WebSphere MQSeries Simplified
#      Technology Guide.
################################################################################
#       Name of the queue manager to connect to
################################################################################
```

```
queueManager=<queue manager name>
################################################################################
#       A list of queue names to process separated by blank spaces
################################################################################
listOfQueues=<queue name> <queue name> .... <queue name>
################################################################################
#       Name of the host that the Queue Manager runs on
################################################################################
hostName=<host name>
################################################################################
#      Name of a SVRCONN channel that the program can use
#      to communication with the queue manager
################################################################################
channelName=<Application connection channel name>
################################################################################
#       TCPIP port number to be used to connect to the channel
################################################################################
portNumber=<port number>
################################################################################
#       Specify a server userid and password that can access the MQ
#       environment
################################################################################
useridMQ=<user id>
passwordMQ=<password>
```

Modify the values in brackets to match those that are applicable to your environment, and save the changes.

Running getQDepth.java

Now that you have updated the properties file, and set the environment, you are ready to execute the program. The commands will work from either a Windows command prompt or from a Unix command line. To run the program contained in the JAR file, enter the command that is appropriate to the JAR file that you downloaded:

```
java com.THREEINGs.WMQ.getQDepth getQDepth.properties
java com.Xephon.getQDepth getQDepth.properties
```

The program will then:

1. Connect to the queue manager identified in the *getQDepth.properties* file.
2. Open the queue for inquiry specified in the *getQDepth.properties* file.

3 Retrieve the queue depth and display it to the screen.

4 Repeat the above for all the other queues specified in the *getQDepth.properties* file.

On our test Windows WMQ instance, we have created two queues, 3INGS_Q1 and 3INGS_Q2 – see Figure 1.

When the getQDepth program is run on this server against both queues, the output is:

```
C:\Java\>java com.THREEINGs.WMQ.getQDepth getQDepth.properties
Q_Name> 3INGS_Q1
Q_Depth> 2
Q_Name> 3INGS_Q2
Q_Depth> 3
```

Your output will obviously differ depending on the queue names and the number of messages that are resident when you run the program.

The program code

The program code contains documentation for each section. This documentation should be sufficient for anyone who is familiar with Java and has a basic familiarity with WMQ. In future articles, we will break down each section and fully describe its purpose and syntax.

The listing for the *getQDepth.java* program is as follows:

```
/**************************************************************************
**  getQDepth.java                                                       **
**  This module is part of the 3-INGs WebSphere MQSeries Java            **
**  Library collection. This Java program is designed to attach to a     **
**  WebSphere MQSeries (WMQ) Queue, and then get the queue depth         **
**  by querying the WMQ attributes                                       **
**************************************************************************
**               Copyright 2004 - 2005 3-INGs Limited                   **
**                         All rights reserved                          **
**   This program is owned by 3-INGs Limited and is copyrighted and      **
**   licensed, not sold.                                                **
**   You may execute, copy, and modify this program in any form         **
**   without payment to 3-INGs Limited, for any purpose including       **
**   developing, using, marketing or distributing programs that         **
**   include or are derivative works of the program.                    **
**************************************************************************/
```

```
/*********************************************************************
**  Establish the Java Package and Class structures necessary to     **
**  run this method on any platform where the 3INGs_WMQ.jar file is  **
**  installed.                                                       **
**  This program is invoked using the command:                       **
**    java com.THREEINGs.getQDepth <Properties File Name>            **
**  Where <Properties File Name> is a customized properties file,    **
**  which is similar to the getQDepth.Properties found in the        **
**  3INGs_WMQ.jar file and which has been edited to reflect the      **
**  WBI attributes of the target server environment                  **
*********************************************************************/
package com.THREEINGs.WMQ;
import java.util.*;
import java.io.*;
import com.ibm.mq.*;
public class getQDepth {
/*********************************************************************
**  Declare the routine that will access the properties file, which  **
**  has been passed to this program                                  **
*********************************************************************/
    private static Properties loadProperties(String fName)
    {
       Properties props = null;
       try
        {
            FileInputStream fis = new FileInputStream( fName );
            props = new Properties();
            props.load(fis);
        }
        catch (FileNotFoundException fnfx)
        {
            System.out.println("Properties file name " + fName + " not
                    found. Make path absolute or check relative path.");
        }
        catch (SecurityException sex)
        {
           System.out.println("Read access denied to file: " + fName + "
                                              check security rights.");
        }
        catch (IOException iox)
        {
            System.out.println("IOException opening Properties
                                               configuration.");
        }
        return props;
    }
      public static void main(String args[]) {
/*********************************************************************
**  Declare the program variables used within the method             **
*********************************************************************/
```

```
            int _msgNum = 1;
            int _msgs2Process = 1;
            int _prtNum = 1414;
            int _qDepth = Ø;
            int _msglngth = Ø;
            String _qmgrName;
            String _qList;
            String _qName;
            String _hostName;
            String _chnlName;
            String _usridMQ;
            String _pwdMQ;
            GregorianCalendar _cal;
            Date _dtime;
            Calendar _today = new GregorianCalendar();
/*********************************************************************
**  Test to see whether the correct number of arguments have been    **
**  passed to the program from the command line, if not then print   **
**  usage instructions for the user and end the program              **
*********************************************************************/
            if (args.length < 1) {
                  System.out.println(" ");
                  System.out.println("Usage is java
 com.THREEINGs.getQDepth <parm file nm>");
                  System.out.println(" ");
                  System.out.println("    Where <parm file nm> is the
                                          name of the parameter");
                  System.out.println("    file that controls the
                                     execution of the program");
                  System.out.println(" ");
                  return;
            } /* if args.length */
/*********************************************************************
**  Use the parameter file name passed in to the program to set up   **
**  the Java parsing mechanism to get the values needed to continue  **
**  processing                                                       **
*********************************************************************/
            Properties config = loadProperties(args[Ø]);
/*********************************************************************
**  Retrieve the values of the variables required from the           **
**  parameter file and assign them to the appropriate variables for  **
**  execution                                                        **
*********************************************************************/
            _qmgrName = config.getProperty("queueManager");
            _qList = config.getProperty("listOfQueues");
            _hostName = config.getProperty("hostName");
            _chnlName = config.getProperty("channelName");
            String pn = config.getProperty("portNumber");
            _usridMQ = config.getProperty("useridMQ");
            _pwdMQ = config.getProperty("passwordMQ");
```

```
/*********************************************************************
**  Check the configuration parameter file variables to ensure that  **
**  appropriate values have been found, if not then error and return **
*********************************************************************/
        if (_qmgrName == null )
        {
            System.out.println("Error in configuration file queueManager
                                        property is missing or null");
            return;
        }
        if (_qList == null )
        {
            System.out.println("Error in configuration file listOfQueues
                                        property is missing or null");
            return;
        }
        if (_hostName == null )
        {
            System.out.println("Error in configuration file hostName
                                        property is missing or null");
            return;
        }
        if (_chnlName == null )
        {
            System.out.println("Error in configuration file channelName
                                        property is missing or null");
            return;
        }
        if (pn == null )
        {
            System.out.println("Error in configuration file portNumber
                                        property is missing or null");
            return;
        }
        if (_usridMQ == null )
        {
            System.out.println("Error in configuration file useridMQ
                                        property is missing or null");
            return;
        }
        if (_pwdMQ == null )
        {
            System.out.println("Error in configuration file passwordMQ
                                        property is missing or null");
            return;
        }
/*********************************************************************
**  Modify the retrieved values as appropriate for further           **
**  processing                                                       **
*********************************************************************/
```

```
            try
            {
                  _prtNum = Integer.parseInt(pn);
            }
            catch (NumberFormatException nfe)
            {
                  System.out.println("Format error for Port Number.
                              Expected an integer and got => " + pn);
                  return;
            }
            MQEnvironment.hostname = _hostName;
            MQEnvironment.channel = _chnlName;
            MQEnvironment.port = _prtNum;
            MQEnvironment.userID = _usridMQ;
            MQEnvironment.password = _pwdMQ;
        try
        {
                  StringTokenizer lstOfQs = new StringTokenizer(_qList);
              while (lstOfQs.hasMoreElements())
              {
                  _qName = lstOfQs.nextToken();
/*********************************************************************
** Connect to the queue manager that has been identified creating a  **
** new object, which can be passed along for processing              **
*********************************************************************/
                  MQQueueManager qm = new MQQueueManager(_qmgrName);
                  MQQueue qInquire =
 qm.accessQueue(_qName,MQC.MQOO_INQUIRE |
                        MQC.MQOO_SET | MQC.MQOO_BROWSE |
 MQC.MQOO_FAIL_IF_QUIESCING |
 MQC.MQOO_INPUT_SHARED);
                       _qDepth = qInquire.getCurrentDepth();
                       System.out.println("Q_Name> " + _qName);
                       System.out.println("Q_Depth> " + _qDepth);
                       qInquire.close();
                  }      /* end of StringTokenizer while */
        } /* end of try */
            catch (MQException mqe) {
/*********************************************************************
**  This catch block will handle any MQSeries exceptions that occur  **
**  and attempt to provide some hints as to what may be going wrong  **
**  when an exception occurs                                         **
*********************************************************************/
System.out.println("\nA WebSphere MQ error occurred : Completion code "
               + mqe.completionCode + " Reason code " + mqe.reasonCode);
/*********************************************************************
**  The switch will attempt to map reasons why the program may have  **
**  failed during MQSeries processing. This diagnostic can be used   **
**  to help debug failures                                           **
*********************************************************************/
```

```
switch(mqe.reasonCode) {
case 2009 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* The channel you specified may not be active       *");
System.out.println("*                                                   *");
System.out.println("*****************************************************");
break;
case 2033 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* During a read of the target queue, the program*");
System.out.println("* reached the end of the messages available         *");
System.out.println("*                                                   *");
System.out.println("*****************************************************");
break;
case 2035 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* The userid listed in the properties file has      *");
System.out.println("* failed the security check while attaching to      *");
System.out.println("* the queue manager. Make sure that the userid      *");
System.out.println("* is part of the mqm group                          *");
System.out.println("*                                                   *");
System.out.println("*****************************************************");
break;
case 2058 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* The Queue Manager Name is incorrect. Please       *");
System.out.println("* check to make sure that the entry in the          *");
System.out.println("* properties file is correct                        *");
System.out.println("*                                                   *");
System.out.println("*****************************************************");
break;
case 2059 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* The queue manager you are trying to access        *");
System.out.println("* may not be active, or the listener for the        *");
System.out.println("* port you are attempting to use may be stopped     *");
System.out.println("*                                                    *")
System.out.println("*****************************************************");
break;
case 2079 :
System.out.println("\n*****************************************************");
System.out.println("* Hint:                                             *");
System.out.println("* This is a normal message indicating that this     *");
System.out.println("* program has accepted one or more truncated        *");
System.out.println("* messages during queue processing                  *");
```

```
System.out.println("*                                                    *");
System.out.println("******************************************************");
break;
default :
System.out.println("\n*************************************************");
System.out.println("*                                                   *");
System.out.println("* The error you encountered is not documented   *");
System.out.println("* in this process. The WebSphere MQSeries (WMQ) *");
System.out.println("* error code should be reviewed using either the*");
System.out.println("* IBM documentation or the mqrc utility from the*");
System.out.println("* command line using the syntax 'mqrc <rc>'     *");
System.out.println("* where rc is the return code from WMQ message  *");
System.out.println("*                                                   *");
System.out.println("*************************************************");
} /* end of switch */
} /* end of catch */
}/* main method */
} /* public class */
```

A PREVIEW OF COMING ATTRACTIONS...

The bulk of this article has been geared towards getting your environment set to run the programs that will be delivered in future issues. The *getQDepth.java* program is a good basic example of a Java WebSphere MQSeries application.

As the series progresses, in-depth reviews of the sections of queue manager programs will be covered, and new functions will be delivered in the published JAR files. We will also cover some of the basic features of delivering code in JAR files of your own creation.

The next program in this series will be the WMQ Get application, which will allow you to browse or read WMQ messages from queues.

Aaron Cain
Independent Consultant
3-INGs Limited (UK)

How to write an authentication routine in WebSphere MQ Version 6.0

Version 6.0 of WebSphere MQ introduces a mechanism whereby user IDs and passwords supplied via the MQCONNX() call may be authenticated. In order to facilitate this, the authorization pluggable service interface has been extended to allow a user to provide their own authentication routine. This article will take a close look at how to use this extended interface to write an authentication routine, and includes some sample code. The implementation of the authorization function provided by the default authorization service will also be considered.

SUPPLYING THE USER ID AND PASSWORD TO BE AUTHENTICATED

The user ID and password to be authenticated are supplied by the application when connecting to the queue manager using the MQCONNX() call. First, note that the connection options structure (or MQCNO) has been extended in this release to allow a new structure, known as a connections security parameters structure (or MQCSP), to be specified:

```
MQCNO = { ...,
          PMQCSP SecurityParmsPtr,
          MQLONG SecurityParmsOffset }
```

The *SecurityParmsPtr* and *SecurityParmsOffset* fields of the MQCNO structure allow an MQCSP to be specified either via a direct pointer or via an offset in typical WMQ fashion. For a full description of all of the fields of the MQCNO structure, please refer to the *Application Programming Reference*. The MQCSP structure itself is defined in the following way:

```
MQCSP = { MQCHAR4   StrucId,
          MQLONG    Version,
          MQLONG    AuthenticationType,
          MQBYTE4   Reserved1,
          MQPTR     CSPUserIdPtr,
```

```
MQLONG    CSPUserIdOffset,
MQLONG    CSPUserIdLength,
MQBYTE8   Reserved2,
MQPTR     CSPPasswordPtr,
MQLONG    CSPPasswordOffset,
MQLONG    CSPPasswordLength }
```

The *StrucId* and *Version* fields are the usual WMQ structure identity and version fields, and the reserved fields are simply present to preserve pointer alignment. The key fields of interest are those that allow a user ID and password to be supplied either via a direct pointer reference or via an offset. These strings are not required to be null-terminated, but the length of both the user ID and password must be specified explicitly. This allows the user ID and password to be, theoretically, of unlimited length. The *AuthenticationType* field indicates the type of authentication to be performed. Valid values are:

- MQCSP_AUTH_NONE – do not authenticate user ID and password fields.
- MQCSP_AUTH_USER_ID_AND_PWD – authenticate user ID and password fields.

The intention is that the user supplied authorization routine will typically authenticate the user ID and password in cases where a value of MQCSP_AUTH_USER_ID_AND_PWD is specified, but will perform no authentication in the case of MQCSP_AUTH_NONE.

The following sample code, written in C, shows a typical invocation of the MQCONNX() call specifying both a user ID and password to be authenticated:

```
int main(int argc, char **argv)
{
   ...
   /*   Declare MQI structures needed                                  */
   MQCNO    ConnectOptions = {MQCNO_DEFAULT};   /* Connection Options */
   MQCSP    SecParms       = {MQCSP_DEFAULT};   /* Security Parameter */
   MQHCONN  Hcon;                  /* connection handle              */
   MQLONG   CompCode;              /* completion code                */
   MQLONG   Reason;                /* reason code                    */
   MQCHAR   QMName[50];            /* queue manager name             */
```

```
   MQCHAR   username[13];           /* username to be authenticated  */
   MQCHAR   password[13];           /* password to be authenticated  */
   strcpy(QMName,   "QM1");         /* queue manager name is QM1     */
   strcpy(username, "smith");       /* username is "smith"           */
   strcpy(password, "mypass");      /* password is "mypass"          */
   /* Set up the content of the MQCSP structure */
   SecParms->CSPUserIdPtr       = &username;
   SecParms->CSPUserIdLength    = strlen(username);
   SecParms->CSPPasswordPtr     = &password;
   SecParms->CSPPasswordLength  = strlen(password);
   SecParms->AuthenticationType = MQCSP_AUTH_USER_ID_AND_PWD;
   ConnectOptions->SecurityParmsPtr = &SecParms;
   ConnectOptions->SecurityParmsOffset = 0;
   /*   Connect to queue manager                                   */
   MQCONNX(QMName,                  /* queue manager                 */
          &ConnectOptions           /* connection options            */
          &Hcon,                    /* connection handle             */
          &CompCode,                /* completion code               */
          &Reason);                 /* reason code                   */
   /* report reason and stop if it failed     */
   if (CompCode == MQCC_FAILED)
   {
     printf("MQCONN ended with reason code %d\n", Reason);
     exit( (int)Reason );
   }
   ...
}
```

THE AUTHENTICATE USER INTERFACE

To implement the authentication routine, the user must provide a service component that conforms to the authorization service interface. In particular, the user-written service component should implement the new MQZ_AUTHENTICATE_USER function. General information on installable services and instructions on how to write and use your own service component are provided in Chapters 19 and 20 of the *System Administration Guide*. The aim of this section is to focus on the parts of the interface, and related structures, provided specifically for the task of performing authentication. Having considered these, it will then be useful to consider some sample code for a service component that implements the MQZ_AUTHENTICATE_USER function.

The MQZ_AUTHENTICATE_USER function is invoked by the

queue manager to authenticate the user ID and password supplied by an application via the MQCONNX() call. It also allows identity context fields to be set. The function is defined as follows:

```
MQZ_AUTHENTICATE_USER ( MQCHAR48       QMgrName,
                        MQCSP          SecurityParms,
                        MQZAC          ApplicationContext,
                        MQZIC          IdentityContext,
                        PMQPTR         CorrelationPtr,
                        PMQBYTE        pComponentData,
                        PMQLONG        pCompCode,
                        PMQLONG        pReason)
```

Each parameter passed into the function supplies information that may be used when performing the authentication. The key parameters of interest are:

- MQCSP SecurityParms – this is a copy of the structure that was supplied by the application when performing the MQCONNX() call. This contains the user ID and password to be authenticated. The main function of the authentication routine, which the user writes, should be to examine the user ID and password, and determine whether these supplied credentials are valid. If so, an OK return condition should be generated by setting the completion code (CompCode) to a value of *MQCC_OK*, and setting the reason to a value of *MQRC_NONE*. If the supplied credentials are not valid, a completion code of value *MQCC_FAILED* and a reason of value *MQRC_NOT_AUTHORIZED* should be returned.
- MQZAC ApplicationContext – this structure contains a variety of information relating to the calling application. The authentication routine may make use of this information when considering whether or not to authenticate a particular set of credentials, or when modifying the identity context data. The information supplied includes the application's process and thread IDs, the application name, the real and effective user ID of the application, the application bindtype, and also some contextual information that helps identify at what point this call to the authentication function was made.

- MQZIC IdentityContext – in addition to performing authentication, the MQZ_AUTHENTICATE_USER interface allows the authentication routine to manipulate the application's current identity context. The identity context is a group of three fields which, when an application puts a message to a queue, are written into the message header as part of the message context. This message context information allows any application that retrieves the message to determine information about the originator of the message. The three fields that constitute the identity context are as follows:

  – UserIdentifier – typically this is the user identifier that is associated with the process under which the application is running.

  – AccountingToken – this field is used to flow a security identifier on the Windows platforms, which is a representation of the Windows SID of the UserIdentifier.

  – ApplicationData – this may be set to any identity-related value.

  More information on identity context and the way in which it is used may be found in Chapter 5 of the *WMQ Security* manual.

  The authentication routine may set any of the identity context fields in the MQZIC structure if it so chooses. So, for example, if the routine alters the *UserIdentifier* field, the identity context information associated with the application will be altered to reflect this change. Any subsequent messages put by the application containing queue-manager-generated identity context information will contain this changed user ID in the message header.

  It is important to note that the *UserIdentifier* field of the identity context is not necessarily related to the user ID supplied by the application via the MQCSP structure. Typically the identity context's *UserIdentifier* field will

contain the user ID associated with the application's process, whereas the user ID supplied for authentication via MQCONNX() may be any user ID string as desired. If a relationship between the user ID supplied for authentication and that written into the identity context is required, it is the responsibility of the calling application and the authentication routine to maintain this.

- MQPTR CorrelationPtr – the correlation pointer allows the user's authorization service to pass resources between the various implemented functions. This pointer may be set to any address during the MQZ_AUTHENTICATE_USER function, and this value is then passed into subsequent calls to the MQZ_CHECK_AUTHORITY, MQZ_CHECK_AUTHORITY_2, MQZ_GET_AUTHORITY, MQZ_GET_AUTHORTIY_2, MQZ_SET_AUTHORITY, and MQZ_SET_AUTHORITY_2 calls. The value is maintained on a per connection to the queue manager basis, and is also passed into the MQZ_FREE_AUTHORITY function to allow resources to be freed – see the next section for details.

THE FREE USER INTERFACE

If the user's implementation of the MQZ_AUTHENTICATE_USER function allocates any resources that are required to be freed, then the MQZ_FREE_USER function must also be implemented. The MQZ_FREE_USER function is typically invoked by the queue manager during a disconnect call.

The MQZ_FREE_USER function is defined as follows:

```
MQZ_FREE_USER ( MQCHAR48 QMgrName,
                PMQZFP   FreeParms,
                PMQBYTE  ComponentData,
                PMQLONG  Continuation,
                PMQLONG  CompCode,
                PMQLONG  Reason )
```

The correlation pointer initially set by the

MQZ_AUTHENTICATE_USER function is passed into the MQZ_FREE_USER function via the *FreeParms* structure. This pointer should be used to reference any resources such as allocated storage, and these resources should then be freed by the function.

WRITING AN AUTHENTICATION ROUTINE

The following is a sample authorization service module showing the basic requirements of how to write an authentication routine. Note that the following code does not actually perform any authentication; however, it provides a framework that can be used as a basis for the user's own implementation.

Each function implementing the authorization service interface will need to return a completion code, a reason code, and a continuation flag. The continuation flag can be one of three values:

- MQZCI_DEFAULT – continuation dependent on queue manager. For MQZ_CHECK_AUTHORITY_2 this has the same effect as MQZCI_STOP.
- MQZCI_CONTINUE – continue with the next component if one exists.
- MQZCI_STOP – do not continue with the next component. Hence, return to the queue manager and user application.

More details on chaining authorization service will be explained next month.

```
/**************************************************************************/
/* BEGIN MODULE                                                           */
/**************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
#include <cmqxc.h>
#include <cmqcfc.h>
#include <cmqzc.h>
/* Define the name of this authorization service module */
#define COMPONENT_NAME "Custom.MQTest.auth.service"
```

```
int primary_process = 0;
MQLONG correlationData = 1000;
/*********************************************************************/
/* Function: MQTEST_AUTHENTICATE_USER                                */
/* Description:                                                      */
/*   An implementation of the new MQZ_AUTHENTICATE_USER interface.   */
/*   The user name and password is passed into this function via the */
/*   pSecurityParms structure.                                       */
/*********************************************************************/
static void MQENTRY MQTEST_AUTHENTICATE_USER(
                                          MQCHAR48 QMgrName,
                                          PMQCSP   pSecurityParms,
                                          PMQZAC   pApplicationContext,
                                          PMQZIC   pIdentityContext,
                                          PMQPTR   pCorrelationPtr,
                                          PMQBYTE  pComponentData,
                                          PMQLONG  pContinuation,
                                          PMQLONG  pCompCode,
                                          PMQLONG  pReason)
{
    switch (pSecurityParms->AuthenticationType)
    {
        case MQCSP_AUTH_NONE:
            /* No authentication is required so return. */
            *pContinuation = MQZCI_CONTINUE;
            *pCompCode     = MQCC_OK;
            *pReason       = MQRC_NONE;
            break;
        case MQCSP_AUTH_USER_ID_AND_PWD:
            /*****************************************************/
            /* Authentication required.                          */
            /* This is where the main routine to perform the     */
            /* authentication should be placed.                  */
            /*****************************************************/
            break;
        default:
            /* Error authentication type incorrectly set. */
            break;
    }
  /* If there is any correlation data, return the pointer to the data.*/
    *pCorrelationPtr = &correlationData;
    *pContinuation = MQZCI_CONTINUE;
    *pCompCode     = MQCC_OK;
    *pReason       = MQRC_NONE;
    return;
}
/*********************************************************************/
/* Function: MQTEST_FREE_USER                                        */
/* Description:                                                      */
/*   An implementation of the new MQZ_FREE_USER interface.           */
```

```
/*****************************************************************/
static void MQENTRY MQTEST_FREE_USER(MQCHAR48 QMgrName,
                                     PMQZFP   pFreeParms,
                                     PMQBYTE  pComponentData,
                                     PMQLONG  pContinuation,
                                     PMQLONG  pCompCode,
                                     PMQLONG  pReason)
{
     /*************************************************/
    /* Free user function:                           */
    /* Release all required resources here.          */
     /*************************************************/
    *pContinuation = MQZCI_CONTINUE;
    *pCompCode     = MQCC_OK;
    *pReason       = MQRC_NONE;
    return;
}
/*****************************************************************/
/* Function: MQStart                                             */
/* Description:                                                  */
/*  An implementation of the new MQZ_INIT_AUTHORITY interface.   */
/*  We need to define the functions used in the module that map  */
/*  to the interface. This function should be exported when      */
/*  compiling the module.                                        */
/*****************************************************************/
void MQENTRY MQStart(MQHCONFIG hc,
                     MQLONG    Options,
                     MQCHAR48  QMgrName,
                     MQLONG    ComponentDataLength,
                     PMQBYTE   pComponentData,
                     PMQLONG   pVersion,
                     PMQLONG   pCompCode,
                     PMQLONG   pReason)
{
    MQLONG CC       = MQCC_OK;
    MQLONG Reason   = MQRC_NONE;
    /* Set primary process flag */
    if ((Options & MQZIO_PRIMARY) == MQZIO_PRIMARY)
        primary_process = 1;
     /*************************************************************/
    /* Initialize the entry point vectors.  This is performed for  */
    /* both global and process initialization, ie whatever the     */
    /* value of the Options field.                                 */
     /*************************************************************/
    if (CC == MQCC_OK)
      MQZEP(hc, MQZID_INIT_AUTHORITY,(PMQFUNC)MQStart,&CC,&Reason);
    if (CC == MQCC_OK)
  MQZEP(hc,MQZID_AUTHENTICATE_USER,(PMQFUNC)MQTEST_AUTHENTICATE_USER,&CC,&Reason);
    if (CC == MQCC_OK)
      MQZEP(hc,MQZID_FREE_USER,(PMQFUNC)MQTEST_FREE_USER,&CC,&Reason);
```

```
    /*****************************************************************/
    /* Set the version number.                                       */
    /*****************************************************************/
    *pVersion   = MQZAS_VERSION_5;
    /*****************************************************************/
    /* Set the return codes.                                         */
    /*****************************************************************/
    *pCompCode = CC;
    *pReason   = Reason;
    return;
}
#if MQ_PLATFORM == OS400
void *MQPlugInit()
{
  return (void *)MQStart;
}
#endif
```

Editor's note: this article will be concluded next month.

May Leung and David Postlethwaite
Software Engineers
IBM (UK)

Cressida Technology has announced that OpenDemand Systems has announced Version 5.0 of OpenLoad, its browser-based, enterprise-functional, load-testing and monitoring solution for dynamic Web sites, applications, and services.

OpenLoad uses WebSphere and DB2 UDB. Its User Scenario Session Finder is an intelligent, script-free wizard that allows users to build dynamic data-driven tests without having to code and debug, or understand, the inner workings of the applications they are testing.

The product also allows users to proactively manage the performance of their critical Web applications from a single browser-based console. This new feature enables users to leverage real-world user scenarios developed for functional and load testing to continually monitor customer transactions and correlate them with their back-end systems to identify and fix problems before they impact Web site visitors.

For further information contact:
URL: www.opendemand.com/openload.

* * *

ReQuest for WebSphere MQ now runs on z/OS. ReQuest currently supports Windows, AIX, Solaris, HP-UX, and SUSE Linux platforms.

Request for WebSphere MQ is a message tracking, message reporting, message replay, point-in-time message recovery, charge-back, accounting, and auditing solution. ReQuest uses filtering technology to analyse critical message activity information already contained in WMQ logs.

For further information contact:
URL: www.cressida.info/products_cressida_ReQuest.htm.

* * *

IFS has announced that Version 7 of IFS Applications allows companies to use IFS Applications' content and processes directly within IBM WebSphere Portal. This capability complements the personal portals available in IFS Applications. It adds the ability to build scalable portals using WebSphere, which improves employee productivity and increases customer loyalty.

IFS now enables companies to include all IFS portlets in an enterprise-wide WebSphere Portal.

For further information contact:
URL: www.ifsworld.com/uk/solutions/ifs_applications.

* * *

ILOG has announced it is the first Business Rule Management Systems (BRMS) software vendor to offer integration with WebSphere Process Server Version 6.0. The integration allows users to better manage their policy-intensive business processes.

A new connector developed by ILOG provides seamless integration between ILOG JRules and WebSphere Process Server 6.0. The combination of the IBM and ILOG products will allow users to make policy changes themselves when creating and using business process automation platforms.

For further information contact:
URL: www.ilog.com/corporate/releases/us/050914_ibm.cfm.