10 MQ update

April 2000

In this issue

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: +44 1635 550955
e-mail: HarryL@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

MQ Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com/mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Editor

Harry Lewis

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 ($33.50) each including postage.

An MQSeries PLTPI

When we migrated from MQSeries for Windows NT 5.0 to Version 5.1 we encountered a small problem: we previously ran Microsoft SNA Server's TPStart program using a start-up file that was registered using the MQSeries **scmmqm** command. However, MQSeries 5.1 doesn't include **scmmqm**. Thus, while the migration process ensures that most commands that were run from the start-up file are automatically started when the queue manager starts, TPStart is an exception. Also, we could not solve our problem using the Windows *Startup* folder as a user is not always logged on to our MQSeries servers.

My first attempt at a solution was to try to run TPStart as an NT service. However, this program is obviously not designed to run as a service as event messages were logged notifying that the TPStart service did not respond to the *start* or *control* functions in a timely fashion.

We then developed a simple C program that executes a list of commands contained in messages in the program's trigger queue. The queue and process definitions ensure that the program is run only once after MQSeries starts and is not run again unless additional messages are put on the queue. For this reason, CICS programmers will understand why the program is called 'MQPLTPI'.

At present, the only command message in the queue is 'START TPSTART'. However, other commands could be added, if required.

Below are sample queue and process definitions for MQPLTPI.

SAMPLE QUEUE DEFINITION

```
DEFINE QLOCAL ('PLTPI.QUEUE') +
       DESCR(' ') +
       PUT(ENABLED) +
       DEFPRTY(0) +
       DEFPSIST(YES) +
       SCOPE(QMGR) +
       GET(ENABLED) +
       MAXDEPTH(5000) +
```

```
       MAXMSGL(4194304) +
       SHARE +
       DEFSOPT(SHARED) +
       MSGDLVSQ(PRIORITY) +
       HARDENBO +
       USAGE(NORMAL) +
       TRIGGER +
       TRIGTYPE(EVERY) +
       TRIGDPTH(1) +
       TRIGMPRI(0) +
       TRIGDATA(' ') +
       PROCESS('PLTPI.PROCESS') +
       INITQ('TEST.INITIATION.QUEUE') +
       RETINTVL(999999999) +
       BOTHRESH(0) +
       BOQNAME(' ') +
       QDEPTHHI(80) +
       QDEPTHLO(20) +
       QDPMAXEV(ENABLED) +
       QDPHIEV(DISABLED) +
       QDPLOEV(DISABLED) +
       QSVCINT(999999999) +
       QSVCIEV(NONE) +
       DISTL(NO) +
       CLUSTER(' ') +
       CLUSNL(' ') +
       DEFBIND(OPEN) +
       REPLACE
```

SAMPLE PROCESS DEFINITION

```
DEFINE PROCESS ('PLTPI.PROCESS') REPLACE +
       DESCR('Programs to run at MQ startup') +
       APPLTYPE(WINDOWSNT) +
       APPLICID('C:\MQpltpi\PLTPI.exe >C:\MQpltpi\PLTPI.log') +
       USERDATA(' ') +
       ENVRDATA(' ')
```

A few points to note are:

- An initiation queue (*TEST.INITIATION.QUEUE* in this case) that has a trigger monitor running is also needed.
- The queue needs to be defined with *TRIGTYPE(EVERY)*, as the triggered program doesn't perform destructive *GET*s. Instead, it browses the queue so that commands can be run again the next time the queue manager is started. If we had used either *TRIGTYPE(FIRST)* or *DEPTH* in the queue definition, MQSeries

would issue another trigger message each time the program closes the queue, as the queue would not be empty.

- The process's *APPLICID* specifies that output should be directed to a log file to assist in debugging.

The source code of the C program is listed below.

PLTPI.C

```
/*************************************************************************/
/*                                                                       */
/* Function:                                                             */
/*                                                                       */
/*   This program browses messages on a queue and issues commands        */
/*   to the system using the contents of this queue. It's intended       */
/*   to run at MQSeries start-up as a triggered program, and, as         */
/*   such, performs a similar function to CICS's PLTPI.                  */
/*                                                                       */
/*   The program:                                                        */
/*                                                                       */
/*   - Gets messages from a queue whose name is in the trigger           */
/*     parameter. The content of the message is the command to           */
/*     be issued.                                                        */
/*                                                                       */
/*   - Stops when the end of the input queue is reached.                 */
/*                                                                       */
/*   - Stops if any MQ error is detected.                                */
/*                                                                       */
/*  Program logic:                                                       */
/*     MQCONNect to message queue manager                                */
/*     MQOPEN message queue for shared input                             */
/*     while no MQI failures:                                            */
/*     - MQGET (with browse option) next message in input queue          */
/*     - Prepare command if GET is successful                            */
/*     - Issue command to system                                         */
/*     MQCLOSE queue                                                     */
/*     MQDISConnect from queue manager                                   */
/*                                                                       */
/*  PLTPI has one parameter - a string (MQTMC2) based on the             */
/*  initiation trigger message. Only the QName and queue manager         */
/*  name fields are used.                                                */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
/*  includes for MQI  */
 #include <cmqc.h>

 int main(int argc, char **argv)
 {

   /*   Declare MQI structures needed                                 */
   MQOD    odG = {MQOD_DEFAULT};    /* Object descriptor for GET     */
   MQMD     md = {MQMD_DEFAULT};    /* Message descriptor            */
   MQGMO   gmo = {MQGMO_DEFAULT};   /* GET message options           */
   MQTMC2   *trig;                  /* Trigger message structure     */

   MQHCONN  Hcon;                   /* Connection handle             */
   MQHOBJ   Hobj;                   /* Object handle                 */
   MQLONG   O_options;              /* MQOPEN options                */
   MQLONG   C_options;              /* MQCLOSE options               */
   MQLONG   CompCode;               /* Completion code               */
   MQLONG   Reason;                 /* Reason code                   */
   MQBYTE   buffer[256];            /* Message buffer                */
   MQLONG   buflen;                 /* Buffer length                 */
   MQLONG   messlen;                /* Message length received       */

   printf("PLTPI start\n");

   if (argc < 2)
   {
      printf("Missing parameter - start program by MQI trigger\n");
      exit(99);
   }

   /*****************************************************************/
   /*                                                               */
   /*   Set the program argument in the trigger message             */
   /*                                                               */
   /*****************************************************************/
   trig = (MQTMC2*)argv[1];        /* -> trigger message            */

   /*****************************************************************/
   /*                                                               */
   /*   Connect to queue manager                                    */
   /*                                                               */
   /*****************************************************************/
   MQCONN(trig->QMgrName,          /* Queue manager                 */
          &Hcon,                   /* Connection handle             */
          &CompCode,               /* Completion code               */
          &Reason);                /* Reason code                   */

   /* Report reason if any and stop  */
   if (Reason != MQRC_NONE)
   {
```

```
   printf("MQCONN ended with reason code %ld\n", Reason);
   exit(Reason);
}

/*****************************************************************/
/*                                                               */
/*   Open the message queue for shared input                     */
/*                                                               */
/*****************************************************************/
memcpy(odG.ObjectName,          /* Name of input queue           */
       trig -> QName, MQ_Q_NAME_LENGTH);
O_options = MQOO_INPUT_SHARED         /* Open queue for shared    */
          + MQOO_BROWSE               /* input with browse        */
          + MQOO_FAIL_IF_QUIESCING;   /* but not if MQM stopping  */
MQOPEN(Hcon,                    /* Connection handle             */
       &odG,                    /* Object descriptor for queue   */
       O_options,               /* Open options                  */
       &Hobj,                   /* Object handle                 */
       &CompCode,               /* MQOPEN completion code        */
       &Reason);                /* Reason code                   */

/* Report reason if any and stop     */
if (Reason != MQRC_NONE)
{
   printf("MQOPEN ended with reason code %ld\n", Reason);
   exit(Reason);
}


/*****************************************************************/
/*                                                               */
/*   Get messages from the message queue                         */
/*   Loop until there is a warning or failure                    */
/*                                                               */
/*****************************************************************/
buflen = sizeof(buffer) - 1;
while (Reason == MQRC_NONE)
{
   gmo.Options = MQGMO_BROWSE_NEXT
               + MQGMO_NO_WAIT; /* no wait                       */

   /**************************************************************/
   /*                                                            */
   /*   In order to read the messages in sequence, MsgId and     */
   /*   CorrelID must have the default value. MQGET sets them    */
   /*   to the values in the message it returns, so re-initialize */
   /*   them before every call.                                  */
   /*                                                            */
   /**************************************************************/
   memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
```

```
   memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));

   MQGET(Hcon,                /* Connection handle                */
         Hobj,                /* Object handle                    */
         &md,                 /* Message descriptor               */
         &gmo,                /* GET options                      */
         buflen,              /* Buffer length                    */
         buffer,              /* Message buffer                   */
         &messlen,            /* Message length                   */
         &CompCode,           /* Completion code                  */
         &Reason);            /* Reason code                      */

   /*  Report reason, if any, and stop  */
   if (Reason != MQRC_NONE)
   {
      printf("MQGET ended with reason code %ld\n", Reason);
   }
   else
   /*  Issue command  */
   {
      buffer[messlen] = '\0';  /* end string ready to use */
      printf("%s\n", buffer);
      system(buffer);
   }
}     /*  End GET message loop  */

/*****************************************************************/
/*                                                               */
/*   Close queue when no messages left                           */
/*                                                               */
/*****************************************************************/
C_options = 0;                  /* No close options             */
MQCLOSE(Hcon,                   /* Connection handle            */
        &Hobj,                  /* Object handle                */
        C_options,
        &CompCode,              /* Completion code              */
        &Reason);               /* Reason code                  */

/*  Report reason, if any  */
if (Reason != MQRC_NONE)
{
   printf("MQCLOSE ended with reason code %ld\n", Reason);
}

/*****************************************************************/
/*                                                               */
/*   Disconnect from MQM                                         */
/*                                                               */
/*****************************************************************/
MQDISC(&Hcon,                   /* Connection handle            */
```

```
        &CompCode,                /* Completion code               */
        &Reason);                 /* Reason code                   */

  /* report reason, if any     */
  if (Reason != MQRC_NONE)
  {
     printf("MQDISC ended with reason code %ld\n", Reason);
  }

  /*****************************************************************/
  /*                                                               */
  /* END OF PLTPI                                                  */
  /*                                                               */
  /*****************************************************************/
  printf("PLTPI processing ended\n");
  return(0);
}
```

Eric Judd
Technical Consultant
Metropolitan Life (South Africa)

Web-enabling legacy applications with MQSeries

For the past 40 years, programmers have written mainframe code for data processing, number crunching, and maintaining billions of records of mission-critical data. To a young Web developer like myself, this code and the systems it runs on have always been referred to as 'legacy'. However, the term 'legacy' is not entirely fair considering the number of large companies that depend on it. (Not to mention how important it is to my own work.) The most important aspect of many companies' Web development initiative today is providing the ability to access mission-critical 'legacy' applications over the Internet.

My experience is that many companies expect this to be the direction in which Web development will of necessity go. For instance, many financial institutions now have some form of Internet banking service,

while on-line shopping is resulting in price wars in many sectors, as price comparisons are now just a finger-click away.

However, regardless of industry sector, and of type of process being Web-enabled and back-end system being accessed, the factors that drive the decision on what connectivity to use are performance and reliability. In this article I have laid out the argument for using MQSeries to provide Web connectivity and have also furnished some views on how this is accomplished.

RELIABILITY

MQSeries has been a part of many large company's infrastructure for a long time. It provides a reliable architecture for the transfer of data from one system to another. The importance of this needs no explanation, especially to *MQ Update*'s audience, but what is often overlooked is that this reliability is part of MQSeries' run-time architecture and is independent of the applications that are using it.

A queue manager running on a server, such as a Web server, runs as an independent program and so doesn't affect the server, even if it's frequently stopped and restarted. Moreover, the impact of this is very modest, as only minimal resources are used for establishing a connection to MQ and sending and receiving messages. The value of this is easily realized by implementing an application that is capable of starting an MQ connection, testing it, detecting a failure, and (on detection of a failure) stopping and restarting the queue manager to fix the problem and try again. MQ's open architecture affords much potential for providing failover redundancy by using multiple queue managers and channels that connect to multiple back-end systems from more than one Web server. As this is the standard way an MQ system is meant to be deployed, it can handle this architecture extremely well. The amount of reliability provided by the MQ connection then depends on how comprehensive a system of redundancy the administrator is willing to build.

SESSION VERSUS NON-SESSION

When a browser sends a request to a Web server, the connection is

referred to as ‘session-less’. Each request the browser makes is effectively a new session that ends when the request is fulfilled. This is in stark contrast to standard 3270-type terminal connections that are traditionally used to connect to mainframe systems, and this difference can pose a serious technical problem when it comes to providing a Web-based interface to an application on a mainframe system.

The session-less nature of the Web browser connection makes MQSeries an excellent match for connecting Web applications to back-end systems. MQ sends messages and receives them independently in an asynchronous manner. The process that sends the message is independent of the process that receives it, and no session needs to be maintained between the sending and receiving applications. Therefore, while processing an update to data on a back-end system over the Web, the connection to the system that holds the appropriate record need not be maintained. This is a critical component of Web access to legacy systems, as a number of things can go wrong between requesting and updating data. It’s important that the data is not affected if the machine running the browser crashes or the user updating the data gets bored and decides to surf the Web for deals on a new car.

There are, however, issues that must be dealt with related to the session-less nature of the connection. Transactions on the back-end system have to be designed for session-less connections. This means that they should not require multiple independent inputs from a user, or, if they do, the transaction must provide both rollback and timeout functionality to handle the possibility of the user aborting in the middle of a multi-screen transaction.

A more serious problem occurs when updating information that first passes to the browser and is then edited and returned to the back-end system. The problem is that the record being edited cannot be locked, as the user may decide not to submit the changes, which may result in the record never being released. However, if the record is not locked, then two people could edit it at the same time, causing one set of changes to be lost. Fortunately, there are simple solutions to this. For instance, when the information is brought to the browser, two copies of it are stored – one of them is edited and the another is stored in its

original form. The updated version is submitted to the Web server along with the original, and the original is compared to the record currently on file and, if it matches, the update is performed. This functionality is critical for Web-based transactions, as it can easily take several minutes to do an update (several hours, if the user happens to go for a coffee or lunch in the middle of an update).

CLIENT VERSUS SERVER

While the decision to use MQSeries is an easy one, the decision to use an MQ client or MQ server at the Web server side of the connection is not so simple. The are advantages to both and the decision depends on a number of factors, such as cost, type of transport layer, functionality required, and the configuration of the Web server or Web farm. I have found that, from a performance standpoint, the choice of client versus server doesn't make a noticeable difference. However, it could make an indirect difference to performance depending on a number of other factors that are dealt with in the sections below.

Using the client component for connectivity at the Web server provides cost benefits, ease of set-up and maintenance, and (potentially) better performance for the server as a whole. The cost issue is simple, as acquiring MQ Server for every Web server quickly drives up software costs. By contrast, a large Web server cluster using multiple clients requires only one MQ server to be set up, thus reducing cost. Set-up and maintenance is also easier in a clustered environment, as all the MQ set-up is done on one server and clients are simply directed to it. Web server performance can also be enhanced, particularly if it uses MQ for only a small percentage of its workload. The MQ client is a relatively dormant component that uses few resources when it is not being accessed. In general, the client is less intrusive on the system.

The MQ server option, while more costly and difficult to implement and maintain than the client option, is better for Web servers that use MQ heavily. The main reason for this is simply that MQ Server is a server. This allows the Web server to use applications that are triggered when a message is received. The potential uses of this

include dynamic updates to Web pages, automated maintenance of administrative data, and Web server monitoring.

An MQ server can also be used as a dynamic local database system in which data can easily be stored and shared without taxing other critical Web systems that may already be under a heavy load. An example of this is a back-end process that runs at night and sends update messages to queue on a Web server that maintains customer statements. When customers query their statements the following day, they're already on the server and can be accessed quickly. While a relational database can be used to achieve this, using the system described above provides a performance advantage – one that increases if the database is under heavy use by other dynamic components of Web applications. In this case the MQ Server provides a simple functional option that would require the purchase of another database server to handle the additional load.

DISTRIBUTED PROCESSING

As Web applications become more complex and server loads increase, so the need to distribute the load over clustered systems becomes critical. A number of options are available for clustering servers, the options being dependent on reliable middleware for handling connectivity. MQ provides an ideal solution for multi-tier applications in clustered Web environments. The key benefits of MQ in this environment are its minimal overhead, its constantly active connection, and its ability to trigger applications.

In a large Web farm, dedicated servers can be set up to handle specific tasks. Thus, a CPU-intensive process in an application can be hived off to a separate server. This set-up also allows advanced management based on the 'unit of work' – a technique that can be used to guarantee accurate results when managing complex applications that run over multiple servers. The increasing loads on today's systems is already driving the need for clustering at server farms. New technologies, like Enterprise Java Beans, provide additional motivation to move towards distributed processing. MQ provides the connectivity in such complex environments.

FUTURE CONSIDERATIONS

E-commerce is expanding in every industry – a process that's being driven by companies' need to keep up with the competition. As Internet-based back-end applications grow in number and complexity, so does the load on back-end systems. A typical mainframe accessed via 3270 terminals will handle only a few hundred users on any given day. The addition of Web-based access can increase load several hundred fold. MQSeries provides an excellent means of connecting these systems, as it controls the processor load on the mainframe by controlling the transaction triggering process. The important thing to bear in mind is that the load on the system will eventually increase beyond the system's capacity to handle it. While the front-end Web and application servers can be clustered and upgraded relatively inexpensively, back-end mainframe system upgrades are considerably more expensive. It has been my experience that MQSeries allows this architecture to be used efficiently and flexibly.

Rich O'Neill (rich@dwl.com)
Solutions Architect
DWL Incorporated (Canada)

Invoking MQSeries tools using ISPF panels

This month's instalment concludes this article on using ISPF panels to invoke MQSeries utilities (the first part of the article appeared in last month's issue).

MQDEFINE

```
)ATTR
@ TYPE(OUTPUT) INTENS(HIGH) CAPS(OFF)  JUST(LEFT)
$ TYPE(INPUT) INTENS(LOW)  PAD(_)
% TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(GREEN)
¢ TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(red)
~ TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(WHITE)
? TYPE(TEXT)   INTENS(LOW)  JUST(ASIS) COLOR(YELLOW)
```

```
# TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(YELLOW)
` TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(green) HILITE(REVERSE)
| TYPE(TEXT)   INTENS(high)  JUST(ASIS) COLOR(blue)
+ TYPE(TEXT)   INTENS(LOW) color(white)
_ TYPE(INPUT)  INTENS(LOW)
)BODY
|------------------------------`MQSeries Utility Panel|----------------------
%COMMAND ===>_ZCMD
+
+
+   MQSeries system id   ....$z   +
+
+
+
+
+
+
+   PF3 = Exit
+
)INIT
  .ZVARS = 'SYSID'
  .HELP = TUTORPAN           /* Insert name of tutorial panel    */
    &sysid=' '
    &pfkey=.pfkey
)PROC
    VER (&SYSID,NB,MSG=mqmsg001)
    &pfkey=.pfkey
)END
```

MQMCOMM

```
)ATTR
@ TYPE(OUTPUT) INTENS(HIGH) CAPS(OFF)  JUST(LEFT)
$ TYPE(INPUT) INTENS(LOW)  PAD(_)
% TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(GREEN)
¢ TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(red)
~ TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(WHITE)
? TYPE(TEXT)   INTENS(LOW)  JUST(ASIS) COLOR(YELLOW)
# TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(YELLOW)
` TYPE(TEXT)   INTENS(HIGH) JUST(ASIS) COLOR(green) HILITE(REVERSE)
| TYPE(TEXT)   INTENS(high)  JUST(ASIS) COLOR(blue)
+ TYPE(TEXT)   INTENS(LOW) color(white)
_ TYPE(INPUT)  INTENS(LOW)
)BODY
|------------------------------`MQSeries Utility Panel|----------------------
%COMMAND ===>_ZCMD
+
+
```

```
+   MQSeries system id   ....$z   +
+
+
+   MQSeries command     ....$comm1                                +
+   MQSeries command     ....$comm2                                +
+   MQSeries command     ....$comm3                                +
+   MQSeries command     ....$comm4                                +
+
+
+
+
+   PF3 = Exit
+
)INIT
  .ZVARS = 'SYSID'
  .HELP = TUTORPAN           /* Insert name of tutorial panel    */
    &sysid=' '
    &COMM1=' '
    &COMM2=' '
    &COMM3=' '
    &COMM4=' '
    &pfkey=.pfkey
)PROC
    VER (&SYSID,NB,MSG=MQMSG001)
    VER (&COMM1,NB,MSG=MQMSG001)
    &pfkey=.pfkey
)END
```

MQSERIES UTILITY EXEC DEFINITIONS

The following EXECs are invoked:

MQMPRLM

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
DO
"DISPLAY PANEL (MQMPRLM)"
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQMPRLM'
'FTCLOSE NAME(MQMPRLM)'
"EDIT DATASET('your.jcl.lib(MQMPRLM)') PANEL(SUBMIT)"
```

```
END
RETURN
```

MQMCOPY

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
"DISPLAY PANEL (MQMCOPY)"
PS = PSN
QUE = QUEUE
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQMCOPY'
'FTCLOSE NAME(MQMCOPY)'
"EDIT DATASET('your.jcl.lib(MQMCOPY)') PANEL(SUBMIT)"
RETURN
```

MQMEMPTY

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
"DISPLAY PANEL (MQMEMPTY)"
PS = PSN
QUE = QUEUE
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQMEMPTY'
'FTCLOSE NAME(MQMEMPTY)'
"EDIT DATASET('your.jcl.lib(MQMEMPTY)') PANEL(SUBMIT)"
RETURN
```

MQMLOAD

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
```

```
"DISPLAY PANEL (MQMLOAD)"
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQMLOAD'
'FTCLOSE NAME(MQMLOAD)'
"EDIT DATASET('your.jcl.lib(MQMLOAD)') PANEL(SUBMIT)"
RETURN
```

MQPRINTQ

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
"DISPLAY PANEL (MQPRINTQ)"
PS = PSN
QUE = QUEUE
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQPRINTQ'
'FTCLOSE NAME(MQPRINTQ)'
"EDIT DATASET('your.jcl.lib(MQPRINTQ)') PANEL(SUBMIT)"
RETURN
```

MQDEFINE

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
"DISPLAY PANEL (MQDEFINE)"
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQDEFINE'
'FTCLOSE NAME(MQDEFINE)'
"EDIT DATASET('your.jcl.lib(MQDEFINE)') PANEL(SUBMIT)"
RETURN
```

MQMCOMM

```
/* REXX */
ADDRESS TSO
"ALLOC F(ISPFILE) DA('your.jcl.lib') SHR REUSE"
```

```
USERID=USERID()
ADDRESS ISPEXEC
"LIBDEF ISPSLIB DATASET ID('your.skel.lib')"
"ADDPOP ROW(2) COLUMN(10)"
"DISPLAY PANEL (MQMCOMM)"
DO WHILE (PFKEY¬=PF03)
SYS='MQM'||SYSID
'FTOPEN'
'FTINCL MQMCOMM'
'FTCLOSE NAME(MQMCOMM)'
"EDIT DATASET('your.jcl.lib(MQMCOMM)') PANEL(SUBMIT)"
"ADDPOP ROW(2) COLUMN(10)"
"DISPLAY PANEL (MQMUTIL)"
END
RETURN
```

MQSERIES UTILITY SKELETON DEFINITIONS

The following skeleton JCL job streams are invoked:

MQMPRLM

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//*      THIS PROCEDURE IS USED TO LIST INFORMATION ABOUT THE LOG
//*
//MQM&SYSID EXEC PGM=CSQJU004
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSUT1    DD DISP=SHR,DSN=&SYSID..BSDS01
```

MQMCOPY

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//* THIS PROCEDURE IS USED TO COPY A QUEUE OR PAGESET TO A DATASET
//*
)SEL &S = P
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
```

```
//CSQUOUT   DD DSN=&USERID..CSQUTIL.OUTPUT,
//             DISP=(NEW,CATLG),
//             SPACE=(CYL,(5,1),RLSE),UNIT=SYSDA,
//             DCB=(RECFM=VBS,BLKSIZE=23200)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
* COPY PAGESET TO 'CSQUOUT'
  COPY PSID(&PSN)
)ENDSEL
)SEL &S = Q
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//CSQUOUT   DD DSN=&USERID..CSQUTIL.OUTPUT,
//             DISP=(NEW,CATLG),
//             SPACE=(CYL,(5,1),RLSE),UNIT=SYSDA,
//             DCB=(RECFM=VBS,BLKSIZE=23200)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
* COPY QUEUE TO 'CSQUOUT'
  COPY QUEUE(&QUE)
)ENDSEL
```

MQMEMPTY

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//* THIS PROCEDURE IS USED TO EMPTY A PAGESET OR A QUEUE
//*
)SEL &S = P
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
* EMPTY PAGE SET
  EMPTY PSID(&PS)
)ENDSEL
)SEL &S = Q
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
* EMPTY QUEUE
  EMPTY QUEUE(&QUE)
)ENDSEL
```

MQMLOAD

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//*  THIS PROCEDURE IS USED TO RESTORE DATASET TO A QUEUE
//*
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//OUTPUTA   DD DSN=&USERID..CSQUTIL.OUTPUT,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
  LOAD  QUEUE(&QUEUE) DD(OUTPUTA)
```

MQPRINTQ

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//*      THIS PROCEDURE IS USED TO PRINT AN MQSERIES QUEUE
//*              (ONLY THE FIRST 80 CHARACTERS)
//PRINTQ   EXEC PGM=CSQ4BVA1,PARM='&SYSID,&QUE'
//STEPLIB  DD   DSN=SYS1.COB2LIB,DISP=SHR
//         DD   DSN=MQM.SCSQANLE,DISP=SHR
//         DD   DSN=MQM.SCSQLOAD,DISP=SHR
//         DD   DSN=your.MQM.LOAD,DISP=SHR
//SYSDBOUT DD   SYSOUT=*
//SYSABOUT DD   SYSOUT=*
//SYSPRINT DD   SYSOUT=*
//SYSOUT   DD   SYSOUT=*
```

MQDEFINE

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//* THIS PROCEDURE IS USED TO GENERATE A BACKUP SET
//* OF YOUR CONFIGURATION
//*
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//OUTPUT1   DD DISP=SHR,DSN=&SYSID..SCSQPROC(DEFS)
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSIN    DD *
```

```
COMMAND DDNAME(CMDINP) MAKEDEF(OUTPUT1)
/*
//CMDINP   DD *
DISPLAY STGCLASS(*)
DISPLAY QUEUE(*) ALL
DISPLAY NAMELIST(*) ALL
DISPLAY PROCESS(*) ALL
DISPLAY CHANNEL(*) ALL
```

MQMCOMM

```
//&USERID.S  JOB   (999,PJJ),'&USERID',
//         CLASS=A,MSGLEVEL=(1,1),
//         MSGCLASS=X,NOTIFY=&USERID
//*
//*  THIS PROCEDURE IS USED TO RUN COMMANDS IN BATCH
//*
//MQM&SYSID EXEC PGM=CSQUTIL,PARM=('&SYSID')
//STEPLIB   DD DISP=SHR,DSN=MQM.SCSQANLE
//          DD DISP=SHR,DSN=MQM.SCSQAUTH
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//INPUT     DD *
  &COMM1
  &COMM2
  &COMM3
  &COMM4
//SYSIN     DD *
  COMMAND DDNAME(INPUT)
```

MQSERIES UTILITY MESSAGE DEFINITIONS

The following message is invoked:

MQMSG00

```
MQMSG001 'GELDIG IS ..........' .ALARM=YES
'No valid value   !!!!'
MQMSG002 'GELDIG IS ..........' .ALARM=YES
'No valid value   !!!!'
```

LOCAL CUSTOMIZATION

- The *MQPRINTQ* option requires program *CSQ4BVA1* (a sample program) to be in library *your.MQM.LOAD*.

- The source for *CSQ4BVA1* is available in the IBM-supplied library *MQMvrm.SCSQCOBS*.
- The *LOAD* option defaults to the dataset generated by the *COPY* option.

Paul Jansen
System Programmer
Interpay (The Netherlands)

Guidelines for MQSeries for OS/390 users

This month's instalment concludes this article on installing and configuring MQSeries on OS/390 (the first part appeared in last month's issue).

CHANGES MADE WHEN CREATING A NEW QUEUE MANAGER

Update the following *parm* members in the *MQM.SCSQPROC* datasets and run jobs indicated.

- *CSQ4CHNL*

  Start channel initiator command and queue definitions.

- *CSQ4INP1*, *CSQ4INP2*, *CSQ4DISX*, and *CSQ4DISQ*

  Where necessary, comment out queue definitions and add any required keywords.

- *CSQ4DISP*

  This contains display commands.

- *CSQ4BSDS*

  This job defines MQ subsystem bootstrap and log datasets.

- *CSQ4PAGE*

  This job defines and formats the MQ subsystem's page set datasets.

- *CSQ4SIDE*

  This job defines 'SideInfo' for APPC support.

- *CSQ4INPX*

  This starts the *LISTENER*, *TRPTYPE(LU62)*, and *TRPTYPE(TCP)*.

- *CSQ4INPX*

  This defines the correct LU name for the listener (the name must be unique to each MQ subsystem). TCP/IP port 1414 is reserved by MQ and is the same for all subsystems.

- *CSQ4APPL*

  This handles application-related queue definitions.

- *CSQ4MQxn*

  This amends queue names for a specific subsystem id.

- *CSQ4ZPRM*

  This job creates the queue manager options module (see *CSQZMQxx*), defines the archive datasets, and re-links module *CSQZPARM*.

- *CSQ4XPRM*

  This job creates the channel initiator options module (see *CSQXMQxx*), defines the correct LU and TCP/IP task names, and re-links module *CSQXPARM*.

- *CSQ4DEFV*

  This job creates the batch adapter module, defines the correct subsystem name in member *CSQ4DEFV*, and re-links module *CSQBDEFV*.

Options

Review and, if necessary, set up the required MQ options in the *MQ**.*SCSQPROC* datasets' *CSQ4ZPRM* and *CSQ4XPRM* members. Use SMP/E's 'usermod' to track the options' macros. An example naming convention that we adopted is:

Subsystem	QMGR options Lmod	ChannelInit options Lmod
MQxn	CSQZMQxn	CSQXMQxn
MQPn	CSQZMQPn	CSQXMQPn

Below are examples of new QMGR option members set up for different subsystems.

- *MQC0(CSQZMQC0)* and *MQC1(CSQZMQC1)*

  These are unique names used for archive log file.

- *MQP1(CSQZMQPH)*

  Used on SYSH with the same log file when SMF data gathering is active.

- *MQP1(CSQZMQPA)*

  Used on SYSA with the same log file when SMF data gathering is not active.

Below are examples of new *CHINIT* option members set up for different subsystems.

- *MQC0(CSQXMQC0)* and *MQC1(CSQXMQC1)*

  These are unique LU ACBNAMEs and TCP/IP started task names.

- *MQP1(CSQXMQPH)*

  Used on SYSH with the same ACBNAME, this is the unique TCP/IP started task name.

- *MQP1(CSQXMQPA)*

  Used on SYSA with the same ACBNAME, this is the unique TCP/IP started task name.

COMMANDS TO ACTIVATE SMF ACTIVITY DYNAMICALLY

```
/ /MQPn DISPLAY TRACE(*)
/ /MQPn START TRACE(ACCTG) DEST(SMF) CLASS(*)
/ /MQPn START TRACE(STAT) DEST(SMF) CLASS(*)
```

It's important to note that you should not start global data gathering unless you intend to implement it for only a very short period. The reason for this restriction is that global data gathering increases CPU activity by about 50%. SMF activity can also be changed dynamically. Refer to the ALTER command.

AUTOMATION

After successful testing, we chose to use automation software to control the starting up and closing down of all MQ and accompanying APPC/ASCH tasks. The queue manager is started using the command shown in the first part of this article, which, in turn, issues the channel initiator task start command from the *MQM.SCSQPROC(CSQ4CHNL)* member.

OPERATING INSTRUCTIONS

Document all *MQ*** and related application started tasks instructions and commands.

MANUALS

Download the on-line manuals.

SHARING MQ SUBSYSTEM IN A PARALLEL SYSPLEX

MQ set-up in a Sysplex environment is designed to facilitate command routing across subsystems in the Sysplex.

All CPFs (command prefix) are registered with MQSeries to enable commands to be entered from any console in the Sysplex and routed to the appropriate system for execution. Command responses are then returned to the originating console.

SCOPE, defined as part of the subsystem name in *SYS1.PARMLIB(IEFSSNxx)*, is used to determine the type of CPF

registration performed by the MQSeries subsystem. A Sysplex started *SCOPE [S]* registers the CPF with MVS at the time the MQSeries subsystem is started and remains active until it terminates.

It is easy to issue *START* commands for multiple subsystems from one LPAR within a Sysplex via the *ROUTE* command – enter the following commands from any LPAR in the same Sysplex to initiate two MQ subsystems on another LPAR

```
/SYSC /MQC0 START QMGR and /SYSC /MQC1 START QMGR
```

The subsystems initiate on *SYSH* and *SYSA* respectively.

Note that this option must be used with caution when a subsystem with the same name is shared across two LPARs in a Sysplex. In other words, the commands:

```
/SYSH /MQP1 START QMGR
```

followed by:

```
/SYSA /MQP1 START QMGR
```

will initiate the same subsystem on both *SYSH* and *SYSA* if entered from another LPAR in the same Sysplex. The later eventually terminates producing voluminous messages stating that the BSDS datasets are already in use by another task.

SHARED ENVIRONMENT SETUP

- The subsystem is called *MQP1*.
- The command prefix used is: */MQP1*.
- *MQP1MSTR* and *MQP1CHIN* members are created in a shared *PROCLIB* on shared volume *SYS000*.
- Bootstrap, log, and page files are allocated using a unique Hlq on a shared volume.
- *QMGR MQP1* is started using the *CSQZMQPH* option member on *SYSH* and QMGR *MQP1* is started using the *CSQZMQPA* option member on *SYSA*.
- The channel initiator and command server are started separately

on *SYSH/A* using the additional *CSQ4CHNA/H* member in *SCSQPROC* and different *CSQXMQPA/H* option members.

- Member *CSQ4CHNL* is concatenated on *SYSH/A* so it shares channel queue (*MQP1*) definitions.

Switching shared systems MQP1 from SYSH to SYSA

- Ensure that the *STOP* command is entered on *SYSH* and verify that the subsystem has terminated successfully.
- Issue the *START* command on SYSA and verify that the subsystem has initiated successfully.

Note that, if *MQP1* is not stopped on SYSH and a subsequent start command is issued from SYSA, abnormal termination error messages are sent to *SYSH SYSLOG*. (See the *CPF* and *SCOPE* sections above.)

- If a VTAM node is set up to support links to remote MQ systems, and the subsystem is required to be switched across shared LPARs, ensure that the links are de-activated before stopping the QMGR task on the current LPAR. Issue VTAM command:

  ```
  V NET,ACT,ID=linkname,INACT,I
  ```

  Then activate the links on the other LPAR before restarting the same QMGR task. Issue VTAM command:

  ```
  V NET,ACT,ID=linkname,SCOPE=ALL
  ```

  The relative links of the channels defined on the MQ subsystem also need to be re-activated. This can be done either by restarting the channel initiator (channels defined in *MQxnCHNL*) or manually using MQ panels.

DOS AND DON'TS

It is important that the *MQM.SCSQPROC(member)* concatenation order is not changed in the procedure supplied for the started task. Failure to observe this rule will result in abnormal termination.

Consider the example below.

```
//CSQINP1   DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4INP1),DISP=SHR
```

```
//CSQINP2   DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4STGC),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4INP2),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4DISQ),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4DISX),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4CHN&LPAR),DISP=SHR CHINI
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4CHNL),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4MQP1),DISP=SHR    MQP1DEFS
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4TLOG),DISP=SHR
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4INPX),DISP=SHR    START LU
//          DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4DISP),DISP=SHR
//*         DD DSN=SYSXA.MQP1.SCSQPROC(CSQ4IVP),DISP=SHR
//CSQOUT1   DD SYSOUT=*
//CSQOUT2   DD SYSOUT=*
//*
```

When checking through abnormal failures, ensure that the channel initiator is started before the listeners.

If a VTAM node is set up to support links to remote MQ systems, and its connections are disturbed when the links are active, the relative links to channels defined in all affected MQ systems need to be reactivated. To do this, first issue the following VTAM command:

```
V NET,ACT,ID=linkname
```

After a few minutes you should see a message in the *MQxnCHIN* log similar to:

```
CSQX500I CHANNEL.linkname.TO.MQxn.LU started
```

If this message isn't logged, reactivate the links manually using MQ's panels. For ease of recognition, set the second qualifier in the channel definition to be the same as the link name used in the VTAM definition.

If you need to stop the MVS MQ subsystem when remote activity is taking place (for instance, during *SEND*/*RECEIVE* operations to other MQ subsystems or when the MVS MQ panels are active), then use *QUIESCE MODE*. This requires a bit of patience, as this mode takes a while to implement – don't be tempted to use the *FORCE* command or cancel the task.

Note that, if after about ten minutes *MQxnCHIN* has stopped but *MQxnMSTR* is still running and there is no remote activity, then there are MQ panels still active. To remedy this, notify all users to exit panels.

ABEND 5C6 and *6C6* are critical errors that you may encounter. If you do come across them, then you'll probably have to recover using either *LOGCOPY* or *ARCHIVE* files. One cause for this abend is *UNSYNCHRONIZED LOG* and *PAGE* datasets.

If *LOG* files are redefined, make sure that the *PAGE* datasets are also redefined. If not, the task will abend with *CODE 5C6* and *6C6* (refer to *CSQ4BSDS* and *CSQ4PAGE* in the *MQM.SCSQPROC* dataset). In a test environment, the quickest solution is to redefine these files. In a production environment, other recovery tasks may need to be performed.

Saida Davies
IBM (UK)

Client Attachment Feature

I thought I'd clarify the status of the Client Attachment Feature for OS/390 that I mentioned in my article 'Guidelines for MQSeries for OS/390 users' (published in the March 2000 issue of *MQ Update*, page 40).

This feature is both chargeable and optional. It's required only if you are going to attach clients to your MQSeries for MVS/ESA subsystem. Once you've installed it, it's ready for use and requires no configuration parameters before you can attach clients to MQSeries for MVS/ESA. Note that client administration is available even if you don't install this feature.

For further information on pricing, refer to the following Web sites:

- Client Attachment Feature

```
http://www.ehone.ibm.com/public/applications/price/cgibin/
pricesw.cgi?Country=GB&LceKey=962E0000&MachType=5695
&MachModel=137&Feature=
```

- Licence for fifty client attachments to MQSeries for OS/390

  ```
  http://www.ehone.ibm.com/public/applications/price/cgibin/
  pricesw.cgi?Country=GB&LceKey=CBF90000&MachType=5655
  &MachModel=A95&Feature=
  ```

Saida Davies
IBM (UK)

An event queue monitor for OS/390

INTRODUCTION

Applications that use MQSeries typically have components running on more than one platform. In such an environment, it's no trivial task to guarantee the availability of each queue manager and the channels that connect them. This demands close monitoring of key resources, such as channel status and queue usage.

MQSeries provides a mechanism to allow the monitoring of its resources. When certain resources change their status, event messages are put onto event queues, and these queues can be triggered so that their messages are read by appropriate programs.

MQSEVT is a program written in ANSI C that runs as a CICS transaction triggered by an event queue. It analyses each event message and takes simple actions, such as writing a message to a system console or resetting and starting a channel. It takes actions on a limited set of event messages, but may easily be tailored to handle other event messages as well.

This article describes the key features of *MQSEVT* and the site-related assumptions that guided its development. This should assist an MQSeries administrator in deploying it adequately in a new scenario.

A full listing of the program may found at the end of the article.

DEPLOYMENT SCENARIO

MQSEVT was designed to run on a star-shaped TCP/IP MQSeries network in which an OS/390 queue manager running MQSeries for MVS Version 1.2 acts as the central hub to approximately 30 remote queue managers that are connected to it and run MQSeries for HP-UX Version 2.2.1 and MQSeries for Windows NT Version 5.0. Its useful deployment is based on the following assumptions:

- Console operators are available at the OS/390 site 7x24 hours.
- Remote nodes send their event messages to the OS/390 hub. Remote channel event messages are suppressed.
- Queues names start with the prefix of the application that consumes its messages.
- Channel names follow the standard:

  ```
  <local-qmgr>.<remote-qmgr>
  ```

- Every queue manager name is defined in DNS as an alias for the node on which it runs.
- *KeepAlive* is appropriately configured at every node to avoid orphaned receiver channels.
- Sender channels are configured to allow for infinite retries.

EVENT HANDLING POLICY

MQSEVT's main goal is to monitor queue servicing and channel status. Therefore, only performance and channel-related events are handled. The specific events handled were chosen after careful observation of how applications behave and the errors that have a significant impact on their availability.

When a channel goes down, *MQSEVT* sends a message with the channel name and TCP/IP return code to the system console. After the underlying link problem is fixed, the channel retry mechanism automatically restarts the channel.

Out-of-sequence messages are handled to allow for the way that Version 2 queue managers control channel status information. This

situation is detected by looking out for different source IP addresses from the same channel. *MQSEVT* resets and starts the channel when this occurs.

When message production and consumption are simultaneous, the queue service rate will reveal problems more quickly than the queue depth. Thus, if a program stops getting messages off a queue or starts to do so too slowly, a queue service high event may reveal an upcoming problem even if queue depth is still insignificant. *MQSEVT* then sends a message to the system console with the queue name and the queue manager name. Queue depth events are not handled, though this may easily be added.

MODIFYING EVENT HANDLING POLICY

Each event is handled by a separate function and its name should mirror the associated reason code name. All handlers must receive the same argument list and return *void*. For example, the handler definition for *MQRC_CHANNEL_STOPPED* should take the form:

```
void  ChannelStopped (
          MQHCONN hConn,
          MQMD dEveMsg,
          PMQBYTE EveMsg,
          long Action
)

{
...
}
```

A table identifies each handled event along with its handler:

```
static EveMsgHandler TabEveMsgHandler [] = {
    { MQRC_CHANNEL_STOPPED,         ChannelStopped       },
    { MQRC_Q_SERVICE_INTERVAL_HIGH, QServiceIntervalHigh }
};
```

where *EveMsgHandler* is a user-defined type:

```
typedef struct  {
            MQLONG  Reason;
            void    (* Handler) (
                MQHCONN hConn,
                MQMD dEveMsg,
```

```
                PMQBYTE EveMsg,
                long Action
);
        }       EveMsgHandler;
```

To add a new handler:

- Add a handler prototype to section:

```
/*-------------------------------------------------------------*/
/*   prototypes                                                */
/*-------------------------------------------------------------*/
```

- Add the event reason code and the handler function name to the events table:

```
EveMsgHandler*   FindEveMsgHandler (MQLONG Reason)
{
    static EveMsgHandler TabEveMsgHandler [] = {
        { MQRC_CHANNEL_STOPPED,         ChannelStopped       },
        { MQRC_Q_SERVICE_INTERVAL_HIGH, QServiceIntervalHigh }
    };

    ...

}
```

- Code the new handler function, following the example of the prototype below:

```
void  (* Handler) (
          MQHCONN hConn,
          MQMD dEveMsg,
          PMQBYTE EveMsg,
          long Action
);
```

To code a new handler function:

- Code a table with all fields that need to be extracted from the event message. The table should contain the *MQLONG* constants that identify each field, for instance:

```
MQLONG  FieldIds [] = {
            MQIACF_REASON_QUALIFIER,
            MQCACH_CHANNEL_NAME,
            MQIACF_ERROR_IDENTIFIER,
            MQIACF_AUX_ERROR_DATA_INT_1,
        };
```

- Define appropriate variables to hold each field, for instance:

```
MQLONG  ReasonQualifier;
MQCHAR  ChannelName [ MQ_CHANNEL_NAME_LENGTH + 1 ];
MQLONG  ErrorIdentifier;
MQLONG  AuxErrorDataInt1;
```

- Extract the fields by calling *GetField*. The fields need to be initialized, as *GetField* doesn't fill variables for fields that are not found in the event message data. For example:

```
ReasonQualifier = 0;
ChannelName [0] = '\0';
ErrorIdentifier = 0;
AuxErrorDataInt1 = 0;

GetField (
    EveMsg,
    FieldIds,
    sizeof (FieldIds) / sizeof (MQLONG),
    &ReasonQualifier,
    ChannelName,
    &ErrorIdentifier,
    &AuxErrorDataInt1
);
```

- Code the appropriate action based on the fields extracted.

SETTING UP THE ENVIRONMENT

- Delete all remote event queues. Redefine QMGR and performance event queues as remote, and point them to the equivalent OS/390 queue manager queues.
- Use 'trigger first' on all OS/390 event queues, pointing them to a single process and *initq*. The process should be a CICS transaction associated with *MQSEVT* in the CICS region chosen.
- Code the process attribute *ENVRDATA* to establish how messages behave on the system console. Its possible values are:
  - EVENTUAL
  - IMMEDIATE
  - CRITICAL.

These correspond to their related CICS *ACTION_** constants. The *ACTION_DEFAULT* macro is used if the attribute is not coded.

- Enable performance events on all monitored queue managers, and code adequate queue service interval values for each monitored queue.

IMPLEMENTATION DETAILS

- The function *GetField* accepts a variable number of arguments and hides the complexity of dealing with PCF structures and undefined parameter order in event messages.
- When extracting queue names, it is advisable to extract both *MQCA_Q_BASE_NAME* and *MQCA_Q_NAME*, and use the non-empty one. Different platforms and versions are not the same in this respect.
- The function *RunCommand* does not check whether the command was performed successfully. It can, however, be tailored to do so, as it processes all replies from the command server.
- The function *OperatorPrintf* has the same syntax as *printf*, albeit with an added parameter to determine how the message will behave on the system console.
- All internal error messages and commands issued are sent to the system console.
- The macro *MQS_PREFIX* is prefixed to every message sent to the system console.
- The Macro *TIME_OUT* determines the wait interval in seconds.

The program may be adapted to run on non-MVS platforms:

- Remove all CICS command-level verbs: *exec cics address*, *retrieve*, *write operator*, and *return.*
- Adapt function *OperatorPrintf* to send messages to the local console.
- Modify function *RunCommand* to send PCF commands.

MQSEVT.C

```
/*
  Processes event messages.

  All unhandled event messages are discarded.

  Messages issued by the program are recorded on the system console
  with a prefix from macro MQS_PREFIX. The 'EnvironmentData' attribute
  from process MQS.ADMIN.EVENT controls how the messages
  are recorded:

      EVENTUAL    Rolls up the screen (ACTION_EVENTUAL).
      IMMEDIATE   Highlight, then wait for operator deletion
                  (ACTION_IMMEDIATE)
      CRITICAL    Highlight, then wait for operator deletion
                  (ACTION_CRITICAL).

  Error messages and actions taken are recorded with attribute
  ACTION_EVENTUAL. Triggered-first under CICS.

  Events handled (based on MQMD.Reason):

      MQRC_CHANNEL_STOPPED
      MQRC_Q_SERVICE_INTERVAL_HIGH

  Situations handled when MQRC_CHANNEL_STOPPED:

  - Disconnect interval expired:
    ReasonQualifier == MQRQ_CHANNEL_STOPPED_OK.

    Ignores.

  - Connection reset by peer:

    1 Remote machine went down. Sender channels still running.
      KeepAlive probes are sent, no reply received. TCP stack
      returns 36, and sender channel ends abnormally.

    2 Remote machine went down and came back. Sender channels still
      running. An application tries to send data through the channel.
      Remote TCP stack sends reset, as the connection no longer
      exists. Local TCP stack returns 36, and sender channel ends
      abnormally.
      TCP/IP return code == 36.

    Ignores.

  - Channel is already active:
    Sender channel running and communication link goes down. Sender
    channel goes to retry. Link comes back, sender channel tries to
```

```
    start new connection, but former receiver channel is still running.
    Receiver channel eventually dropped as a result of KeepAlive
    mechanism.
    ReasonQualifier == MQRQ_CHANNEL_STOPPED_ERROR &&
    ErrorIdentifier == xxxxx514 (msg CSQX514E).

    Ignores.

  - Message sequence error:
    ReasonQualifier == MQRQ_CHANNEL_STOPPED_ERROR &&
    ErrorIdentifier == xxxxx506 (msg CSQX506E).

    Channel reset
    Channel start.

  - Other errors:

    Records the message below at the system console:

      Channel <channel-name> with problems (tcp=<return-code>)

  - Observations:

    1 'Network unreachable' and 'Connection or remote listener
      unavailable' are not given special treatment any more.
    2 Consequences of the channel being stopped manually are not
      handled - with ptf UQ18727, an event is raised with reason
      code MQRC_CHANNEL_STOPPED_BY_USER.

  Situations handled when MQRC_Q_SERVICE_INTERVAL_HIGH:

  - Records msg at the system console:

      Queue <qmgr>/<queue> is not being read.

*/

/*--------------------------------------------------------------------*/
/*   Includes.                                                        */
/*--------------------------------------------------------------------*/
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <stdarg.h>
#include    <cmqc.h>
#include    <cmqcfc.h>

/*--------------------------------------------------------------------*/
/*   Defines.                                                         */
/*--------------------------------------------------------------------*/
```

```
#define REPLY_LEN               4096
#define BUF_LEN                 1024
#define MSG_LEN                 160
#define CMD_LEN                 80
#define MQS_PREFIX              "MQSEVENT"
#define TIME_OUT                5

#define ACTION_EVENTUAL         3
#define ACTION_IMMEDIATE        2
#define ACTION_CRITICAL         11
#define ACTION_DEFAULT          ACTION_EVENTUAL

/*--------------------------------------------------------------------*/
/*   User-defined data types.                                         */
/*--------------------------------------------------------------------*/
typedef struct  {
            MQLONG  Reason;
            void    (* Handler) (
                MQHCONN hConn,
                MQMD dEveMsg,
                PMQBYTE EveMsg,
                long Action
            );
        }       EveMsgHandler;

/*--------------------------------------------------------------------*/
/*   Prototypes.                                                      */
/*--------------------------------------------------------------------*/
void    ChannelStopped (
            MQHCONN hConn,
            MQMD dEveMsg,
            PMQBYTE EveMsg,
            long Action
        );
void    QServiceIntervalHigh (
            MQHCONN hConn,
            MQMD dEveMsg,
            PMQBYTE EveMsg,
            long Action
        );
void    GetField (PMQBYTE EveMsg, MQLONG* FieldIds, int TotFields,
                  ...);
void    RunCommand (MQHCONN hConn, char* Command);
void    OperatorPrintf (long Action, char* Fmt, ...);
char*   TrimRight (char* Str, int Len);
char*   StrLower (char* Str);

EveMsgHandler*    FindEveMsgHandler (MQLONG Reason);

/*--------------------------------------------------------------------*/
```

```
/*   Main.                                                          */
/*-----------------------------------------------------------------*/
void    main (int argc, char** argv)
{
    MQHCONN     hConn = MQHC_DEF_HCONN;
    MQHOBJ      hEveQ;

    MQOD        dEveQ       = { MQOD_DEFAULT };
    MQMD        dEveMsg     = { MQMD_DEFAULT };

    MQLONG      oOpen;
    MQLONG      oClose;
    MQGMO       oGet        = { MQGMO_DEFAULT };

    MQLONG      CompCode;
    MQLONG      Reason;

    MQLONG      MsgLen;
    MQCHAR      Buf [ BUF_LEN ];

    int         Action;
    char*       pAction;

    EveMsgHandler*   pEveMsgHandler;

    DFHEIBLK*   pEib;
    MQTM        TrigMsg;
    short int   TrigMsgLen;
    long int    Resp;

    /*-------------------------------------------------------------*/
    /*    Address CICS EIB.                                        */
    /*-------------------------------------------------------------*/
    exec cics   address
                eib (pEib)
    ;

    /*-------------------------------------------------------------*/
    /*    Receives trigger msg handed by CKTI.                     */
    /*-------------------------------------------------------------*/
    TrigMsgLen = sizeof (TrigMsg);

    exec cics   retrieve
                into (&TrigMsg)
                length (TrigMsgLen)
                resp (Resp)
    ;

    if  (Resp != DFHRESP(NORMAL))
        exec cics return;
```

```
/*-------------------------------------------------------------*/
/*    Message handling for system console.                     */
/*-------------------------------------------------------------*/
pAction = TrimRight (TrigMsg.EnvData, sizeof (TrigMsg.EnvData));
StrLower (pAction);

if  (strcmp (pAction, "eventual") == 0)
    Action = ACTION_EVENTUAL;
else if  (strcmp (pAction, "immediate") == 0)
    Action = ACTION_IMMEDIATE;
else if  (strcmp (pAction, "critical") == 0)
    Action = ACTION_CRITICAL;
else
    Action = ACTION_DEFAULT;

/*-------------------------------------------------------------*/
/*    Opens event queue                                        */
/*-------------------------------------------------------------*/
strncpy (
    dEveQ.ObjectName,
    TrigMsg.QName,
    sizeof (dEveQ.ObjectName)
);

oOpen   = MQOO_INPUT_AS_Q_DEF
        + MQOO_FAIL_IF_QUIESCING;

MQOPEN (
    hConn,
    &dEveQ,
    oOpen,
    &hEveQ,
    &CompCode,
    &Reason
);

if  (CompCode != MQCC_OK)   {
    OperatorPrintf (
        ACTION_EVENTUAL,
        "MQOPEN %s: %ld",
        TrimRight (dEveQ.ObjectName, sizeof (dEveQ.ObjectName)),
        Reason
    );
    exec cics return;
}

/*-------------------------------------------------------------*/
/*    Analyses all event messages.                             */
/*-------------------------------------------------------------*/
for  ( ; ; )   {
```

```
/*-------------------------------------------------------------*/
/*    Get next event message.                                  */
/*-------------------------------------------------------------*/
memmove (dEveMsg.MsgId, MQMI_NONE, sizeof (dEveMsg.MsgId));
memmove (dEveMsg.CorrelId, MQCI_NONE, sizeof
                                        (dEveMsg.CorrelId));

oGet.Options    = MQGMO_WAIT
                + MQGMO_NO_SYNCPOINT
                + MQGMO_CONVERT
                + MQGMO_FAIL_IF_QUIESCING;

oGet.WaitInterval = TIME_OUT * 1000;

MQGET (
    hConn,
    hEveQ,
    &dEveMsg,
    &oGet,
    sizeof (Buf),
    Buf,
    &MsgLen,
    &CompCode,
    &Reason
);

/*-------------------------------------------------------------*/
/*    Queue is empty: there's nothing else to do...            */
/*-------------------------------------------------------------*/
if  (Reason == MQRC_NO_MSG_AVAILABLE)   break;

if  (CompCode != MQCC_OK)   {
    OperatorPrintf (
        ACTION_EVENTUAL,
        "MQGET %s: %ld",
        TrimRight (dEveQ.ObjectName, sizeof
                    (dEveQ.ObjectName)),
        Reason
    );
    exec cics return;
}

/*-------------------------------------------------------------*/
/*    Ignores non-event messages, in case the transaction      */
/*    is run against a non-event queue.                        */
/*-------------------------------------------------------------*/
if  (strncmp (dEveMsg.Format, MQFMT_EVENT, MQ_FORMAT_LENGTH))
    continue;

/*-------------------------------------------------------------*/
```

```
        /*    Finds a handler for this event message using         */
        /*    MQCFH.Reason.                                        */
        /*---------------------------------------------------------*/
        pEveMsgHandler = FindEveMsgHandler (((PMQCFH) Buf)->Reason);

        /*---------------------------------------------------------*/
        /*    Ignores messages that don't have a handler.          */
        /*---------------------------------------------------------*/
        if  (pEveMsgHandler == (EveMsgHandler*) NULL)
            continue;

        /*---------------------------------------------------------*/
        /*    Handles message.                                     */
        /*---------------------------------------------------------*/
        (* pEveMsgHandler->Handler) (hConn, dEveMsg, (PMQBYTE) Buf,
                                     Action);
    }

    /*-------------------------------------------------------------*/
    /*    Closes event queue.                                      */
    /*-------------------------------------------------------------*/
    oClose  = MQCO_NONE;

    MQCLOSE (
        hConn,
        &hEveQ,
        oClose,
        &CompCode,
        &Reason
    );

    if  (CompCode != MQCC_OK)   {
        OperatorPrintf (
            ACTION_EVENTUAL,
            "MQCLOSE %s: %ld",
            TrimRight (dEveQ.ObjectName, sizeof (dEveQ.ObjectName)),
            Reason
        );
        exec cics return;
    }
    exec cics return;
}
```

This article concludes in next month's issue of *MQ Update* with the remainder of the code for MQSEVT.C.

Paulo Marcel Coelho Aragão (Brazil)

Unisys is to sell MQSeries on its ClearPath HMP IX Enterprise Servers. Unisys is to provide MQSeries product functionality as an X/Open-compliant Resource Manager with the Unisys Open/OLTP 2200 Transaction Manager. This, says Unisys, allows message queuing functionality to be tied to the recovery, availability, transactional persistency, and high performance attributes of the ClearPath IX and related 2200 Series systems.

General product availability is targeted for second quarter of this year.

For further information contact:

Unisys Corp, PO Box 500, Blue Bell, PA 19424-0001, USA
Tel: +1 215 986 4011
Web: http://www.unisys.com

Unisys Ltd, Bakers Court, Bakers Road, Uxbridge UB8 1RG, UK
Tel: +44 1895 237137
Fax: +44 1895 862093

* * *

ETI Software has announced ETI*Extract Version 4.1. The company says that the product's enhancements are aimed at improving integration between e-commerce, back-office, and decision support applications. Although no mention was made to improvements to the product's support for MQSeries (which, admitedly, was only introduced in November last year), other enhancements include an upgraded MetaStore, which gets improved security and distributed support, an audit trail for metadata, and improvements to the product's ergonomics to reduce the number of keystrokes required to achieve administrative tasks.

The product is out now; details on pricing are available on request from the vendor.

For further information contact:

Evolutionary Technologies International, 816 Congress Avenue, Suite 1300, Frost Bank Plaza, Austin, TX 78701, USA
Tel: +1 512 327 6994
Fax: +1 512 327 6117
Web: http://www.eti.com

Evolutionary Technologies Limited, Denmark Court, 18 Market Place, Wokingham, Berkshire RG40 1AL, UK
Tel: +44 118 977 1221
Fax: +44 118 977 9800

* * *

Meanwhile, IBM announced that it's reselling ETI's ETI*Accelerator for MQSeries, which combines data access and transformation with the message delivery attributes of MQSeries. It's aimed at users building enterprise application integration infrastructures and adding new levels of data transformation and metadata management to data messaging.

Out now on AIX, Solaris, HP-UX, and NT servers, it costs US$30,000.

IBM also has announced that it's ported MQSeries and Tivoli Storage Manager to NUMA-Q.