37

MQ update

July 2002

In this issue

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

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.

Editor

Madeleine Hudson
E-mail: MadeleineH@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 £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.

Printed in England.

Scheduled MQ object back-ups on AIX

INTRODUCTION

Making regular back-ups of your MQ configuration is an easy way to ensure quick recovery from an MQ object corruption, for example, a deleted queue or a corrupt log.

Linear logging provides media image back-up and recovery using the **rcdmqimg** and **rcrmqobj** commands: circular logging does not provide this function. System back-up of the file system can also be performed. Restoring objects with these methods can take some time.

An alternative and easy to use method is *SupportPac MS03 – saveqmgr*. It interrogates the attributes of all the objects defined to a queue manager and saves them to a file. The format of this file is suitable for use with **runmqsc** and can be viewed and edited to select the desired objects to restore. Running the saveqmgr utility regularly can also provide an audit trail of changes to your MQ environment.

The following is a quick and easy way to schedule saveqmgr daily and store the output for a defined number of generations. The script is written using the korn shell for use on AIX but the logic of the script could be ported to other platforms. The script runs the saveqmgr utility for a specified MQ queue manager. It writes the output to a date-stamped file and deletes files older than a specified age.

SCRIPT

```
#!/bin/ksh
#qmgrsave
# Script to automate collection of saveqmgr (MSØ3) output
# and perform housekeeping of the files.
# Input parameter is the name of the Queue Manager
# schedule using crontab
# Author:  Rab McGill
# Date:    28/2/Ø2
# set path where saveqmgr and find commands are.
PATH=.:/data/MQM/data1Ø/scripts:/usr/bin/
export PATH
# Input arguments
QMGR=$1
```

```
# output path
lpath="/data/MQM/data10/defs/"
#date stamp for output file name
TODAY=$(/bin/date +%b-%d-%y)
# clear output file incase you want to run twice to same file
>${lpath}${QMGR}_$TODAY.tst
# run saveqmgr
saveqmgr -m $QMGR -f ${lpath}${QMGR}_$TODAY.tst
#housekeeping - delete files older than a month
find $lpath -name $QMGR"_*" -type f -mtime +32 -exec rm -r {} \;
```

CRONTAB ENTRY

The above script can be scheduled using crontab.

```
30 00 * *  /data/MQM/data10/scripts/qmgrsave QP1 1>/dev/null 2>/dev/
null
```

This runs the qmgrsave script every day at 12.30 am.

Rab McGill
Senior Technician (UK)

API Exits – a new interface

This article describes some of the features of the newest interface available to developers and administrators of MQSeries on distributed platforms: the API Exit.

On MQSeries for OS/390 there has always been available in the CICS environment an interface called the API-Crossing Exit. This is invoked every time an MQI call is made by an application, allowing the writer of an exit to inspect or modify the parameters to the MQI call. A similar exit point is now available on the distributed platforms but with some significant differences. In this article I will explain some of the features of this new interface and show how to write an exit. I will include the source code for a simple exit, which could form the basis of a more sophisticated statistics-gathering package.

There are many potential uses of the API Exit interface. Ideas that have been suggested include compression, statistics gathering, encryption, message logging, monitoring, and enforcing local rules for application programs. A number of products already exist which provide some of these services.

However, as they typically work by using their own mechanisms for intercepting or replacing the standard MQI libraries they are unlikely to work well together or will have problems when new versions or patches are provided for MQSeries.

This new interface provides an IBM-sanctioned way for them to intercept the calls and a number of software vendors have already expressed interest in converting their products to the new form. Several of these had the opportunity to comment on early prototypes and specifications for the interface, to ensure that IBM was developing something that they would be able to use.

An application does not need to use the MQI directly in order for the API Exit to be called. Higher-level interfaces such as the AMI or MQAI are all implemented internally with calls to the MQI. These are then intercepted by the API Exit code. Also it does not matter in which language the application is written: all language bindings, such as the Java classes or the COBOL version of the MQI, pass through the same intercept points in the MQI libraries.

The API Exits are transparent to applications, which do not have to be recompiled or relinked. All existing applications will automatically be passed through configured API Exits.

AVAILABILITY

The new interface was first shipped in CSD#3 for MQSeries V5.2. While initially it will be supported only on Solaris it will actually work on all of the distributed platforms – Unix, Linux, Windows NT, OS/400 – at the same time. The enabling code is common to all of the MQSeries implementations on those platforms and has no platform-specific features.

The reason that the interface is only documented to be on Solaris is purely a question of the amount of resource available to test it while the CSD was being developed. At some future time the interface will be officially supported on all of the distributed platforms. When I wrote the exit that appears later in this article I actually developed it on Linux and AIX first before moving it to Solaris. The header file changes needed to compile an exit are included in CSD#3 for all of these platforms.

Although the API Exits are not currently being provided on OS/390 the specification for the interface was developed in conjunction with that

product's development team to ensure that it could be implemented at some future date.

Documentation for the API Exit is included in the *MEMO.PTF* file included in the CSD, which can be downloaded from the anonymous ftp server at *ftp.software.ibm.com* in subdirectory *software/mqseries/fixes/solaris52/U478291*.

Any subsequent patches will also include this function.

Included in the CSD is the source code for an example exit that can be configured to write to a log file every parameter to every MQI call made by an application program. This is a very useful trace capability, but the size of the code needed to format the function calls may obscure features of the API Exit.

CONFIGURATION

The API Exit is configured for each queue manager on a system by adding stanzas to either the *system-wide mqs.ini* file or the queue manager's *qm.ini* file. In *mqs.ini* you can define template settings which will be copied to newly created queue managers and you can also define exits which will be invoked for all queue managers on the box. In the *qm.ini* file you can place exit definitions that are specific to that queue manager.

All of the stanzas have the same format, containing the path to the exit module, the name of the entry-point function, a sequence number, and a piece of data passed to the exit when it is initialized. This data is useful for passing some control parameters to the exit and can be interpreted in any way the exit-writer wishes. The CICS API-Crossing Exit only allows a single exit to be configured for a queue manager but the new interface makes it possible to have many independent exits, which are called in turn by the queue manager. These exits are invoked in the order given by their sequence number. The sequence numbers do not need to be consecutive (eg 1,2,3…) but can be spread out (eg 10,20,30). If two exits are given the same sequence number, the order in which they are called is not defined.

The path to the exit module should always be a fully qualified pathname with a complete directory string. This will stop users from attempting to bypass the exit by setting environment variables such as LD_LIBRARY_PATH. The filename of the compiled exit code on Unix ought to have no suffix similar to installable services such as the

Authorization Component (amqzfu). On AIX, Linux, and HP-UX you will need to provide more than one variant of the API Exit to handle non-threaded and multi-threaded applications, and the queue manager will automatically append '_r' to the configured name of the exit on these platforms. On Windows NT the exit module should have a *.DLL* suffix.

The list of exits being used by the queue manager is only read at start-up time. If the list is changed the queue manager must be stopped and restarted. Updating the exit module while the queue manager is running may not be possible, depending on how the operating system deals with changes to in-use files. If a module is changed while the queue manager is running you might find some applications using the old version of the exit code, while newly started applications use the later version of the module.

Although the queue manager does not really drive the exits from its internal processing it does attempt a 'consistency' check during start-up by loading all configured exits. This ensures that the modules do exist in the places stated in the ini files, and that they have the correct format for dynamic loading. This is not a 100% guarantee that the exits will actually work but it is a reasonable attempt. If a defined exit cannot be loaded, the queue manager will report an error and fail to start.

WHEN IS THE EXIT INVOKED AND WHAT CAN IT DO?

The API Exit module may get executed at least twice for every MQI call made by every application connected to a queue manager. This sounds as though it might be a significant overhead for exits that are only interested in a small number of MQI verbs, but the exit itself is given the choice of how many verbs it is prepared to deal with. It is the exit code that registers an interest in which verbs are to be called, and if it does not register for a particular verb then there is minimal overhead involved in processing this.

Almost all of the MQI parameters can be modified by an exit. For example, an exit can replace input and output data buffers, change the flags in a structure's options field, and override queue names set by the application.

One of the few parameters that cannot be changed is the name of the queue manager during MQCONN. It was necessary to impose this rule in order to maintain acceptable performance of each application's start-up; the configuration of the exits is loaded into the queue manager as it starts and this cache is read by the application after it has done most of the MQCONN

work, so it is already connected by the time any exit could attempt to modify the name.

An exit can also call the MQI directly, using the application's connection handle. This could be used to put messages to exit-defined queues or issue MQINQ to discover attributes of objects being used by the application. Calls made from one exit are not passed to other exits; this stops potential problems with recursion and keeps the work of an exit away from modification by other exits in a chain.

BEFORE and AFTER processing

When an application makes an MQI call, agent processes acting on the application's behalf do most of the real work. However, there is a phase of marshalling the parameters before going to the agent and then unpacking received data and handling return codes. The API Exit has been inserted into the MQI libraries so that it is called just before the transfer to the agent process and then again just after the transfer back. These procedures are referred to as the BEFORE and AFTER processing points, and they are run within the application process.

When the MQI libraries invoke the BEFORE exit code, each configured exit which has registered its interest is given a chance to inspect and modify the parameters that will be sent to the agent process. Any modifications to the parameters made by one exit will be 'inherited' by subsequent exits, which do not know, or need to know, about other exits in the chain. The BEFORE exit points are called in the configured sequence order. The BEFORE exits will be called before the queue manager has had the opportunity to do validation on most of the parameters; it is possible that an exit may be passed an invalid data buffer that would normally be caught by the queue manager.

After the queue manager has done its work the AFTER exit points are called in the same way. However, this time the configured sequence is handled in reverse. So an exit with sequence number 30 will be called before an exit with sequence number ten. Reversing the order allows each exit to undo its own work if necessary, before another exit gets the chance to handle the call.

As an example of this, consider two exits that might have been written by two independent people and which should have no knowledge of one another. One exit is designed to compress and uncompress messages; the

other is an encryption package. We need to ensure that, during MQPUT, messages are compressed before being encrypted, but during MQGET the decrypt has to be done first in order that decompression can be successfully completed. As the operations for these exits are done in the BEFORE phase, during MQPUT, and the MQGET work is done in the AFTER phase, it should be clear why the reversal of order is needed.

In fact for MQGET there are two AFTER stages that are called. Data conversion such as ASCII/EBCDIC mapping has always been done in the application process after a message has been removed from the queue. Some exits are likely to be interested in the raw message and some in the transformed message. For example, a compression component will need to uncompress the data before any data conversion is performed, but an exit that is carrying out monitoring of message content will probably be interested in the 'readable' form of the message data after it has been converted. So the API Exit interface allows exits to request that they be called at either or both of these stages.

MQSeries internal operations

The API Exit is invoked only for applications that are using the MQI. This includes some of the MQSeries-provided programs, such as trigger monitors and the command server. However, it does not include some of the message generation that is carried out internally by the queue manager. In particular, report messages such as Expiry, COA, or COD, and event messages such as Queue Full, are not passed to the API Exit. Trigger messages are also not passed to API Exits. However, work done by the channel programs (the MCAs) is all done via the MQI and so these will be intercepted. Applications using 'trusted' bindings (the FASTPATH option to MQCONNX) are intercepted by API Exits.

MQSeries clients

The API Exit is not run directly inside MQSeries client programs (using CLNTCONN channels). However, it is still possible to intercept and modify what the clients are doing.

When a client program is running, all of its MQI calls are sent across the channel and are executed remotely by a proxy process. The proxy might be the *amqcrsta* program or a thread running within the **runmqlsr** process. The API Exit is still called as it intercepts this proxy's MQI calls.

This design has both benefits and disadvantages. The biggest drawback is that the exit does not have access to the real application's environment, for example its security context. On the other hand, running the exit inside the proxy removes the need to distribute exits to many client-only machines; removes the need to write duplicate function in Java to support the Java client; and solves the problem of dealing with data conversion, which is also carried out inside the proxy. If the exit were to be run inside the client process there would have to be some way of either moving data conversion (and user-written conversion exits) to the client or adding more performance-degrading network traffic to the client protocol so that conversion is still done in the server. While neither choice is perfect, supporting the API Exit in the proxy seemed to be the best solution.

The MQAXC structure described in the reference material does contain a flag to indicate that the SVRCONN end of a client channel is driving an API Exit. There is one small catch to this: the proxy does make a few MQI calls during initialization of a SVRCONN channel, and although these are also intercepted by the API Exit it is not until they have completed that we know that it really is a SVRCONN and not some other channel type. Use the shipped example API Exit *amqsaxe0* to trace the MQI calls made by the proxy to see how this works.

If you want to transfer more environmental information from the client program, it ought to be possible to write a pair of channel security exits. The SVRCONN end of these could then pass the data to the initialization routine of an API Exit. One way of achieving that could be to implement both the API Exit and the security exit in the same module, which would be loaded and executed by the same thread in the proxy process; a global variable in that module could be used to anchor 'shared' control blocks.

DESIGNING AN API EXIT

The documentation for the API Exit interface describes all of the control blocks and function calls made from the queue manager into the exit. In summary, there are two new control blocks. One of these describes the application context (the MQAXC structure) and the other contains information that tells the exit why and how it has been called (the MQAXP structure). Writers of channel exits will spot a resemblance between the MQAXP and the MQCXP structures; they have similar purposes. The MQAXP includes several fields that can be used to pass data between different instances of this exit and across MQI calls. It also has a field that

can be used for cooperating exits to manage shared state information, but I suspect that not many people will use this capability.

The rest of the parameters to an API exit are approximately the same as the equivalent MQI call, but most have been given an extra layer of indirection. For example, a simple variable in the MQI is now passed via a pointer; a buffer of data is shown to the exit as a pointer-to-a-pointer.

The interface specification is given only in terms of a C language binding. While it might be possible to write an API Exit in other languages on some platforms, only C has been tested and supported. In particular, C++ is not recommended because there might not be a suitable runtime start-up environment available to the exit; this is similar to the requirements for channel exits.

Initialization and termination

Whenever an application calls MQCONN or MQCONNX all configured API Exits are loaded (if the module is not already loaded in this process) and its initialization function is called. This is the function named in the *qm.ini* file. The initialization function should do whatever is necessary to create its state-management variables for this connection. Typically this will involve allocating some memory, which will be used to anchor other data structures later. The exit will then register with the queue manager to say which phases of which MQI verbs it wishes to handle. Many exits, for example, will not be interested in the lesser-used verbs such as MQINQ and MQSET. Use the MQXEP function to pass the addresses of the different functions you want to intercept.

No configuration parameters are provided with MQSeries to have API Exits used only with certain classes of applications or when specific queues are being used; managing this is the exit's responsibility. One suggestion is to use the ExitData parameter set in the *qm.ini* file to act as an indicator to exits as to which types of application it might want to intercept and report on.

The termination function, if you register to receive it, will be called after the application thread has completed its MQDISC. You will not be able to call MQI functions at this point as the connection handle will have been destroyed; any MQI tidy-up should be done during the BEFORE phase of MQDISC. The termination function should only do things such as free any allocated storage and close files.

You can call MQXEP at any time to change any MQI functions you want to intercept; however, most exits will probably set up the list during the initialization phase and leave it alone. The most likely reason to change the interception points is when the exit is running inside an MCA. I expect that many exits will not be interested in doing any work when a regular channel is running; however, they may want to intercept MQI clients. As the flag saying whether or not a channel is a SVRCONN is not available immediately, an exit will need to register for several MQI calls and inspect the environment field in the MQAXC structure before deciding whether or not to continue. By the time the first MQPUT or MQGET is processed the MQXE_MCA or MQXE_MCA_SVRCONN indicator will be reliable.

Handling errors and debugging exits

The exit can return error codes direct to the application or cause further processing of the MQI call to be halted. If the exit is in the BEFORE phase an error code will stop further exits in the chain from being called; the real MQI call is aborted and exits which have registered for the AFTER phase will be called if they have a lower sequence number than the exit that generated the error. If the exit is in the AFTER phase the chain of exits continues to be run.

The documentation contains a table of rules showing which exits get called when errors occur, but the fundamental rule is that exits need to be given the opportunity to clean-up when another exit in the chain (or the real MQI call) has returned an error.

I would recommend that exits running in the AFTER phase do not return errors that would get sent to applications as this might mask the fact that the real MQI work was done successfully. The queue manager does not attempt to reverse any operations that have been completed. For example, if an exit running AFTER an otherwise successful call to MQOPEN indicates an error, the application does not know that it will still need to MQCLOSE the queue. The only exception to this rule is that errors generated during MQCONN will cause the connection to be broken.

You can use the ExitPDArea in the MQAXP structure for debugging and tracing an exit. This is a 48-byte area returned after each invocation of an exit function whose contents are written to the MQSeries trace records whenever trace is turned on. During initial development of an exit, putting printf statements in the exit might be allowed, but this should not be done for a production-level exit as there are no guarantees about the environment

in which applications might be running. The ExitPDArea is a convenient place to put information about your exit's behaviour; it will then be visible to service and support staff that might need to read MQSeries traces.

If a real disaster occurs and the exit code causes something like an SEGV, the application program might exit. However, this does not affect the integrity of the queue manager processes and its data and transactional semantics are maintained. Of course, if the application is running as a trusted program then the normal considerations apply there: anything running in the same process as queue manager code has the opportunity to corrupt critical resources.

Security contexts

The API Exit is run with the same context as the surrounding application both for operating system and for MQSeries purposes. If an exit is going to use the MQI, for example, to put messages to its own queue, this queue will need to be accessible to all users. Use the 'nobody' group on the **setmqaut** command to grant everybody access to these objects. The sample exit included in this article requires this command:

```
setmqaut -t q -m qmgr -n APIX.STAT -g nobody +put
```

When running inside an MCA the exit will initially have full MQSeries rights (assuming the program is being run by the mqm user). If the channel turns out to be a SVRCONN proxy the queue manager modifies the security context so that the exit will only have access to the same objects as the MCAUSER.

Multi-threaded environments

Solaris, the only platform supported initially, requires that all applications and hence all exits are compiled for multi-threaded operations. On some other systems, notably HP-UX and AIX, some applications might be threaded and some not. For these systems, two versions of your exit will need to be compiled and installed. The queue manager will automatically load the correct version of the exit.

Normal multi-threaded development concerns will apply here. For example, you might need to implement cross-thread locking to stop two threads from simultaneously updating a global state variable. However, the ExitUserArea can be used as a thread-safe anchor point for memory allocation as a new copy of the structure is assigned to each MQCONN. Although the queue

manager attempts to load exits during start-up only the multi-threaded version will be loaded and there cannot be any check that a non-threaded variant is also available.

Handling transactions

It is very strongly recommended that an exit does not change the transactional state of an application. For example, if the application only uses NO_SYNCPOINT options when handling messages, an exit which is also manipulating its own messages should not return from an MQI call with an outstanding SYNCPOINT message. If the application is using SYNCPOINT messages the exit may decide to also use SYNCPOINT options and rely on the application's MQCMIT to complete the transaction.

When an external transaction coordinator is being used, such as WebSphere, this will not be visible to API Exits. The two-phase transaction completion flows are mapped so that the API Exit just sees calls to MQI verbs. The entry to the xa_prepare is treated as though it were the BEFORE phase of an MQCMIT. The final stage of transaction resolution, which might be xa_commit or xa_rollback, will be transformed into the AFTER phase of MQCMIT or MQBACK.

Note that, at the time of writing, none of this external coordination function had been included in CSD#3; the API Exit was not being called at all. This is a bug that has been accepted by IBM development and will be fixed in the next CSD, if not before.

Message expansion

One thing to be aware of when writing exits that modify data during an MQGET is how message expansion is handled.

An application provides a buffer into which the retrieved data is to be returned. With the API Exit it is possible to change the data before the application sees it. In some cases the API Exit will have used its own buffer, expecting to transform the data in some way before completion of the MQGET. If the application buffer is too small for the message data after the exit has changed it, there is no way to restore the message to its original position on the queue. Depending on the options used by the application's MQGET, the exit might decide to return an error or warning code.

This is identical to the situation of a data conversion exit, which expands a message beyond the length available in the application's buffer, but is

perhaps more likely to happen with exits involved in transformations such as encryption or compression. An exit writer needs to be aware of common behaviour, such as applications that perform an MQGET with a zero-length buffer, in order to discover the real length of a message and then reissue the call.

THE SAMPLE EXIT

The exit included below is a simple piece of code whose intent is to demonstrate the most important features of API Exits. For each application it collects basic statistics, showing the total number of messages and bytes and the maximum length of messages. When an application ends, this information is written in a message to a queue, from where it can be retrieved.

There is no attempt to make the code bullet-proof; if an application abends just before the MQDISC the statistics are not put to the queue. I also haven't provided an analysis tool as this can be done trivially. For example, I've used an awk script that parsed the output from a version of the *amqsget* program (I needed to increase its message buffer size from 101 bytes) to summarize the gathered information.

There are a number of obvious extensions to this exit, such as maintaining statistics for individual queues, but adding code to handle that would have obscured the structure. Another enhancement could be to write the statistics at some 'batch' interval and not just during termination to avoid losing information from applications which abend.

APIXSTAT.C

```
/* MODULE: apixstat.c
 * DESCRIPTION: Example API Exit to count MQ message operations
 * AUTHOR: Mark Taylor, IBM Hursley
 * DATE: January 2002
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
#include <cmqxc.h>
/* Definition of what we're going to log. This contains info about
 * all the messages put and got by this application, no matter what
 * queue. I could start tracking number of open queues, operations per
 * queue etc, but that starts to make this example too big and would
 * obscure the basic exit.
```

```
 */
struct mqstats {
    MQLONG Environment;
    long BytesPut;
    long BytesGot;
    long BytesBrowsed;
    long MsgPut;
    long MsgGot;
    long MsgBrowsed;
    long MaxLengthPut;
    long MaxLengthGot;
};
/* Standard MQ Entrypoint. Not directly used, but
 * required by some platforms.
 */
void MQStart(){;}
/* Declare internal functions. All except
 * the entrypoint can be static as they're not used externally.
 */
MQ_INIT_EXIT EntryPoint;
static MQ_TERM_EXIT Terminate;
static MQ_PUT_EXIT  PutCounter;
static MQ_PUT1_EXIT Put1Counter;
static MQ_GET_EXIT  GetCounter;
static MQ_DISC_EXIT WriteStats;
static void ClientCheck(struct mqstats *,PMQAXP,PMQAXC);
/* Collect statistics on messages put to queues.
 * Need to do this for both MQPUT and MQPUT1 calls.
 */
static void MQENTRY PutCounter ( PMQAXP    pExitParms
                    , PMQAXC    pExitContext
                    , PMQHCONN  pHconn
                    , PMQHOBJ   pHobj
                    , PPMQMD    ppMsgDesc
                    , PPMQPMO   ppPutMsgOpts
                    , PMQLONG   pBufferLength
                    , PPMQVOID  ppBuffer
                    , PMQLONG   pCompCode
                    , PMQLONG   pReason
                    )
{
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = *ppmqstats;
  if (pmqstats->Environment == -1)
    ClientCheck(pmqstats,pExitParms,pExitContext);
  pmqstats->MsgPut ++;
  pmqstats->BytesPut += *pBufferLength;
  if (*pBufferLength > pmqstats->MaxLengthPut)
    pmqstats->MaxLengthPut = *pBufferLength;
  return;
}
static void MQENTRY Put1Counter ( PMQAXP    pExitParms
```



```
                    , PMQAXC    pExitContext
                    , PMQHCONN  pHconn
                    , PPMQOD    ppObjDesc
                    , PPMQMD    ppMsgDesc
                    , PPMQPMO   ppPutMsgOpts
                    , PMQLONG   pBufferLength
                    , PPMQVOID  ppBuffer
                    , PMQLONG   pCompCode
                    , PMQLONG   pReason
                    )
{
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = *ppmqstats;
  if (pmqstats->Environment == -1)
    ClientCheck(pmqstats,pExitParms,pExitContext);
  pmqstats->MsgPut ++;
  pmqstats->BytesPut += *pBufferLength;
  if (*pBufferLength > pmqstats->MaxLengthPut)
    pmqstats->MaxLengthPut = *pBufferLength;
  return;
}
/* Collect statistics on messages got from queues.
 * Only update the stats if the retrieval was successful (or there
 * was a warning, probably due to truncation). Browsed messages
 * are monitored separately from destructive retrieval.
 */
static void MQENTRY GetCounter ( PMQAXP    pExitParms
                    , PMQAXC    pExitContext
                    , PMQHCONN  pHconn
                    , PMQHOBJ   pHobj
                    , PPMQMD    ppMsgDesc
                    , PPMQGMO   ppGetMsgOpts
                    , PMQLONG   pBufferLength
                    , PPMQVOID  ppBuffer
                    , PPMQLONG  ppDataLength
                    , PMQLONG   pCompCode
                    , PMQLONG   pReason
                    )
{
  MQLONG options;
  MQLONG browseopts = (MQGMO_BROWSE_FIRST |
                  MQGMO_BROWSE_NEXT  |
                  MQGMO_BROWSE_MSG_UNDER_CURSOR);
  PMQGMO gmo = *ppGetMsgOpts;
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = *ppmqstats;
  if (pmqstats->Environment == -1)
    ClientCheck(pmqstats,pExitParms,pExitContext);
  if (*pCompCode != MQCC_FAILED )
  {
    options = gmo->Options;
    if (options & browseopts)
```

```
    {
      pmqstats->MsgBrowsed ++;
      pmqstats->BytesBrowsed += **ppDataLength;
    }
    else
    {
      pmqstats->MsgGot ++;
      pmqstats->BytesGot += **ppDataLength;
      if  (**ppDataLength > pmqstats->MaxLengthGot)
   pmqstats->MaxLengthGot = **ppDataLength;
    }
  }
  return;
}
/* This function is called during MQDISC, before the actual disconnect
 * has happened.It builds a message from the statistics gathered and
 * puts it to a queue, which must have been previously defined.
 * The defaults are used for putting messages, except for forcing an
 * out-of-sync operation.
 * This MQPUT1 does not itself get passed to an API exit, as the calls
 * are not recursive.
 */
static void MQENTRY WriteStats ( PMQAXP    pExitParms
                   , PMQAXC    pExitContext
                   , PPMQHCONN ppHconn
                   , PMQLONG   pCompCode
                   , PMQLONG   pReason
                   )
{
  MQLONG LocalCompCode = MQCC_OK;
  MQLONG LocalReason   = MQRC_NONE;
  MQLONG BufferLength;
  MQBYTE Buffer[1024];
  MQMD   MsgDesc = {MQMD_DEFAULT};
  MQOD   ObjDesc = {MQOD_DEFAULT};
  MQPMO  PMO     = {MQPMO_DEFAULT};
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = *ppmqstats;
  sprintf((char *)Buffer,
  "App:%-28.28s User:%-12.12s Env:%08X MsgPut:%d MsgGot:%d MsgBrw:%d
BytesPut:%d BytesGot:%d BytesBrw: %d MaxPut:%d MaxGot:%d\n",
    (pmqstats->Environment == MQXE_MCA_SVRCONN)?
   "ClientProxy" : pExitContext->ApplName,
    pExitContext->UserId,
    pmqstats->Environment,
    pmqstats->MsgPut,pmqstats->MsgGot,pmqstats->MsgBrowsed,
    pmqstats->BytesPut,pmqstats->BytesGot,pmqstats->BytesBrowsed,
    pmqstats->MaxLengthPut,pmqstats->MaxLengthGot);
  BufferLength = strlen((char *)Buffer);
  strcpy(ObjDesc.ObjectName,"APIX.STAT");
  memcpy(MsgDesc.Format,MQFMT_STRING,8);
  PMO.Options |= MQPMO_NO_SYNCPOINT;
```



```
  PMO.Options |= MQPMO_FAIL_IF_QUIESCING;
  /* Don't use the application's CC/RC variables, as
   * we want to continue even if this MQPUT1 fails.
   */
  MQPUT1(**ppHconn,
    &ObjDesc,
    &MsgDesc,
    &PMO,
    BufferLength,
    Buffer,
    &LocalCompCode,
    &LocalReason
    );
  return;
}
/* This is called just after the MQDISC, and can be used to
 * tidy-up any resources we've used during the run.
 */
static void MQENTRY Terminate ( PMQAXP   pExitParms
                  , PMQAXC   pExitContext
                  , PMQLONG  pCompCode
                  , PMQLONG  pReason
                  )
{
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = *ppmqstats;
  free(pmqstats);
  return;
}
/* Initialization function. This will be called each time an
 * application connects to the queue manager.
 */
void MQENTRY EntryPoint ( PMQAXP   pExitParms
                   , PMQAXC   pExitContext
                   , PMQLONG  pCompCode
                   , PMQLONG  pReason
                   )
{
  MQLONG            rc             = MQRC_NONE;
  char buf[256]={Ø};
  struct mqstats **ppmqstats = (void *)&pExitParms->ExitUserArea;
  struct mqstats *pmqstats = NULL;
  /* This version string will be logged in MQ's trace so we
   * can see it's been called.
   */
#ifdef _REENTRANT
  char *t="threaded";
#else
  char *t="unthreaded";
#endif
  sprintf(buf,"APIXSTAT ver %s %s (%s)\n",__DATE__,__TIME__,t);
  memcpy(pExitParms->ExitPDArea,buf,48);
```

```
  /* This is the anchor point for our per-connect information.
   * The pointer can be stored in the ExitParms structure, and will be
   * returned on all future calls by this particular thread/connection.
   * We don't need to worry about threads or locks when updating the
   * MQAXP structure as thread-specific versions have already been
   * allocated by the queue manager. Of course, any system calls made
   * here (like malloc) will need to be thread-safe.
   * ExitUserArea is a 16 byte array, easily large enough to hold a
   * pointer value.
   */
  pmqstats = malloc(sizeof(struct mqstats));
  if (pmqstats)
  {
    *ppmqstats = pmqstats;
    bzero(pmqstats,sizeof(struct mqstats));
    pmqstats->Environment = pExitContext->Environment;
    if (pmqstats->Environment == MQXE_MCA)
      pmqstats->Environment = -1; /* Mark as unknown for now ... */
  }
  else
  {
    /* If the malloc failed, then fail as fast as possible.
     */
    rc = MQRC_API_EXIT_ERROR;
    pExitParms->ExitResponse = MQXCC_FAILED;
  }
/* Register for the verbs we're interested in - MQPUT(1), MQGET, MQDISC
* and for the terminate routine so we can clear up allocated resources.
* If I wanted to track only committed message operations then I'd need
* to also intercept MQCMIT and MQBACK, and keep counts for the state of
* PUT and GOT messages. That's too much work for this example.
   */
  if (rc == MQRC_NONE)
  {
    MQXEP ( pExitParms->Hconfig, MQXR_BEFORE, MQXF_PUT
     , (PMQFUNC) PutCounter, Ø, pCompCode, pReason);
    MQXEP ( pExitParms->Hconfig, MQXR_BEFORE, MQXF_PUT1
     , (PMQFUNC) PutCounter, Ø, pCompCode, pReason);
    MQXEP ( pExitParms->Hconfig , MQXR_AFTER, MQXF_GET
     , (PMQFUNC) GetCounter, Ø, pCompCode, pReason);
    MQXEP ( pExitParms->Hconfig , MQXR_BEFORE, MQXF_DISC
     , (PMQFUNC) WriteStats, Ø, pCompCode, pReason);
    MQXEP ( pExitParms->Hconfig , MQXR_CONNECTION, MQXF_TERM
     , (PMQFUNC) Terminate,  Ø, pCompCode, pReason);
  }
  if (rc != MQRC_NONE)
  {
    pExitParms->ExitResponse = MQXCC_FAILED;
  }
  return;
}
/* For the first few MQI calls, we don't know if
```

```
 * an MCA is going to be a SVRCONN or not. The only calls
 * being intercepted by this particular module (PUT/GET)
 * will be done after that decision has been made, so by the
 * time we get here we'll be able to set the flag unconditionally and
 * then continue. It's not until the decision is made that you can
 * check the userids. More sophisticated exits may like to do more
 * tracking in this function before eg unregistering calls so a regular
 * MCA isn't tracked.
 */
static void ClientCheck(struct mqstats *pmqstats,
                   PMQAXP pExitParms,
                   PMQAXC pExitContext)
{
  pmqstats->Environment = pExitContext->Environment;
  return;
}
```

COMPILING THE SAMPLE EXIT

Here are Makefiles for several operating systems.

Solaris

```
all: apixstat
apixstat :  apixstat.c Makefile
   /opt/SUNWspro/bin/cc -KPIC -mt -o $@ $@.c \
            -L/opt/mqm/lib -dy -G -lmqmzf -lmqm -lmqmcs -lmqmzse
```

AIX

```
all: apixstat apixstat_r
apixstat :  apixstat.c Makefile
   $(CC) -e MQStart -o $@   apixstat.c $(LDOPT) \
   -L/usr/mqm/lib  -lmqmzf -lmqm
apixstat_r : apixstat.c Makefile
   cc_r -D_REENTRANT  -e MQStart -o $@   apixstat.c \
     -L/usr/mqm/lib -lmqmzf_r -lmqm_r
```

Linux

```
all: apixstat apixstat_r
LDOPT=-Wl,-rpath /opt/mqm/lib
apixstat :  apixstat.c Makefile
   $(CC) -fPIC -shared -e MQStart -o $@ apixstat.c \
   $(LDOPT) -L/opt/mqm/lib                          \
   -lmqmzf -lmqm -lmqmcs -lmqmzse -ldl -lc
apixstat_r : apixstat.c Makefile
   $(CC) -D_REENTRANT -D_POSIX_PTHREAD_SEMANTICS -fPIC  \
   $(LDOPT) -shared -e MQStart -o $@ apixstat.c -L/opt/mqm/lib \
     -lmqmzf_r -lmqm_r -lmqmcs_r -lmqmzse -ldl -lc -lpthread
```

CONFIGURING THE EXIT

First, create the APIX.STAT queue and set permissions on it so that anyone can put messages to it. The exit uses default message options so you might want to set the queue to default to persistent messages.

Then copy the compiled exit to the */var/mqm/exits* directory and add this stanza to the *qm.ini* file for your queue manager:

```
ApiExitLocal:
   Sequence=100
   Function=EntryPoint
   Module=/var/mqm/exits/apixstat
   Name=SampleApiExit
```

Finally, restart the queue manager. As applications run, you should see the depth of the APIX.STAT queue increase.

A simple test, running both clients and local applications, produced the following output from amqsget:

```
message <App:runmqsc      User:metaylor     Env:00000004 MsgPut:0
MsgGot:0 MsgBrw:0 BytesPut:0 BytesGot:0 BytesBrw: 0 MaxPut:0 MaxGot:0 >
message <App:ClientProxy  User:root         Env:00000002 MsgPut:2
MsgGot:0 MsgBrw:0 BytesPut:12 BytesGot:0 BytesBrw: 0 MaxPut:7 MaxGot:0>
message <App:ClientProxy  User:root         Env:00000002 MsgPut:27
MsgGot:0 MsgBrw:0 BytesPut:1857 BytesGot:0 BytesBrw: 0 MaxPut:81
MaxGot:0 >
message <App:ClientProxy  User:root         Env:00000002 MsgPut:1
MsgGot:0 MsgBrw:0 BytesPut:5 BytesGot:0 BytesBrw: 0 MaxPut:5 MaxGot:0>
message <App:amqsget      User:metaylor     Env:00000000 MsgPut:0
MsgGot:30 MsgBrw:0 BytesPut:0 BytesGot:1874 BytesBrw: 0 MaxPut:0
MaxGot:81 >
no more messages
Sample AMQSGET0 end
```

SUMMARY

The API Exit interface has been a requirement for some time and now that it is available it will make it much easier to write a variety of systems-level functions. It has been designed to give a lot of flexibility to exit-writers and we are looking forward to seeing new products that fit into this area.

Mark E Taylor
Technical Strategist, IBM Hursley (UK)

Loganal – an MQSeries error log analysis utility

INTRODUCTION

With all Unix and Windows NT/2000 MQSeries installations, the three sets of error logs are a principal source of diagnostic information concerning what has occurred within a queue manager. In addition to listing all the regular (and perfectly normal) activities, such as queue manager starts and stops and channel starts and stops, it will also contain entries for any error conditions that have been detected. This makes them an essential tool in the diagnosis of problems.

On Windows NT/2000 and Unix, error messages will be routed to specific log files based on the following conditions (directory structures shown are for NT).

- If the queue manager is available and its name is known, the destination will be that specific queue manager's logs. These will usually be located within *\var\mqm\qmgrs\<qmname>\errors*.
- If the queue manager name is known but the queue manager is not available, messages will be written to *\var\mqm\qmgrs\@SYSTEM\errors*.
- For errors within client applications the destination will be *\var\mqm\errors*.

Each of these locations will have a set of up to three error logs. Each log has a capacity of 256K, with *AMQERR01.LOG* always being the one that is actively being written in MQSeries. When *AMQERR01.LOG* is about to exceed 256K it is copied to *AMQERR02.LOG*. However, before this operation can complete, *AMQERR02.LOG* is in turn copied to *AMQERR03.LOG*. The previous contents of *AMQERR03.LOG* are discarded, and so the logs cycle.

A common problem when examining the logs is knowing which of them may contain the error you are searching for and where within that log file the message is located. You may want to search by date/time or perhaps you want to look for a specific error message whose code you already know.

Loganal is a utility which will help make this process a lot easier.

THE LOGANAL UTILITY

Loganal has a rich set of control parameters and entering the command **loganal -h** will display all the options that are available.

LOGONAL

```
loganal -h
loganal start    on: Ø5/Ø1/Ø2
                 at: 14:Ø9:53
MQ Error Log Analayis Utility
Usage: loganal [-options]
Options:
-h        Explain parameters
-c  file  Config File to get parameters from    (Default - none)
-d  dir   MQ log file directory           (Default - Current Directory)
-L1 file  MQ Error Log to analayse        (Default - AMQERRØ1.LOG)
-L2 file  Additional MQ Error Log to analyse     (Default - none)
-L3 file  Additional MQ Error Log to analyse     (Default - none)
          Note:
          If more than L1 is used - L1 must be
          the most recent log with L2 older and
          and L3 the oldest
-D  date  The date of log entries to check       (Default - today)
-T1 time  The start time of entries to analyse   (Default ØØ:ØØ:Ø1)
-T2 time  The end time of entries to analyse     (Default 23:59:59)
-s  code  Search for this AMQnnnn error          (Default - none)
-X1 code  Exclude this AMQnnnn error             (Default - none)
-X2 code  Exclude this AMQnnnn error             (Default - none)
-X3 code  Exclude this AMQnnnn error             (Default - none)
-S  s|d   Summary or Detail. s = Summary         (Default - d)
                             d = detail
```

Many of the parameters assume default values if none is provided by the user.

SOME POINTS TO BEAR IN MIND

Commonly used search options can be stored in a file and loganal invoked to read and use them. It does this via the -c parameter. An example of what that file might look like is shown below.

LOGANAL.CFG

```
-d=\temp
-L1=Amqerr01.log
-D=04/01/02
-T1=16:00:00
-T2=16:30:00
-s=AMQZ002
```

Use of this file would be invoked with the command **loganal -c loganal.cfg**.

Note that, if this option is used, it must be the only one passed on the command line. All other parameters must be specified within the file.

The date specified with the -D parameter must have a format matching that within the logs, ie if dates are written out as mm/dd/yy in the log file this is the format you must specify for your search.

If you want to search through multiple log files using the -L1, -L2, and -L3 options you must specify them in reverse order with the oldest file first. This is because of the way the utility searches through the files based upon the start and end times you specify. Thus, if you were searching through all three standard error logs, the command would look something like this:

```
loganal -L1 AMQERR03.LOG -L2 AMQERR02.LOG -L3 AMQERR01.LOG …….
```

THE OUTPUT

The output can be displayed either in detail format with one line for each error message within the log or as a summary of the number of occurrences of each AMQ error message type. Here are a couple of examples.

EXAMPLE ONE

```
>loganal -D 04/01/02 -T1 12:16:00 -T2 12:19:00
loganal start    on: 05/01/02
                 at: 14:48:29
SummaryDetail   d
Directory       .\
SearchDate      04/01/02
Start Time      12:16:00
End Time        12:19:00
Log 1           AMQERR01.LOG
************ Results ************
04/01/02  12:16:01 AMQ9002: Channel program started.
04/01/02  12:16:02 AMQ9002: Channel program started.
04/01/02  12:17:04 AMQ9208: Error on receive from host A164367N
```

```
(161.2.158.172).
Ø4/Ø1/Ø2  12:17:28 AMQ92Ø8: Error on receive from host A164367N
(161.2.158.172).
********** End Results **********
loganal End      on: Ø5/Ø1/Ø2
                 at: 14:48:29
```

EXAMPLE TWO

```
>loganal -D Ø4/Ø1/Ø2 -S s
loganal start    on: Ø5/Ø1/Ø2
                 at: 17:5Ø:58
SummaryDetail    s
Directory        .\
SearchDate       Ø4/Ø1/Ø2
Start Time       ØØ:ØØ:Ø1
End Time         23:59:59
Log 1            AMQERRØ1.LOG
************ Results ************
Error Code       Occurrences
AMQ9ØØ1              14
AMQ9ØØ2              14
AMQ92Ø8              4Ø
AMQ9545              14
********** End Results **********
loganal End      on: Ø5/Ø1/Ø2
                 at: 17:5Ø:58
>
```

THE SOURCE CODE (C)

Note: when compiling for Unix the variable MYENV should be set to ‘Unix’, and for Windows NT/2000 to ‘NT’.

```
 /* Program name: LOGANAL                                             */
 /* Description: Utility to analyse MQSeries Error Logs               */
 /*********************************************************************/
 /*   LOGANAL has many optional parameters ...                        */
 /*        - these are fully described in the "disphelp" function     */
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include <time.h>
 /* If you are on Unix - set MYENV to "unix", if on NT set it to "NT"*/
 #define MYENV "NT"
 void disphelp(void);
 void sorterrs(char *strings[], int num); int main(int argc, char
**argv) {    char     CurrTime[1Ø];     /* Hold Current Time         */
  char CurrDate[1Ø];         /* Hold Current Date                   */
 struct tm *newtime;         /* Structure for time fields           */
```

```
time_t long_time;           /* Time variable                          */
 int ParmCount;             /* Parameters passed                      */
 char GetConfig = 'n';      /* Parms from config file ?               */
 char SummaryDetail[2] = "d";  /* Summary or Detail report ?          */
 char ConfigFile[24] = "loganal.cfg"; /* Parameter config file        */
 char LogDir[80];    /      * MQ Error Log Directory                 */
 char SlashChar[2];         /* Directory Separator                    */
 char SearchDate[10];       /* Error Date to search for               */
 char SearchFor = 'n';      /* Search for specific error ?            */
 char SearchErr[10] = "AMQ0000";  /* AMQnnnn code to search for       */
 char LogErr[10];           /* AMQnnnn code found in log              */
 char OutputFileName[24] = "  ";  /* File for report output           */
 char OutputFile = 'n';     /* Report to output file ?                */
 char log1[24] = "AMQERR01.LOG";  /* Primary Error Log                */
 char log2[24] = "none";    /* Additional Error Log                   */
 char log3[24] = "none";    /* Additional Error Log                   */
 char LogFile[100];         /* Log File path/name                     */
 char StartTime[10] = "00:00:01"; /* Start Time to search for         */
 char EndTime[10]   = "23:59:59"; /* End Time to search for           */
 char FBuffer[100];         /* Buffer for reading files               */
 char HoldBuf[100];         /* Buffer for previous line               */
 int OutCtr = 0;            /* Number of errors found                 */
 int HoldCtr = 0;           /* Hold Counter                           */
 char HoldDate[10];         /* Hold Date                              */
 char HoldAMPM[4];          /* AM, PM or '  '                         */
 char HoldTime[10];         /* Hold Time                              */
 char WorkTime[10];         /* Temp work field                        */
 char SummErr[20][8] = {"xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx",
                                                        "xxxxxxx",
        "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx",
        "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx",
        "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx", "xxxxxxx"};
     /* Error Codes found                                          */
 int NextErr = 0;           /* Temp work field                        */
 static int ErrCnt[20];     /* Error Code occurrences                 */
 char *ptstr[20];           /* Temp array for sorting                 */
 char LineOut[20][80];      /* Temp array for sorting                 */
 int Exclude = 0;           /* Errors to exclude ?                    */
 char Excluded[3][8] = {"none", "none", "none"}; /* AMQnnnn codes to
                                                      exclude         */
 int Workhhi;               /* Working variable for time              */
 char Workhhs[2];           /* Working variable for time              */
 int LogIntTime;            /* Error's time as integer                */
 int StartIntTime;          /* Start time as integer                  */
 int EndIntTime;            /* End time as integer                    */
 char *token;               /* Work variable for strtok               */
 char *fparm1;              /* Work variable - config file            */
 char *fparm2;              /* Work variable - config file            */
 char fparm3[80] = "\0";    /* Work variable - config file            */
 int i = 1;                 /* Counter variable                       */
 int j,k;                   /* Counter variables                      */
 char blank[] = " ";        /* Constant                               */
```

```
 int pos;                  /* Work variable                          */
 int OK;                   /* Work variable                          */
 char TooLate = 'n';       /* Work variable                          */
 char HoldLog[3][24];      /* Work variable                          */
 FILE *fpc, *fpl, *fpout;  /* File pointers                          */
/* Get System Date and current Time */
 time( &long_time );
newtime = localtime( &long_time );
strftime(CurrDate, 10, "%d/%m/%y", newtime);
 CurrDate[8] = '\0';
strftime(CurrTime, 10, "%H:%M:%S", newtime);
 CurrTime[8] = '\0';
 /* Display Start Messages                                          */
 printf(" \n");
 printf("%s start \t on: %s \n", argv[0], CurrDate);
 strcpy(SearchDate, CurrDate);
 SearchDate[8] = '\0';
 printf ("    \t at: %s \n", CurrTime);
 printf(" \n");
 /* Set appropriate directory separator for Unix or NT              */
 if (memcmp(MYENV, "NT", 2) == 0)
 {
  strcpy(LogDir,  ".\\");
  SlashChar[0] = '\\';
  SlashChar[1] = '\0';
 }
 else if (memcmp(MYENV, "unix", 4) == 0)
 {
  strcpy(LogDir, "./");
 strcpy(SlashChar, "/");
 }
 else
 {
  printf("The setting of MYENV is invalid ... \n");
  printf("Exiting .... \n");
 }

 /* Read in all parameters passed and process ...                   */
 ParmCount = 1;
 while (argc > ParmCount)
 {
  if (ParmCount < argc && memcmp(argv[ParmCount], "-c", 2) == 0)
  {
   /* Parms in a config file - make sure this is the only           */
   /* parameter passed .....                                        */
   GetConfig = 'y';
   if ((memcmp(argv[ParmCount+1], "-", 1) ==0) ||
    (argc > 3))
   {
    printf("-c Option if used must be the only one ... \n");
    printf("Exiting ... \n");
    return 1;
```

```
 }
 strcpy(ConfigFile, argv[ParmCount+1]);
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-d", 2) == 0)
{
 /* Log File directory selected                                    */
 strcpy(LogDir, argv[ParmCount+1]);
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-D", 2) == 0)
{
 /* Search Date for errors selected                                */
 strcpy(SearchDate, argv[ParmCount+1]);
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-f", 3) == 0)
{
 /* Output File for results                                        */
 strcpy(OutputFileName, argv[ParmCount+1]);
 OutputFile = 'y';
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-h", 2) == 0)
{
 /* Display Help text                                              */
 if (argc > 2)
 {
  printf("Invalid parameters passed ... \n");
  printf("-h parameter must be on its own \n");
  return 9;
 }
 disphelp();
 return 0;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-L1", 3) == 0)
{
 /* Primary Log File to process                                    */
 strcpy(log1, argv[ParmCount+1]);
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-L2", 3) == 0)
{
 /* Second Log File to process                                     */
 strcpy(log2, argv[ParmCount+1]);
 ParmCount += 2;
}
if (ParmCount < argc && memcmp(argv[ParmCount], "-L3", 3) == 0)
{
 /* Third Log File if needed */
 strcpy(log3, argv[ParmCount+1]);
 ParmCount += 2;
```

```
   }
   if (ParmCount < argc && memcmp(argv[ParmCount], "-X1", 3) == Ø)
   {
    /* Exclude this specific AMQnnnn from the search                */
    if (SearchFor == 'y')
    {
     printf("Exclude (-Xn) and Search For (-s) cannot be used together
     ... \n");
     printf("Exiting ...... \n");
     return 1;
    }
    strcpy(Excluded[Ø], argv[ParmCount+1]);
    Exclude = 1;
    ParmCount += 2;
   }
   if (ParmCount < argc && memcmp(argv[ParmCount], "-X2", 3) == Ø)
   {
    /* Exlude second error code from search                         */
    if (SearchFor == 'y')
    {
     printf("Exclude (-Xn) and Search For (-s) cannot be used together
     ... \n");
     printf("Exiting ...... \n");
     return 1;
    }
    strcpy(Excluded[1], argv[ParmCount+1]);
    Exclude = 1;
    ParmCount += 2;
   }
   if (ParmCount < argc && memcmp(argv[ParmCount], "-X3", 3) == Ø)
   {
    /* Exclude third error code from search                         */
    if (SearchFor == 'y')
    {
     printf("Exclude (-Xn) and Search For (-s) cannot be used together
... \n");
     printf("Exiting ...... \n");
     return 1;
    }
    strcpy(Excluded[2], argv[ParmCount+1]);
    Exclude = 1;
    ParmCount += 2;
   }
   if (ParmCount < argc && memcmp(argv[ParmCount], "-s", 2) == Ø)
   {
    /* Search for a specific AMQnnnn code in the logs               */
    if (Exclude == 1)
    {
     printf("Exclude (-Xn) and Search For (-s) cannot be used together
     ... \n");
     printf("Exiting ...... \n");
     return 1;
```

```
  }
  SearchFor = 'y';
  strcpy(SearchErr, argv[ParmCount+1]);
  ParmCount += 2;
 }
 if (ParmCount < argc && memcmp(argv[ParmCount], "-S", 2) == 0)
 {
  /* Specify report level of detail                                  */
  strcpy(SummaryDetail, argv[ParmCount+1]);
  ParmCount += 2;
 }
 if (ParmCount < argc && memcmp(argv[ParmCount], "-T1", 3) == 0)
 {
  /* Start time for error search                                     */
  strcpy(StartTime, argv[ParmCount+1]);
  ParmCount += 2;
 }
 if (ParmCount < argc && memcmp(argv[ParmCount], "-T2", 3) == 0)
 {
  /* End time for error search                                       */
  strcpy(EndTime, argv[ParmCount+1]);
  ParmCount += 2;
 }
} /* End While Loop */
if (GetConfig == 'y')
{
/* Parameters are to be read from a file                            */
fpc = fopen(ConfigFile, "r");
if (fpc == NULL)
{
  printf("Error reading Config File %s ...\n", ConfigFile);
 printf("Aborting run ... \n");
 return 1;
}
printf("Reading parms from config file .... \n");
while (fgets(FBuffer, sizeof(FBuffer), fpc) != NULL)
{
 /* Extract line from config file by getting tokens                 */
 token = strtok(FBuffer, "=" );
 while(token != NULL)
 {
  /* While there are tokens in "FBuffer" */
  if (i == 1)
    fparm1 = token;
  else
    fparm2 = token;
  /* Get next token: */
  token = strtok( NULL, "=" );
  i++;
 }
  /* Remove trailing blanks from fparm2                              */
 pos = strcspn(fparm2, blank);
```

```
strncpy(fparm3, fparm2, (pos-1));
fparm3[pos-1] = '\0';
 /* Set all variables from  config file entries                    */
if (strcmp(fparm1,"-d")==0)
{
 strcpy(LogDir, fparm3);
}
if (strcmp(fparm1,"-D")==0)
{
 strcpy(SearchDate, fparm3);
}
if (strcmp(fparm1,"-f")==0)
{
 strcpy(OutputFileName, fparm3);
 OutputFile = 'y';
}
if (strcmp(fparm1,"-L1")==0)
{
 strcpy(log1, fparm3);
}
if (strcmp(fparm1,"-L2")==0)
{
 strcpy(log2, fparm3);
}
if (strcmp(fparm1,"-L3")==0)
{
 strcpy(log3, fparm3);
}
if (strcmp(fparm1,"-X1")==0)
{
 strcpy(Excluded[0], fparm3);
 Exclude = 1;
}
if (strcmp(fparm1,"-X2")==0)
{
 strcpy(Excluded[1], fparm3);
 Exclude = 1;
}
if (strcmp(fparm1,"-X3")==0)
{
 strcpy(Excluded[2], fparm3);
 Exclude = 1;
}
if (strcmp(fparm1,"-s")==0)
{
 SearchFor = 'y';
 strcpy(SearchErr, fparm3);
}
if (strcmp(fparm1,"-S")==0)
{
 strcpy(SummaryDetail, fparm3);
}
```

```
 if (strcmp(fparm1,"-T1")==Ø)
 {
  strcpy(StartTime, fparm3);
 }
 if (strcmp(fparm1,"-T2")==Ø)
 {
  strcpy(EndTime, fparm3);
 }
} /* End While(fgets ....                                          */
if (fclose(fpc) != Ø)
 printf("Error Closing Config File %s \n", ConfigFile);
} /* End if (GetCongif = = 'y') ...                                */
if (OutputFile == 'y')
{
 /* Output to be routed to a file ...                             */
 fpout = fopen(OutputFileName, "w");
 if (fpout == NULL)
 {
  printf("Error opening Output File %s ...\n", OutputFileName);
  printf("Aborting run ... \n");
  return 1;
 }
}
/* Echo parameters selected including those that have             */
/* taken their default values - either to screen or output file   */
printf("\n");
if (OutputFile == 'y')
{
  fprintf(fpout, "SummaryDetail \t%s \n", SummaryDetail);
 fprintf(fpout, "Directory \t%s \n", LogDir);
 fprintf(fpout, "SearchDate \t%s \n", SearchDate);
 if (OutputFile == 'y')
     printf("Output File \t%s \n", OutputFileName);
 fprintf(fpout, "Start Time \t%s \n", StartTime);
 fprintf(fpout, "End Time \t%s \n", EndTime);
 fprintf(fpout, "Log 1 \t\t%s \n", log1);
 if (memcmp(log2, "none", 4) !=Ø)
     fprintf(fpout, "Log 2 \t\t%s \n", log2);
 if (memcmp(log3, "none", 4) !=Ø)
     fprintf(fpout, "Log 3 \t\t%s \n", log3);
 if (memcmp(Excluded[Ø], "none", 4) != Ø)
     fprintf(fpout, "Excluded \t%s \n", Excluded[Ø]);
 if (memcmp(Excluded[1], "none", 4) != Ø)
     fprintf(fpout, "Excluded \t%s \n", Excluded[1]);
 if (memcmp(Excluded[2], "none", 4) != Ø)
     fprintf(fpout, "Excluded \t%s \n", Excluded[2]);
 fprintf(fpout, "\n************ Results ************\n\n");
}
else
{
  printf("SummaryDetail \t%s \n", SummaryDetail);
 printf("Directory \t%s \n", LogDir);
```

```
 printf("SearchDate \t%s \n", SearchDate);
 if (SearchFor == 'y')
     printf("Search For \t%s \n", SearchErr);
 printf("Start Time \t%s \n", StartTime);
 printf("End Time \t%s \n", EndTime);
 printf("Log 1 \t\t%s \n", log1);
 if (memcmp(log2, "none", 4) !=Ø)
     printf("Log 2 \t\t%s \n", log2);
 if (memcmp(log3, "none", 4) !=Ø)
     printf("Log 3 \t\t%s \n", log3);
 if (memcmp(Excluded[Ø], "none", 4) != Ø)
     printf("Excluded \t%s \n", Excluded[Ø]);
 if (memcmp(Excluded[1], "none", 4) != Ø)
     printf("Excluded \t%s \n", Excluded[1]);
 if (memcmp(Excluded[2], "none", 4) != Ø)
     printf("Excluded \t%s \n", Excluded[2]);
 printf("\n************ Results ************\n\n");
}
/* Read each Log File specified in turn ... */
strcpy(HoldLog[Ø], log1);
strcpy(HoldLog[1], log2);
strcpy(HoldLog[2], log3);
HoldCtr = Ø;
while ((memcmp(HoldLog[HoldCtr], "none", 4) != Ø) &
 (TooLate == 'n'))
{
 strcpy(LogFile, LogDir);
 strcat(LogFile, SlashChar);
 strcat(LogFile, HoldLog[HoldCtr]);
 fpl = fopen(LogFile, "r");
 if (fpl == NULL)
 {
  printf("Error reading Log File %s ...\n", LogFile);
  printf("Aborting run ... \n");
  return 1;
 }
/* Keep reading Log File until time is past search window          */
   while ((fgets(FBuffer, sizeof(FBuffer), fpl) != NULL) &
  (TooLate == 'n'))
{
 OK = 1;
 if (memcmp(FBuffer, "AMQ", 3) == Ø)
 {
  /* Previous line will contain the error's date and time          */
  strncpy(HoldDate, HoldBuf, 8);
  strncpy(HoldTime, HoldBuf+1Ø, 8);
  strncpy(HoldAMPM, HoldBuf+19, 2);
  HoldDate[8] = '\Ø';
  HoldTime[8] = '\Ø';
  HoldAMPM[2] = '\Ø';
  strncpy(LogErr, FBuffer, 7);
  LogErr[7] = '\Ø';
```

```
}
else
{
 OK = Ø;
 strcpy(HoldBuf, FBuffer);
}
/* Has this error been excluded ?                                    */
for (i=Ø; i < 3; i++)
{
 if (memcmp(LogErr, Excluded[i], 7) == Ø)
  OK = Ø;
}
if (OK == 1)
{
 /* Check Date and Time...                                           */
 if (memcmp(HoldAMPM, "PM", 2) == Ø)
 {
  strncpy(WorkTime, HoldTime+3, 2);
  strncpy(WorkTime+2, HoldTime+6, 2);
  WorkTime[4] = '\Ø';
  strncpy(Workhhs, HoldTime, 2);
  Workhhs[2] = '\Ø';
  Workhhi = atoi(Workhhs) + 12;
  LogIntTime = (atoi(WorkTime) + (Workhhi * 1ØØØØ));
 }
 else
 {
  strncpy(WorkTime, HoldTime, 2);
  strncpy(WorkTime+2, HoldTime+3, 2);
  strncpy(WorkTime+4, HoldTime+6, 2);
  WorkTime[6] = '\Ø';
  LogIntTime = atoi(WorkTime);
 }
 strncpy(WorkTime, StartTime, 2);
 strncpy(WorkTime+2, StartTime+3, 2);
 strncpy(WorkTime+4, StartTime+6, 2);
 WorkTime[6] = '\Ø';
 StartIntTime = atoi(WorkTime);
 strncpy(WorkTime, EndTime, 2);
 strncpy(WorkTime+2, EndTime+3, 2);
 strncpy(WorkTime+4, EndTime+6, 2);
 WorkTime[6] = '\Ø';
 EndIntTime = atoi(WorkTime);
 if ((memcmp(SearchDate, HoldDate, 8) == Ø) &
  (LogIntTime >= StartIntTime)  &
  (LogIntTime <= EndIntTime))
  OK = 1;
 else
  OK = Ø;
  /*  Passed Date and Time test ...                                  */
}
/* If we are on the correct date - are we past the time window ?  */
```

```
if ((memcmp(SearchDate, HoldDate, 8) == Ø) &
 (LogIntTime > EndIntTime))
 TooLate = 'y';
if ((OK ==1) &
 (SearchFor == 'y'))
{
 /* Search for specific error                                    */
 if (memcmp(LogErr, SearchErr, 7) == Ø)
  OK = 1;
 else
  OK = Ø;
}
if ((OK == 1) &
 (memcmp(SummaryDetail, "s", 1) == Ø))
{
 /* Summary report chosen ...                                    */
 /* Have we already seen this error ?                            */
 i = Ø;
 j = Ø;
 while ((memcmp(SummErr[i], "xxxxxxx", 7) != Ø) &
     (i < 2Ø))
 {
  if (memcmp(LogErr, SummErr[i], 7) == Ø)
  {
   ErrCnt[i]++;
   j = 1;
  }
  i++;
 }
 /* New error seen ... ?                                         */
 if (j == Ø)
 {
  strcpy(SummErr[NextErr], LogErr);
  ErrCnt[NextErr]++;
  OutCtr++;
  NextErr++;
 }
}
 /* Output for "Detail" report to screen                         */
if ((OK == 1) &
 (memcmp(SummaryDetail, "d", 1) == Ø) &
 (OutputFile == 'n'))
{
 HoldBuf[18] = ' ';
 HoldBuf[19] = '\Ø';
 strcat(HoldBuf, FBuffer);
 printf("%s", HoldBuf);
 OutCtr++;
}
/* Output for "Detail" report to file                            */
if ((OK == 1) &
 (memcmp(SummaryDetail, "d", 1) == Ø) &
```

```
  (OutputFile == 'y'))
 {
  HoldBuf[18] = ' ';
  HoldBuf[19] = '\Ø';
  strcat(HoldBuf, FBuffer);
  fprintf(fpout, "%s", HoldBuf);
  OutCtr++;
 }
} /* End while (fgets(FBuffer ...                                    */
if (fclose(fpl) != Ø)
  printf("Error Closing Log File %s \n", ConfigFile);
HoldCtr++;
} /* End while (memcmp(HoldLog[i], "none", 4 ....                    */
/* Summary Output ....                                               */
if ((memcmp(SummaryDetail, "s", 1) == Ø) &
 (OutCtr > Ø))
{
  for (i = Ø; memcmp(SummErr[i], "xxxxxxx", 7) !=Ø; i++)
 sprintf(LineOut[i], "%s %d", SummErr[i], ErrCnt[i]);
 for (k = Ø; k < 18; k++)
 {
  ptstr[k] = LineOut[k];
 }
 /* Sort the errors found                                            */
 sorterrs(ptstr, k);
 /* Output summary of errors to screen or a file                     */
 if (OutputFile == 'y')
 {
  fprintf(fpout, "Error Code \tOccurrences \n\n");
  for (k = Ø; k < 18; k++)
  {
   if (memcmp(ptstr[k], "AMQ", 3) == Ø)
   fprintf(fpout, "%s \n", ptstr[k]);
  }
 }
 else
 {
  printf("Error Code \tOccurrences \n\n");
  for (k = Ø; k < 18; k++)
  {
   if (memcmp(ptstr[k], "AMQ", 3) == Ø)
   printf("%s \n", ptstr[k]);
  }
 }
}
/* We didn't find any errors that matched the search parameters     */
if (OutCtr == Ø )
 printf("No matching log entries were found .... \n");
/* Output trailer block                                             */
if (OutputFile == 'y')
{
 fprintf(fpout, "\n********** End Results **********\n\n");
```

```
   if (fclose(fpout) != Ø)
    printf("Error Output File %s \n", OutputFileName);
  }
  else printf("\n********** End Results **********\n\n");
  printf("%s End \t on: %s \n", argv[Ø], CurrDate);
  strcpy(SearchDate, CurrDate);
  SearchDate[8] = '\Ø';
 time( &long_time );
 newtime = localtime( &long_time );
 strftime(CurrTime, 1Ø, "%H:%M:%S", newtime);
  CurrTime[8] = '\Ø';
  printf("    \t at: %s \n", CurrTime);
  printf(" \n");
  return Ø;
 } /* End Main                                                          */
 /* Display utility help                                               */
 void disphelp(void)
 {
   printf("MQ Error Log Analayis Utility\n\n");
 printf("Usage: loganal [-options]\n");
 printf("Options:\n");
 printf("-h    Explain parameters\n");
 printf("-d  dir   MQ log file directory\t\t\t (Default - Current
     Directory)\n");
 printf("-c  file  Config File to get parameters from\t (Default -
     none)\n");
 printf("-L1 file MQ Error Log to analayse\t\t (default -
     AMQERRØ1.LOG)\n");
 printf("-L2 file  Additional MQ Error Log to analyse\t (Default -
     none)\n");
 printf("-L3 file  Additional MQ Error Log to analyse\t (Default -
     none)\n");
 printf("  Note:\n");
 printf("  If more than L1 is used - L1 must be\n");
 printf("  the most recent log with L2 older and\n");
 printf("  and L3 the oldest\n");
 printf("-D  date  The date of log entries to check\t (Default -
today)\n");
 printf("-T1 time  The start time of entries to analyse\t (Default
ØØ:ØØ:Ø1)\n");
 printf("-T2 time  The end time of entries to analyse\t (Default
23:59:59)\n");
 printf("-s  code  Search for this AMQnnnn error\t\t (Default -
none)\n");
 printf("-X1 code  Exclude this AMQnnnn error\t\t (Default - none)\n");
 printf("-X2 code  Exclude this AMQnnnn error\t\t (Default - none)\n");
 printf("-X3 code  Exclude this AMQnnnn error\t\t (Default - none)\n");
 printf("-S  s|d   Summary or Detail. s = Summary\t (Default - d)\n");
 printf("    d = detail\n");
 printf(" \n");
 } /* End disphelp                                                      */
```

```
/* Sort the errors found                                          */
void sorterrs(char *strings[], int num)
{
 char *temp;
 int i, j;
 for (i = 0; i < num -1; i++)
 for (j = i + 1; j < num; j++)
  if (strcmp(strings[i], strings[j]) > 0)
  {
   temp = strings[i];
   strings[i] = strings[j];
   strings[j] = temp;
  }
}
/* End LOGANAL */
```

Chris Bell
Systems Consultant, British Airways (UK)

Exploring the MQSI V2 Neon nodes

Unless you intend to write your own parser, complex messages (ie messages which cannot be described as XML or fixed-format legacy) must be parsed using a Neon input format.

Transformation to a complex format is achieved using a Neon output format. In V2.0.2 some new Neon processing nodes have appeared in the list of IBM primitives and it is not immediately obvious which of these Neon nodes is relevant if a message flow needs to deal with complex messages.

The nodes which existed prior to MQSI V2.0.2 are:

- NeonFormatter.
- NeonRules.

The Neon nodes introduced in MQSI V2.0.2 are:

- NeonMap.
- NeonTransform.
- NeonRulesEvaluation.

If any Neon node is used in a V2 message flow the broker must have access to the Neon database via an ODBC connection. This is achieved by means of a configuration file made available via a system variable.

The main difference between the two sets of nodes is that the NeonFormatter and NeonRules nodes use the MQSI_PARAMETERS_FILE system variable, which points to an equivalent of the *MQSIRuleng.mpf* file.

The NeonMap, NeonTransform, and NeonRulesEvaluation nodes use the new NN_CONFIG_FILE_PATH system variable, which points to the directory containing the *neonreg.dat* configuration file.

Samples exist in the installation files that indicate the information required in these configuration files.

HOW IT USED TO BE PRIOR TO MQSI V2.0.2.

Prior to MQSI V2.0.2 there was no NEONMSG parser available. The parser that handled Neon messages was very limited. The logical message model, for example, could not be presented to a computer node for the message flow developer to access individual fields within the data hierarchy as it could for, say, an XML message or a message parsed using an MRM definition.

The only way this could reasonably be achieved was by defining a Neon Output Format which exactly matched either a legacy definition in the MRM or a well-defined XML message. So a Neon transformation would be used to alter the data to legacy or XML, then a ResetContentDescriptor node would be used effectively to reparse the newly transformed message.

In this way the Neon transformation acted a little like an adaptor, which simply altered the message to a format that could be understood by either the XML parser or a parser created from an MRM definition. The message could then be presented to a compute node, making available all the fields within the logical message model.

Example

The MQInput node refers to a Neon input format PersonIn.

The incoming Neon message is:


```
John;Smith;37
```

A NeonFormatter node refers to a target format of, say, PersonOut.Legacy or PersonOut.XML

The NeonFormatter node changes the message to either a legacy format:

```
John   Smith   Ø37
```

or an XML format:

```
<PersonIn>
<FirstName>John</FirstName>
<LastName>Smith</LastName>
<Age>37</Age>
</PersonIn>
```

The next node, a ResetContentDescriptor node, specifies a domain of MRM and a message set and message identifier which reflect the legacy layout or simply a domain of XML. When the next compute node is reached the message will be reparsed using the chosen domain (MRM or XML). The logical message model is now available to a compute node, which can refer to individual data items, for example:

```
  SET OutputRoot.XML.Message.CustomerName =
InputBody.PersonIn.LastName;
```

This is a very cumbersome means of utilizing Neon formats and requires extra Neon formats and possibly additional MRM definitions simply to access data within the message.

THE SITUATION WITH MQSI V2.0.2

In MQSI V2.0.2 the NEONMSG domain was introduced, which went one step further than the previous NEON domain. The NEONMSG parser creates a logical message node from the Neon input message – a hierarchy of data items – which is accessible to the compute node or any other node that needs to refer to the individual data items in the message body.

THE NEW NEON NODES

One of the things to remember about Neon is that any transformation involves an input format and an output format. Transformations take place on bit streams so messages are constructed before entering the node and

deconstructed after leaving the node.

When you configure the nodes below, the message must already have been parsed. This is often done via the MQInput node, specifying a domain name of NEONMSG and the name of the input format, which describes the incoming bit stream.

When you configure one of the new Neon nodes you must specify the name of an input format to use next time you want the message to be parsed. This option is referred to, confusingly, as the 'Output Message Type' in the properties box. More specifically, this means the format of the message leaving (output from) the node, which must be in an appropriate Neon input format. Where a transformation is involved the Neon output format is referred to as the target format.

This can cause problems when developing a message flow when you name an 'Output Message Type' that belongs to another broker. If you try to trace the message content the trace node will need to parse the bit stream and will be unable to find the input format.

The NeonTransform Node

This is a new node which allows the message to be transformed using the new NEONMSG domain. The wire format of the data is altered in accordance with the input and output formats in the Neon database.

So, in our previous example, if the output format changed the data to upper case and suffixed the data with a hash sign, it would appear as follows:

```
(0x1000000)NEONMSG    = (
                (0x7000000)MessageType = 'MsgInNew'
                (0x7000000)Format      = 'Wireformat'
                (0x3000000)Firstname   = 'JOHN#'
                (0x3000000)Lastname    = 'SMITH#'
                (0x3000000)Age   = '29#'
```

and the output bit stream would contain:

```
JOHN#SMITH#29#
```

By specifying an input format name in the Output Message Type property field you are telling the message flow how to parse the message next time the data is to be examined.

The NeonMap Node

The main difference between a NeonMap node and the NeonTransform node is that the data itself is not altered. The operations performed by the Neon Output Control components are not applied to the data, eg suffix, prefix, maths expressions, etc.

So in the above example, even if the output format changed the data to upper case and suffixed the data with a hash sign, it would appear as follows:

```
(0x1000000)NEONMSG    = (
                (0x7000000)MessageType = 'MsgInNew'
                (0x7000000)Format      = 'Wireformat'
                (0x3000000)Firstname   = 'John'
                (0x3000000)Lastname    = 'Smith'
                (0x3000000)Age   = '29'
```

If the message were written to an MQOutput node at this stage the output bit stream would contain:

```
JohnSmith29
```

Neon Map nodes are useful when the data items are required by subsequent processing nodes. The control fields (delimiters, tags, etc) are removed and the data presented to the message flow for manipulation by compute nodes, etc. Using the NeonMap node the data can be output in XML if required.

The NeonRules Node and NeonRulesEvaluation Node

These nodes are only required in order to migrate your existing V1 rules engines into a V2.2 message flow while still allowing your content-based routing and transformation to be performed by your V1 rules.

If you have no existing MQSI V1 environment, ie no existing V1 rules engine, then you have no need for either of these nodes. All your content-based routing can be performed by your new V2 message flows.

The main difference between the two nodes is the configuration file used to access the database as described above. Other differences relate to the properties and message flow behaviour and are not discussed in this article.

Ken Marshall
MQSeries Consultant
MQSolutions (UK)

MQ news

Promenix has recently announced services for migrating to the latest versions of WebSphere MQ Integrator, aimed at sites using older generation integration broker technology.

The new services are geared specifically to companies that are currently using an integration broker from a vendor that has declared the version they are running as being end-of-life. Such a vendor is Sybase, which has announced the end-of-life for several New Era of Networks products, including MQ Integrator and NEONet.

The migration services include project management, delta training, migration analysis, product functionality analysis, architecture and design, interface development, gap development, and go-live and post-production support.

It also provides these services for sites wanting to migrate from IBM's MQSI 1.1 and 2.0.2. to WebSphere MQ Integrator 2.1.

For further information contact:
Promenix, 130 Commons Court, Chadds Ford, PA 19317, USA.
Tel: +1 610 361 1560.
Fax: +1 610 361 7549.
Web: http://www.promenix.com.

* * *

Mackinney Systems has released Version 5.4 IBM has recently announced new WebSphere infrastructure software, encompassing WebSphere Application Server, MQ, Business Integration, Portal, and 'enterprise modernization' software.

New WebSphere MQ software includes the new WebSphere MQ Event Broker, which allows businesses to publish information to specific subscribers according to their unique preferences. According to IBM, it delivers real-time, highly personalized information across the network, Internet, or pervasive devices, to tens of thousands of users around the world.

IBM WebSphere Business Integration Version 4.1 is a new offering that includes technology acquired from CrossWorlds Software earlier this year, and MQ technologies. The product is designed to assist companies in automating business processes that integrate multiple applications, ie managing customer relationships and supply chains.

Additional products included in the announcement were WebSphere Application Server Version 5, WebSphere Portal Version 4.1, WebSphere 'enterprise modernization' tools to help customers reuse existing software, and data management portfolio enhancements.

For more information contact your local IBM representative.
Web: http://www.ibm.com/websphere.

* * *