29

MQ

November 2001

In this issue

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.

Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

Managing 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.

HIS and its MSMQ MQSeries Bridge part 3: administration and performance

Concluding the series of articles on Host Integration Server (HIS) 2000 and its MSMQ MQSeries Bridge, we look this month at administration and performance.

CONTROLLING THE BRIDGE

- To control the Bridge, you right-click an object in the Bridge Manager and select options. For example, you can stop, start, pause, and resume operations, and the command selected affects all objects in the object tree.
- Periodically, you can refresh the cache memory to close queues needed by other applications. To do this, you right-click a Bridge Service or Message Pipe and choose the Refresh Cache selection.
- The Bridge configuration is stored in the registry and you can use the Registry Editor to back-up this configuration:
  - in a non-clustered set-up use: *HKEY_LOCAL_MACHINE\Software\Microsoft\MQBridge\Server*
  - in a clustered set-up use: *HKEY_LOCAL_MACHINE\Cluster\Software\Microsoft\MQBridge\Server*.
- MQSeries doesn't support queue manager names with hyphens (–), so the Bridge provides a feature to provide name replacement, enabled by going to the Bridge Properties, Advanced tab. In MQSeries you can use a period (.) and the Bridge will convert the period back to a hyphen.
- Win2K does not allow computer names to use a period so they can't be used in naming a remote MQSeries Queue Manager. The Bridge provides a feature, which is enabled in the Registry by default, allowing name replacement, so that in Win2K you can use a hyphen and the Bridge will convert the hyphen to a period when the message arrives. You can use the Registry Editor to change this default. The required key is: *HKEY_LOCAL_MACHINE\Software\Microsoft\MQBridge\Server*.

- There are two queues for messages that are undeliverable:
  - MQBridge Dead Letter Queue: messages sent through the non-transactional message pipe
  - MQBridge Xact Dead Letter Queue: messages sent through the transactional message pipe.

  You can locate these queues in NT 4 using the MSMQ Explorer under enterprise_name, enterprise_servers, bridge_computer, and in Win2K using Active Directory Users and Computers, Domain Controllers or Computers, bridge_computer, msmq.
- The Bridge converts foreign queue and queue manager names to upper case (UPPER CASE), since the Bridge can only send messages to MQSeries queue managers and queues that have upper case names.
- You should wait up to 15 minutes after enabling or disabling encryption to allow sufficient time for the Bridge to work correctly.
- To convert a Bridge into a cluster resource: in the *Program Files\Host Integration Server\system* directory, type **BCLUSTER** at the command line, click Add Bridge Resource; in the Bridge Manager right-click the Microsoft MSMQ-MQSeries Bridge Service, select Bring Online. In W2K, to run the cluster only on the local node, run **BCLUSTER**, click Remove Resource. In W2K you need to stop the Bridge Service on all local nodes before you bring the Bridge cluster resource online.
- It's often necessary to modify the access for users created during the install. You can do this by using Active Directory Sites and Services; right-click the foreign site, click the Security tab and select Domain Admins, select Full Control.
- To set up a foreign site in W2K you should install MSMQ server with routing, run Active Directory Sites and Services to create a new foreign site and new foreign computer, and right-click the *Services\MsmqServices* folder. To create a routing link between the foreign site and the default first site name: enter a number in the Routing Link Cost – 1 has the highest priority and 0 means no link. Select Property, select the Site Gates tab and add the Bridge

computer to be a member. Using Active Directory Users and Computers, Domain Controllers or Computers, bridge computer, msmq, select Property, Sites tab and add the computer to the foreign site created earlier. Reprocess the MSMQ service on the Domain Controllers.

- It's recommended that the MQSeries client and MQSeries server connection be tested using IBM's AMQSPUTC and AMQSGETC tools.
- You need to use the Add Schema program before you can enable Bridge encryption. You do this by logging in as the Schema Administrator. Go to the *Program Files\Host Integration Server\system* directory and type this command line: **addschma hiserver.schema**. Using the Bridge Manager, go to Enterprise, Computers, computer name. Right-click the Bridge Service, select Properties, select the Advanced tab, select the Support MSMQ to Bridge Encryption check box, select OK, click Yes, click OK; restart the Bridge Service. To disable encryption: advanced tab, clear the Support MSMQ to Bridge Encryption box, reprocess the Bridge Service.
- One final point to note is that the Bridge rejects MSMQ Messages with *MQMSG_AUTH_LEVEL_MSMQ20* authentication under Win2K.

PERFORMANCE ISSUES

The Bridge is not CPU-bound, allowing other back-end applications to run on the same machine. Generally, the more queue managers added to a Bridge, the more throughput increases without much degradation. You should set the number of threads that are utilized for each of the message pipes to reflect the number of queue managers that the Bridge will service, since there is no noticeable difference in performance if you put in more threads than MQSeries Queue Managers.

The settings that do have an impact are on each message pipe and can be configured by right-clicking, selecting Properties, Batch tab, and three configurable batch options:

- Default:
  - maximum number of: 10
  - maximum accumulated size: 1024
  - maximum accumulated time: 512.
- Try sample settings such as:
  - maximum number of: 1200
  - maximum accumulated size: 350000
  - maximum accumulated time: 256.

Optional fields such as Reply To Queue Manager and Encryption can be used to increase the amount of time required to send messages. Encryption is a costly operation when a call requires an MSMQ SQL database lookup. Limiting the number of times that the MSMQ and MQSeries protocols are needed helps to keep overheads to a minimum. It's expensive on the network to open and close queues, but if a queue is not closed after long periods of time it has consequences for the machine.

Stephen Ibaraki, ISP
Chief Architect, iGen Knowledge Solutions (Canada)

Writing exits for enableNet Data Integrator (eNDI)

A BRIEF HISTORY OF ENDI

FTF/MQ

enableNet Data Integrator is a tool for moving bulk data across an MQSeries infrastructure. It has been around in various incarnations since 1996 and was originally known as FTF/MQ, which is an abbreviation for File Transfer Facility for MQSeries. In its original form it performed the task of moving files across networks using the facilities of MQSeries V2, which was the latest level available. There

were many motives, which are still valid now, for choosing this method of file transfer over traditional file transfer products. These products are typically modelled on the free Unix FTP utility, which implements a synchronous transfer model between a client and server. They typically have the following characteristics and limitations:

- *Connection-oriented*: a network session or conversation is established between the source and the destination. The underlying transport mechanism may be reliable or unreliable. Sessions are created, used, and terminated for each and every individual file transfer request.
- *Non-network transparent*: because they have to interact directly with the network layer protocols, most file transfer utilities cannot isolate the source and the destination processes from the idiosyncrasies of the network layer.
- *Non-modular*: in most file transfer utilities the source and the destination processes interact directly with the file systems and are directly responsible for manipulating the files on both the source and the destination.
- *Limited restart and recovery*: most file transfer utilities provide limited restart and recovery logic to recover from system and network failures. In most cases the entire file transfer request has to be restarted from scratch in case of system and network failures.
- *Limited workflow and application integration*: tools such as FTP provide very limited – if any – workflow and business application integration hooks.
- *Lack of centralized monitoring and administration*: providing a consistent management and administration framework for both the file transfer utility as well as the underlying transport mechanism is limited to very few file transfer utilities.
- *Non-scalable*: eNDI is the only tool available that can move a single file to a destination across multiple network paths in parallel.

enableNet Data Integrator addresses and alleviates these limitations by the clever use of MQSeries facilities and additional features.

Even in its original form the product incorporated the ability to make calls to user exits, which allows business processes to be incorporated within the flow of data. A toolkit for writing user exits is supplied with the product and is the focus of this article. I shall expand on where exits fit into the transfer in the architecture section.

e-Adapter

In 1998, there came the introduction of connectors, which differ slightly from exits. Business process integration exits are user-customizable and can be employed in a 'plug and play' manner. They are called at strategic points during the data transfer transaction. Connectors are internal integration points that can be used to override the internal I/O in favour of custom or specialized I/O. This provides a level of data access far above and beyond the capabilities and limitations of base file I/O. For example, the enableNet Data Integrator Connectors can perform any type of data access and data manipulation within the data transfer transaction. This includes databases and/or proprietary data sources. In general, exits allow user-written processes to execute and, potentially, access the file being transferred, but connectors works at the I/O level so that the data is read or written out directly to an alternative data store, not necessarily a file.

Because the product was no longer restricted to file transfer, but could be used as a more general data integration tool, it was thought that the name FTF/MQ didn't reflect the facilities available. With the advent of e-everything in 1998, e-Adapter was thought to be a suitable name to reflect the credentials of the enhanced product.

enableNet Data Integrator

In 2000, CommerceQuest launched a B2B offering named enableNet, which allows business to business integration covering point to point communication right through to wholesale e-business marketplaces. Business rules, data transformation, and workflow capabilities can be executed at the hub or spokes by a tool known as enableNet Business Process Integrator (eBPI). I won't go into any of the details of enableNet, but at the core of this offering is the bulk data movement product. In order to maintain consistency of branding it was decided to bring e-Adapter under the umbrella of enableNet by renaming it enableNet Data Integrator. With the latest release comes the virtual

machine toolkit for writing exits and connectors using the eBPI scripting capabilities and various other enhancements, including support for XML-formatted transfer requests. The product currently has implementations on the following platforms: Windows NT, Windows 2000, HP-UX, Sun Solaris, AIX, SCO, OS/390, OS/400, VAX VMS, VAX Alpha, OS/2, Windows 98/Me (with MQWin), 4690 (with CommerceQuest's 4690 MQSeries client).

ENDI ARCHITECTURE

eNDI comprises three components: a Manager, a Sender, and a Receiver.

Exits provide for pre- and post-processing during a transaction. These exits are designed to be customizable for specialized processing or they can consist of existing business modules. The exits operate as part of the base transfer request and allow for autonomous processing steps to be included as part of the processing. The exits are called as either the first processing step or last processing step of the Manager, Sender, or Receiver.

Connectors are a unique type of exit. The Connector is called in place of reading or writing a file so that any type of data processing can take place and there is no limitation or restriction to a data format or a file.

Manager

The Manager is responsible for maintaining the state of all processing, dispatch requests, and load balancing. Multiple managers in strategic locations can cater for transaction frequency. The Manager component accepts the request for processing. If a Manager pre-processing exit is installed, control will be passed to this module. Given positive results from the exit, the Manager will continue processing. The Manager will dispatch the request to the appropriate Sender and continue processing subsequent requests. Additionally, the Manager will accept replies from the other components, thus completing the transaction. After all correspondence between components is complete for a given transaction, the last activity the Manager will perform is to call the post process exit (if one is installed) and finally reply to the calling interface.

Sender

The Sender will accept a request that was dispatched from the Manager. If a Sender pre-processing exit is installed, control will be passed to this module. Given positive results from the exit, the Sender will continue processing. After the user has been authorized to read the data, if a Connector is installed it is the responsibility of this module to hand the data to the Sender (via the Connector interface), otherwise the Sender itself will be responsible for reading the data object via its base file I/O support. The Sender will send the data as messages to the target Receiver component. Following successful submission of the data, if a post-processing exit is installed control is passed to the exit followed by a confirmation of activity to the Manager. A sender post-process exit is executed even if there was a failure sending the file.

Receiver

The Receiver is responsible for receiving the incoming messages and reconstructing the target data object. Like the Sender, multiple Receivers will cater to concurrent processing. The Receiver will accept incoming data from a Sender. If a Receiver pre-processing exit is installed control will be passed to this module. Given positive results from the exit, the Receiver will continue processing. After the user has been authorized to write the data, if a Connector is installed it is the responsibility of the Receiver to pass the data to the Connector so that the Connector can perform its desired actions, otherwise the Receiver will write the data object via its base file I/O support. Following the processing of the data, if a post-processing exit is installed control is passed to the exit followed by a confirmation of activity to the Manager. A receiver post-process exit is executed even if there was a failure writing out the file.

Operation

The Manager, Sender, and Receiver are not required to operate on the same platform. Furthermore, the request does not have to be made on the system where the source data resides. The request can be made from any node in the infrastructure. Data Integrator will resolve the source and target destinations.

The status sub-system is responsible for directing the status messages

to the appropriate systems. There are several queries and interfaces available to present and manipulate the data.

Processing flow

Figure 1 illustrates the processing flow of a typical transfer. Notice that the exit points are executed at each of the components before and after processing. This allows total flexibility in user process integration.

THE STRUCTURE OF EXITS

Outline

eNDI uses the dynamic load library (DLL) facilities available on all platforms that support eNDI. The components dynamically load the

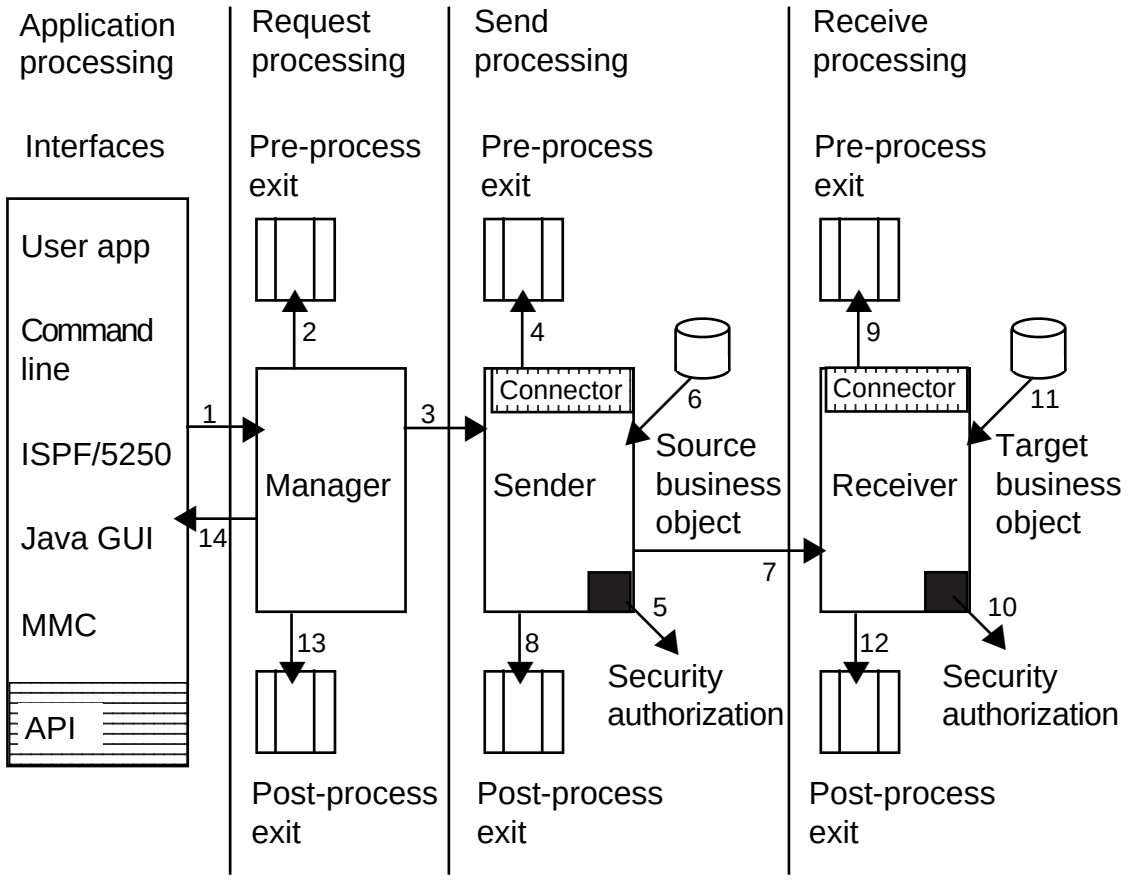

Figure 1: Processing flow of a typical transfer

user-specified exit and execute the specified functionality. All user-exit functionality must be compiled into DLLs and the entry points must be exported according to the platform specifications. (See Compiling and Generating DLLs, below.) eNDI invokes the entry points with arguments according to the specific exit number. All exit modules (except those for the VM Connector) must be written in the C language. The specific data structures for each of the exits are found in the *ftfc.h* that resides in the eNDI directory. These data structures and all the API calls are described in detail in the product documentation.

Headers

The header file *ftfc.h* is included in the development toolkit supplied with the eNDI product. This header file in turn uses some MQSeries definitions, so it includes the MQSeries header file *cmqc.h*. The file *ftfc.h* defines many constants, structures, and function prototypes that are used by the API; it should always be kept up-to-date when installing new versions of eNDI.

Parameters passed to the exit

When an exit is invoked by eNDI through its published entry point, a standard set of parameters is passed to that entry point.

Note that none of the transfer parameters can be altered by the eNDI exits, so they cannot be used for dynamic routing or load balancing after the transfer has started. Any such utilities would have to be invoked before making the transfer request.

FTFExitInfo

This structure is passed to all types of exit. It is defined as follows:

```
typedef struct _FTFExitInfo
{
   FTFCHAR  ftfid[FTF_MAX_FTFID_SIZE];
   FTFLONG  exitNumber;
   FTFCHAR  *pDllName;
   FTFCHAR  *pEntryPoint;
   FTFCHAR  *pUserId;
   FTFCHAR  *pPassword;
   FTFLONG  cUserData;
   FTFCHAR  *pUserData;
   FTFRC    rc;
   FTFRC    rc2;
```

```
   FTFVOID  *pInternal;
   FTFIdentifiersInfo Ids;
} FTFExitInfo;
```

The *ftfid* parameter holds the unique identifier for the data transfer currently underway. This identifier is passed along with all stages of the transfer and can be used to get information relating to it from the status sub-system.

The *exitNumber* parameter informs the exit code of the particular exit that is being invoked. This is useful when writing code for multiple exit types using the same entry point. The exit numbers are as follows:

- 1,2 – reserved for special compression/decompression.
- 3,4 – manager's pre- and post-processing.
- 5,6 – sender's pre- and post-processing.
- 7,8 – receiver's pre- and post-processing.
- 9,10 – sender and receiver connector.

This article will deal with exit numbers three to eight.

The *pDllName* and *pEntryPoint* elements are pointers to the name of the currently executing dll and the code entry point.

The *pUserId* and *pPassword* elements are pointers to the current user id and password in use by the eNDI component during the current transfer. The password is not currently implemented but may be used in future releases.

The *pUserData* and *cUserData* elements carry data that has been passed by the transfer request relating to the exit. This equates to the text passed on the command line in the *–exitdata* parameter, and the *cUserData* is the length of that data. In many exits and connectors this acts as parameters to the user exit code and modifies its behaviour in some way. An example of this is in the supplied Message Connector, where the *–exitdata* parameter allows you to specify whether the data is to be split up by record length, tags, or delimiters, whether the messages are to be persistent, etc.

rc1 and *rc2* contain primary and secondary return codes. *rc1* should be tested on entry to the user code, particularly for Manager post-process exits as this reflects the status of the transfer so far. By setting

these return codes the user can influence the overall transfer and cause it to succeed or fail. An example of this may be, say, the sender pre-process exit extracts the data from a database to a file, then the sender process sends that file. If the database is currently off-line and the extract fails, by returning a non-zero value in *rc1* and specifying your own specific value indicating 'database off-line' the transfer will not proceed beyond that point. The status sub-system will indicate a sender exit failure and display your user failure code. I will go into more detail about how to place entries into the eNDI status sub-system later.

Note that if either the sender or receiver nodes have a failure the Manager post-process exit is always executed. This is why the *rc1* parameter should be checked on entry to a manager post-process exit.

Note also that, as the Manager post-process exit has the transfer status passed to it, this can be used to interface to external system monitoring tools, such as Unicenter or Tivoli, to give detailed information about the eNDI transfers.

The *FTFIdentifiersInfo* is a structure containing three identifiers, which are populated by user data passed into the request. These can be any value that the user chooses to employ. Unlike the *pUserData* parameter, these values do not relate to one specific exit; instead, the same data is passed to all exits and connectors.

FTFExitRequestInfo

This structure is passed as the second parameter to Manager pre and post exits only; it contains all the current information about the data transfer in flight and is defined as follows:

```
typedef struct _FTFExitRequestInfo {
   FTFExitQMgrsInfo       *pQMgrs;
   FTFExitSourceFileInfo  *pSource;
   FTFExitTargetFileInfo  *pTarget;
   FTFExitJobInfo         *pJob;
   FTFExitUserInfo        *pUser;
} FTFExitRequestInfo;
```

The *FTFExitQMgrsInfo* structure contains the names of the queue managers of the four eNDI nodes involved in the transfer.

FTFExitJobInfo contains details of some of the options used to

determine the behaviour and display of the transfer job. Many of these options are for the more advanced functions of eNDI, which are beyond the scope of this article.

The *FTFExitSourceFileInfo* and *FTFExitTargetFileInfo* structures are dealt with below.

The *FTFExitUserInfo* structure contains the transfer label and group name. The label is defined by the user as part of the request parameters (*-label* on the command line) and *groupName* is an internal eNDI value.

FTFExitSourceFileInfo

This structure is passed as the second parameter to the sender pre- and post-processing exits and the manager's exits as described above. It contains all the relevant details about the source file. Note that the sender has no details about the target file or the other job-related information, only the source file.

```
typedef struct _FTFExitSourceFileInfo {
   FTFCHAR         *pFileName;
   FTFBOOL         isStaged;
   FTFBOOL         isStagePersistent;
   FTFBOOL         isDataPersistent;
   FTFBOOL         isCompressed;
   FTFFileTypeInfo fileType;
   FTFLONG         bufNo;
} FTFExitSourceFileInfo;
```

This structure holds information about how the file is to be read and transferred. The *pFileName* parameter is a pointer to the full path of the source file. The two staging values are for when the transfer is staged at an intermediate point. With eNDI, the data is normally transmitted as non-persistent messages (these are fully recoverable, but, again, beyond the scope of this document), but this can be overridden and this value passes its current state. Transfers can be compressed by eNDI and the 'isCompressed' value passes the requested option for this transfer. The fileType is either *FTF_TEXT* or *FTF_BINARY* and *bufNo* is an internal value only used on OS/390.

FTFExitTargetFileInfo

This structure is passed as the second parameter to the receiver's pre- and post-processing exits and the manager's exits. It contains all the

relevant details about the target file. Note that the receiver has no details about the source file or the other job-related information, only the target file.

```
typedef struct _FTFExitTargetFileInfo {
   FTFCHAR             *pFileName;
   FTFBOOL             isCompressed;
   FTFFileTypeInfo     fileType;
   FTFFileModeInfo     fileMode;
   FTFFileOrgInfo      fileOrg;
   FTFLONG             cDirectoryBlocks;
   FTFRecordFormatInfo recordFormat;
   FTFLONG             lrecl;
   FTFLONG             blockSize;
   FTFCHAR             unitName [FTFREQ_UNITNAME_SIZE];
   FTFCHAR             volser   [FTFREQ_VOLSER_SIZE];
   FTFAllocUnitInfo    allocationUnit;
   FTFLONG             primaryAllocSize;
   FTFLONG             secondaryAllocSize;
   FTFTextFileWrapInfo textWrapRecord;
   FTFBOOL             createDirectory;
   FTFBOOL             isDataPersistent;
   FTFLONG             bufNo;
} FTFExitTargetFileInfo;
```

Much of this information is related to the particular file type and organization of OS/400 and OS/390 file types. Apart from those, the type of information is very similar to the source file information: file name, compression value, binary/text value, and persistence.

Exit definitions

Manager pre and post process

A typical definition for a manager pre or post exit is as follows:

```
DLLEXPORT void MyMgrExit(     FTFExitInfo *pExitInfo,
                              FTFExitRequestInfo *pInfo)
```

This exposes the entry point to the compiler. Obviously, for this to be portable across platforms, the *DLLEXPORT* is defined using the compiler switches based on the target platform. For Win32 this is:

```
#define DLLEXPORT _declspec(dllexport)
```

For MVS, declarations similar to the following are required:

```
#define DLLEXPORT
#pragma export(MyMgrExit)
```

For Unix, all the non-static functions are exposed, so a null definition of *DLLEXPORT* is all that's required:

```
#define DLLEXPORT
```

Sender pre and post process

```
DLLEXPORT void MySdrExit(     FTFExitInfo *pExitInfo,
                              FTFExitSourceFileInfo *pInfo)
```

Receiver pre and post process

```
DLLEXPORT void PopupRcvExit(  FTFExitInfo *pExitInfo,
                              FTFExitTargetFileInfo *pInfo)
```

Status messages

Status messages are a vital part of eNDI. The e-Adapter status subsystem manages messages that describe the status of an e-Adapter data transfer request. These status messages can be generated by e-Adapter or from within any user exit or connector. In order to allow your business process to fully integrate into the data transfer, an API call is available to place status messages into eNDI. The API call is defined as follows:

```
FTFVOID FTFSubmitStatusMsg    (FTFExitInfo *info,
                        FTFCHAR *customComponent, FTFCHAR *customType,
                        FTFCHAR *errorText, FTFCA *ftfca);
```

The pointer to the *FTFexitInfo* structure, which was passed to the exit, must be passed back to this call. The second parameter allows the user to specify a name for the component, such as 'My Custom Exit'. The next parameter is also text, which appears in the custom status field and should be used for a brief description about what task the exit is performing, for example 'Processing Source Data'. The third parameter is again text, and appears in the 'error' field of the status message. An example of this may be "Processing failed – unable to access data 'MyData'".

The last parameter is a structure containing the return codes of the function call.

Processing

If the user exit function is designed to run specifically as, say, a sender pre-process exit then the first piece of processing work to do is to

check whether this is actually being invoked at the correct exit point. The following code does exactly this by comparing the actual exit number with the exit type expected.

```
if (pExitInfo->exitNumber != FTF_EXITNO_SOURCE_PRE)
{
   pExitInfo->rc = 200;   /*May or may not wish to fail the transfer*/
   sprintf(str,"MyExit - Exit no %d", pExitInfo->exitNumber);
   FTFSubmitStatusMsg(pExitInfo, str,
         "Exit not executed", "Must be sender pre", &ftfca);
   return;
}
```

All the entry points take the same first parameter, which means that any of the *ExitInfo* parameters will be valid as long as the code is called by eNDI.

If the same code is to be called by all exit types it is still necessary to make three different entry points, as in the following example:

```
DLLEXPORT void PopupMgrExit(   FTFExitInfo *pExitInfo,
FTFExitRequestInfo *pInfo)
{
   ExitCode(pExitInfo, pInfo->pSource->pFileName,
               pInfo->pTarget->pFileName);
}
DLLEXPORT void PopupSdrExit(FTFExitInfo *pExitInfo,
                            FTFExitSourceFileInfo *pInfo)
{
   ExitCode(pExitInfo, pInfo->pFileName, NULL);
}
DLLEXPORT void PopupRcvExit(   FTFExitInfo *pExitInfo,
FTFExitTargetFileInfo *pInfo)
{
   ExitCode(pExitInfo, NULL, pInfo->pFileName);
}
```

In the example above, the function *ExitCode()* takes the *pExitInfo* parameter, the source file name, and the target file name. The *pExitInfo* is needed for calls to the status sub-system and any other information specifically relating to the exit in operation. Both source and file name are only available to the manager exits so the *ExitCode()* function should be designed to handle NULL values in either of these parameters.

Return value

When processing is complete, a value must be placed in the *pExitInfo->rc* variable, which indicates to the eNDI component the success or

otherwise of the user process. If a non-zero value is returned the transfer will stop executing at that point and indicate a failure to the overall status system. As mentioned earlier, in the event of a failure at any component the manager post process exit is still invoked.

Synchronous or asynchronous?

eNDI exits can be invoked only as a synchronous process, so the transfer is blocked until the code returns control to the calling component. To run an exit asynchronously it is necessary to spawn or fork a new process, then return from the original one. There may be a good reason for doing this, but it should only be done if the transfer's success is not dependent on the success of the user process.

Compiling and generating DLLs

For Windows development using Visual Studio or Visual C++ a new project should be defined, specifying an empty project of type 'Win32 dynamic link-library'. Unless extensive use of MFC or other Windows utility functions is required, all the libraries specified in the link section can be removed except for *mqm.lib*, *ftflib.lib* and *ftfxmoma.lib*. Obviously, the path to the eNDI and MQSeries libraries needs to be set up using the directories tab under the tools/options menu.

When compiling under Unix systems the code must be compiled for sharing (+z on HP, no extra options for Solaris) and linked as a shared library (-b for HP, -G for Solaris). Make sure that the linked library has execute access and that the MQSeries shared libraries and the *libftf.so* (or *.sl* depending on platform) are in the *LD_LIBRARY_PATH*.

SAMPLE PROGRAM 1 – DISPLAY TRANSFER DETAILS (WINDOWS GUI)

This program is very simple, but is designed to demonstrate many of the features discussed above.

It is a Win32 dll named *Popup.dll*, which displays a Windows message box displaying the contents of some of the important parameters. It is designed to operate on any of the exit points during a transfer and can, if desired, be used on all the exit points, demonstrating graphically the process flow.

POPUP.DLL

```
/* Standard includes */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <afxwin.h>
/* eNDI includes      */
#include "ftfc.h"
#if defined(__cplusplus)
   extern "C" {
#endif
static void ExitCode(FTFExitInfo *pExitInfo, char *spath, char *dpath);
#ifdef WIN32
   #define DLLEXPORT _declspec(dllexport)
#endif
/* The 3 entry points for manager, sender, and receiver respectively */
DLLEXPORT void PopupMgrExit(   FTFExitInfo *pExitInfo,
FTFExitRequestInfo *pInfo)
{
   ExitCode(pExitInfo, pInfo->pSource->pFileName, pInfo->pTarget-
>pFileName);
}
DLLEXPORT void PopupSdrExit(FTFExitInfo *pExitInfo,
FTFExitSourceFileInfo *pInfo)
{
   ExitCode(pExitInfo, pInfo->pFileName, NULL);
}
DLLEXPORT void PopupRcvExit(FTFExitInfo *pExitInfo,
FTFExitTargetFileInfo *pInfo)
{
   ExitCode(pExitInfo, NULL, pInfo->pFileName);
}

/* Text definitions for the exit names for use in status displays */
static char *ExitNames[] =
{   "Compression", "Decompression", "Manager Preprocess", "Manager
Postprocess",
   "Sender Preprocess", "Sender Postprocess", "Receiver Preprocess",
   "Receiver Postprocess", "Sender Connector", "Receiver Connector"
} ;
/* The main part of the code, common to all the exits */
static void ExitCode(FTFExitInfo *pExitInfo, char *spath, char *dpath)
{
   char *str = (char *)calloc(1,2000);
   static char *unknown = "Unknown";
   FTFCA ftfca;
   int rc = true;
   /* Check for bad pointer to the main exit information */
   if (!pExitInfo)
      return;
```

```
   /* Format the string to display in the message box */
   sprintf(str,
      "Demo Popup %s Exit in progress. \n\n"
      "FTFID:\t\t%s\nDLL name:\t%s\nEntry Point:\t%s\n"
      "User Id:\t\t%s\nPassword:\t%s\n"
      "Source File:\t%s\nDest File:\t\t%s\n"
      "Id1:\t\t%s\n"
      "Id2:\t\t%s\n"
      "Id3:\t\t%s\n"
      "rc1 = %-5ld\trc2 = %-5ld\n",
      ExitNames[pExitInfo->exitNumber-1],
      pExitInfo->ftfid,pExitInfo->pDllName, pExitInfo->pEntryPoint,
      pExitInfo->pUserId ? pExitInfo->pUserId : unknown ,
      pExitInfo->pPassword ? pExitInfo->pPassword : unknown,
      spath ? spath : unknown, dpath ? dpath : unknown ,
      pExitInfo->Ids.Id1, pExitInfo->Ids.Id2, pExitInfo->Ids.Id3,
      pExitInfo->rc, pExitInfo->rc2
      );
   /* Check that the transfer hasn't already failed somewhere else */
if (pExitInfo->rc!=Ø)   /* This can only happen for a post process exit
*/
   {
  strcat( str, "User exit not executed - FTF failed" );
  AfxMessageBox(str, MB_OK | MB_ICONSTOP );
  sprintf(str,"Demo Popup %s Exit", ExitNames[pExitInfo->exitNumber-1]);
      FTFSubmitStatusMsg(pExitInfo, str, "Exit not executed",
               "User exit not executed - FTF failed", &ftfca);
   }
   else
   {
/* We're ok up to this point - now give the user a choice to succeed or
fail */
   strcat( str, "Make this exit succeed?");
   rc = AfxMessageBox(str, MB_YESNO | MB_ICONQUESTION );
   sprintf(str,"Demo Popup %s Exit",ExitNames[pExitInfo->exitNumber-1]);
      if (rc == IDYES)
      {
      /* User chose to succeed - good return value and status message */
         pExitInfo->rc = Ø;
         FTFSubmitStatusMsg(pExitInfo, str, "User Input - PASS",
               "User choice: Pass", &ftfca);
      }
      else
      {
      /* User chose to fail - bad return value and status message */
         pExitInfo->rc = -123;
         FTFSubmitStatusMsg(pExitInfo, str,
            "User Input - FAIL", "User choice: Fail", &ftfca);
      }
   }
   /* Clean up & return */
```

```
   free(str);
   return;
}
#if defined(__cplusplus)
   }
#endif
```

Command Line

To invoke this exit on the command line, the following options can be used:

```
ftf -label Popups -oqm M_QMGR -lqm L_QMGR -sqm S_QMGR -dqm D_QMGR -spath
c:\hello.txt -dpath c:\hello.txt -ID1 "This is ID1" -ID2 "This is ID2" -
ID3 "This is ID3" -exit 3 -exitdll Popup.dll -exitentry PopupMgrExit -
exit 4 -exitdll Popup.dll -exitentry PopupMgrExit -exit 5 -exitdll
Popup.dll -exitentry PopupSdrExit -exit 6 -exitdll Popup.dll -exitentry
PopupSdrExit -exit 7 -exitdll Popup.dll -exitentry PopupRcvExit -exit 8
-exitdll Popup.dll -exitentry PopupRcvExit
```

Notes

This command line specifies that, to invoke the command, it will connect directly to *L_QMGR*, specifying *M_QMGR* as the transfer manager, *S_QMGR* as the source of the data, and *D_QMGR* as the destination. The *spath* and *dpath* specify the source and destination filenames, and the identifiers *ID1* to *ID3* are specified.

For each of the exits the exit number must be specified first, followed by the *-exitdll* (the name) and *-exitentry* (the code entry point) parameters. In this example no parameters are passed to the dll, so the *–exitdata* parameter is not used.

Note that when eNDI is started as a Windows service, there is a choice whether the service is allowed to interact with the Windows desktop. Unless the eNDI service is set to allow this, the popup message boxes will not be visible, but will block the eNDI transfers. This is true for all interactive exits, so care should be taken on how the service is started.

SAMPLE PROGRAM 2 – START A SYSTEM COMMAND (WINDOWS AND UNIX)

This program is designed to be portable to any platform. On OS/390 it only prints the system command to the console. Full job submission

is more complicated and would not serve the purpose of clarifying how to write eNDI exits. The exit will start a command line specified in the *–exitdata* transfer parameter and performs some macro substitutions. For example, *$dpath* on the command line gets substituted for the destination path specified in the transfer. It also allows for synchronous or asynchronous command execution and the ability to pass the command-line return code back to the eNDI component to determine the transfer success.

CMDEXIT.DLL (OR CMDEXIT SHARED LIBRARY ON UNIX)

```
/* Standard includes */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
/* eNDI includes      */
#include "ftfc.h"
/* Platform-specific definitions for function export */
static void ExitCode(FTFExitInfo *pExitInfo, char *spath, char *dpath);
#ifdef WIN32
   #define DLLEXPORT _declspec(dllexport)
#elif defined(UNIX) || defined(OS2)
   #define DLLEXPORT
#elif defined(OS400)
   #define DLLEXPORT
#elif defined(MVS)
   #define DLLEXPORT
   #pragma export(CmdMgrExit)
   #pragma export(CmdSdrExit)
   #pragma export(CmdRcvExit)
#endif
#if defined(__cplusplus)
   extern "C" {
#endif
DLLEXPORT void CmdMgrExit(FTFExitInfo *pExitInfo, FTFExitRequestInfo
*pInfo)
{
   ExitCode(pExitInfo, pInfo->pSource->pFileName, pInfo->pTarget-
>pFileName);
}
DLLEXPORT void CmdSdrExit(FTFExitInfo *pExitInfo, FTFExitSourceFileInfo
*pInfo)
{
   ExitCode(pExitInfo, pInfo->pFileName, NULL);
}
DLLEXPORT void CmdRcvExit(FTFExitInfo *pExitInfo, FTFExitTargetFileInfo
*pInfo)
```

```
{
   ExitCode(pExitInfo, NULL, pInfo->pFileName);
}
static void ExitCode(FTFExitInfo *pExitInfo, char *spath, char *dpath)
{
   char commandString = calloc(1,2000);  /* The command string */
   char statString = calloc(1,200);      /* The status message string */
   char *tmpstr = NULL, tmpnum[30];      /* Temporary storage */
   int systemrc=0;                       /* System return code */
   unsigned int i, j;                    /* Counters */
   FTFCA ftfca;                          /* FTFSubmitStatusMsg return */
   int UseRet = FALSE;           /* Switch to use the cmd return code */
   if (pExitInfo->pUserData)
   {
      /* Allocate some space for temporary string */
      tmpstr = malloc(pExitInfo->cUserData + 1);      /* Allocate it */
/* Copy the user data into temporary area */
sprintf(tmpstr, "%.*s", pExitInfo->cUserData ,pExitInfo->pUserData);
      /* Substitute any $sysret, $spath and $dpath variables for actual
paths...*/
#ifdef WIN32
if (!strstr(tmpstr,"$async "))   /* On Windows start new process if
$async */
strcpy(commandString,"start ");
#endif
      /* The next test is if the userdata contains $fail.
      *   If so, don't execute if transfer so far failed    */
      if (strstr(tmpstr,"$fail “) && pExitInfo->rc!=0)
      {
         sprintf(statString,"User command not executed - FTF failed"
                      " (Code %ld) and $fail specified”,pExitInfo->rc);
         sprintf(tmpnum,"EXIT %ld SYSTEM CMD”, pExitInfo->exitNumber);
         FTFSubmitStatusMsg(pExitInfo, tmpnum, "RETURN CODE",
statString, &ftfca);
         return;
      }
      /* Now make the macro substitutions if present */
      if (strstr(tmpstr,"$sysret") || strstr(tmpstr,"$spath") ||
                                      strstr(tmpstr,"$dpath"))
      {
         for (i=0, j=strlen(commandString); i<strlen(tmpstr); i++)
         {
 if (!strncmp(&tmpstr[i],"$sysret ",8))/* sysret return code required */
           {
          /* Use the system command return code for eNDI return code. */
          /* Don't pass this to command */
               i += 7;            /* Skip over $sysret and space */
               UseRet = TRUE;
               continue;
           }
 if (!strncmp(&tmpstr[i],"$fail ",6))/* sysret return code required */
```

```
            {
 /* If eNDI has failed, don't do the exit. Don't pass this to command */
               i += 5;            /* Skip over $fail and space */
               continue;
            }
#ifdef WIN32
            if (!strncmp(&tmpstr[i],"$async ",7))/* Asynchronous process
required */
            {
               /* NT only - choose sync/async exit, (used above). */
               /* Don't pass this to command */
               i += 5;            /* Skip over $sync and space */
               continue;
            }
#endif
      /* Look for source path substitution - can only do this on mgr &
sdr exits */
if (!strncmp(&tmpstr[i],"$spath",6) && spath)/* spath substitution reqd
*/
            {
       strcat(commandString, spath);   /* Append the real source path */
               i += 5;                 /* Skip over '$spath' */
       j = strlen(commandString);   /* Reset the command string index */
               continue;
            }
/* Look for dest path substitution - can only do this on mgr & rcv exits
*/
if (!strncmp(&tmpstr[i],"$dpath",6) && dpath)/* spath substitution reqd
*/
            {
       strcat(commandString, dpath); /* Append the real traget path */
       i += 5;                        /* Skip over '$dpath' */
       j = strlen(commandString);   /* Reset the command string index */
               continue;
            }
         commandString[j++] = tmpstr[i];   /* Just copy the next byte */
         commandString[j] = '\0';      /* Add a trailing terminator */
         }
      }
      else
strcat(commandString, tmpstr);/* No special macros, just the command */
      /* Make up the command string as requested */
#if defined (UNIX)|| defined (OS400) || defined (WIN32)
      systemrc = system(commandString);
#elif defined (MVS) || defined(OS2)
      printf("%s\n", commandString);
#endif
      sprintf(statString,"Command \"%.900s\" returned
(%d)",commandString, systemrc);
      sprintf(tmpnum"EXIT %ld SYSTEM CMD",pExitInfo->exitNumber);
      FTFSubmitStatusMsg(pExitInfo, tmpnum, "RETURN CODE", statString,
```

```
&ftfca);
   }
   /* If $sysret specified, use the system return code for exit error
level.   */
   /*   The eNDI component will check this code on return.
*/
   if (UseRet)
      pExitInfo->rc = systemrc;
   return;
}
#if defined(__cplusplus)
   }
#endif
```

Command Line

To invoke this exit on the command line to execute the *notepad.exe* utility to display the destination file after it has arrived at the receiver, the following command line could be used:

```
ftf -label "Command Line" -oqm M_QMGR -lqm L_QMGR -sqm S_QMGR -dqm
D_QMGR -spath c:\hello.txt -dpath c:\hello.txt -exit 8 -exitdll
CmdExit.dll -exitentry CmdRcvExit -exitdata "notepad $dpath"
```

Notes

This command line specifies that to invoke the command it will connect directly to *L_QMGR*, specifying *M_QMGR* as the transfer manager, *S_QMGR* as the source of the data, and *D_QMGR* as the destination. The *spath* and *dpath* specify the source and destination filenames.

On this command line, the *–exitdata* parameter is specified, which, for this exit, determines what process is executed. Note that, when using Windows, a GUI process such as notepad is always executed asynchronously when using the system() call, so it is not possible to get the return code from the process. A command script will be executed synchronously unless the macro *$async* is specified on the command line, in which case it is started as a separate process.

Valid macro substitutions are:

- *$async* – Windows only. Starts a command script with ‘start’ at the beginning of the command line. The text will be taken out of the command passed to the system.

- *$spath* – substitutes the source path value in place of this macro. This value is not available to either receiver exit.
- *$dpath* – substitutes the destination path value in place of this macro. This value is not available to either sender exit.
- *$sysret* – when specified, will use the return code from the system() call to pass back as the return code of the exit.
- *$fail* – if specified, the exit will check the current status of the transfer and not execute the exit if the current status is non-zero.

This is not intended to be a complete command-execution exit, it serves merely as an example of how exits are structured. There is a similar exit supplied with the product that also supports OS/390 (and all other platforms).

SUMMARY

enableNet Data Integrator is a powerful – possibly essential – tool for file transfer in an MQSeries environment. With the exploitation of the exit and connector capabilities the product becomes a focal point for enterprise data integration. The connectors allow data to be extracted or inserted from alternative sources to the file system, while the user exits can be exploited to fully integrate the data transfer into the overall business process flow.

This article illustrates the simplicity with which enableNet Data Integrator exits can be written. Hopefully, it will enable current customers to fully exploit its features and potential customers to see the benefits of the product over and above the file transfer capability.

Chris Howarth
Senior Systems Engineer, CommerceQuest (UK)

MQ message throughput reporting on AIX

How do you know how much work is going through your MQ infrastructure? Determining how busy your MQ infrastructure is can be a difficult exercise. Monitoring CPU use and message throughput

can be a valuable aid in capacity planning, but an easy, automated method is required to gather this information and track it over time.

This article provides scripts and jobs for automated collection of message throughput statistics across MQ channels. We are running Version 4.3.3.0 of AIX and Version 5.1 of MQ. The collection script is written using the korn shell for use on AIX, but the logic of the script could be ported to other platforms. The output of the script is transferred using FTP to our mainframe and is then processed by SAS, but it can be processed by any data manipulation tool, eg Excel.

The message throughput statistics can then be correlated with CPU utilization figures for capacity forecasting.

CHANNEL STATISTICS

MQ provides useful information about message throughput in the channel statistics. This can be accessed by running the **DIS CHSTATUS** command, eg:

```
DIS CHSTATUS(QP01.TO.QP02) msgs
AMQ8417: Display Channel Status details.
   CHANNEL(QP01.TO.QP02)                   XMITQ(QP02)
   CONNAME(127.0.0.1)                      CURRENT
   CHLTYPE(SDR)                            STATUS(RUNNING)
   MSGS(344989)
```

There have been 344,989 messages sent across channel *QP01.TO.QP02* since it started. Note that the statistics are reset when the channel is stopped or becomes inactive. If you set the disconnect interval to a reasonable length of time or to 0, this should not be an issue. The collection script interval can also be set quite short, which should avoid any potential problems.

The following collection script runs the **DIS CHSTATUS** command for a specified MQ channel. It checks the number of messages sent (or received) every specified interval, eg 600 seconds. It writes the gathered information to a history file.

COLLECTION SCRIPT

```
#!/bin/ksh
# Channel message counts
# Captures message counts across named channel.
```

```
# Input arguments
channel=$1        #channel name e.g. QP01.TO.QP02
qmgr=$2           #Queue Manager name e.g. QP01
interval=$3       #Gather stats every x seconds eg 600
days=4            # days history to keep
outfile=/data/MQM/data10/defs     # directory for output file
trap 'rm temp_$$ ; exit' 0 1 2 15 # removes temp files on exit (clean or
otherwise)
echo 'dis chstatus('$channel') all' > temp_$$  # create temp file as
input into runmqsc
echo end >>temp_$$
oldcount=`runmqsc $qmgr < temp_$$ | grep ' MSGS(' | cut -c9-|cut -d ")"
-f1` # initialize msgs
                                                                  # count
if [ -z "$oldcount" ]   # if oldcount not set by runmqsc
                        #ie the channel  is stopped  then set to 0.
  then
  oldcount=0
fi
tot=0
echo  Trans Total Date\(dd/mm/yyyy\) Time  Channel >> $outfile/$channel
# column headings
                                                                      #
in output file
h=`date +%H`    # calculate number of intervals to do till midnight
m=`date +%M`
s=`date +%S`
i=`expr $h \* 3600  + $m \* 60 + $s`
# calculate number of iterations loop should do till end of the day
endofday=`expr 86400 - 600 - $interval` #midnight minus 10 minutes and 1
interval,
                                        #allow for drift in the loop
while [ $i -lt $endofday ]    # loop until just before midnight,
          # don't check time every loop because of the overhead
do
  count=`runmqsc $qmgr < temp_$$ | grep ' MSGS(' | cut -c9-|cut -d ")" -
f1`
  if [ -z "$count" ]       # if count not set by runmqsc function then
set to 0.
    then
    count=$oldcount
  fi
  if [ $count -lt $oldcount ]  # channel stopped/disconnected
    then
    count=0
    oldcount=0
  fi
  date=`date +%d/%m/%Y`   # dd/mm/yyyy e.g. 18/07/2001
  time=`date +%H:%M:%S`   #  hh:mm:ss e.g. 15:32:38
  trans=`expr $count - $oldcount`
```

```
  tot=`expr $tot + $trans`
  echo $trans $tot $date $time $channel  >> $outfile/$channel
  if [ "$count" ]  # check that count was set okay by the runmqsc
function.
  then
   oldcount=$count
  fi
  sleep $interval
  i=`expr $i + $interval`
done
# Housekeeping
tail -n   `expr '(' 86400 /  $interval ')' \* $days` $outfile/$channel >
t_$$
                                                        # keep $days days
cp t_$$ $outfile/$channel
rm t_$$
# End of script
```

SCHEDULING

Schedule this script using crontab:

```
10 00 * * * /data/MQM/data10/scripts/chanmess.sh QP01.TO.QP02 QP01 600
```

This runs the script at 12.10 am every day, with an interval of 600 seconds. Set the interval lower if your channels disconnect more frequently.

HISTORY FILE

The output from the script is written to a history file with the same name as the channel in the *$outfile* directory specified in the script, eg */data/MQM/data10/defs/QP01.TO.QP02.*

The file output is:

```
Trans Total Date(dd/mm/yyyy) Time Channel
0 0 18/07/2001 00:10:01 QP01.TO.QP02
14 14 18/07/2001 00:20:01 QP01.TO.QP02
21 35 18/07/2001 00:30:02 QP01.TO.QP02
23 58 18/07/2001 00:40:02 QP01.TO.QP02
11 69  13/07/2001 00:50:03 QP01.TO.QP02
```

As you can see, one record is written every interval with the number of messages sent during the interval, a running total since the start of the day, the interval end time, and the channel name.

The script performs housekeeping on this file, keeping a number of days' history specified by the *$days* variable.

If the channel stops/disconnects during the collection interval a value of 0 will be recorded for the message count. You may have to experiment with the interval time and the disconnect interval to optimize your collection.

POST PROCESSING

The history file can be transferred daily using FTP and processed by any data manipulation product, in this case SAS on the mainframe.

An example of the FTP job is detailed here.

```
//MQMCDTØ5 JOB ,ISOS-TSTMW-RAB,CLASS=4,MSGCLASS=J
//* FTP Channel message counts.
//FTP      EXEC PGM=FTP,REGION=2Ø48K,
//     PARM='172.31.81.81 (timeout 18Ø'
//STEPLIB  DD DSN=SYS1.SEZATCP,DISP=SHR
//SYSTCPD  DD DSN=TP.VOTCPBP.PARMLIB(TCPDATA),DISP=SHR
//SYSFTPD  DD DUMMY
//NETRC    DD DUMMY
//SYSMDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//OUTPUT   DD SYSOUT=*
//INPUT    DD *
userid
password
cd  /data/MQM/data1Ø/defs/
get QPØ1.TO.QPØ2      'mq.prod.msgs(chan1)'   (REPLACE
get QPØ2.TO.QPØ1      'mq.prod.msgs(chan2)'   (REPLACE
QUIT
```

This job is automatically scheduled to run every day at 6.00 am.

SAS PROCESSING

The first job creates a long-term history database. It keeps 35 days' interval data and one year's daily totals for each channel being monitored.

```
//MQMCDT1Ø JOB ,ISOS-TSTMW-RAB,CLASS=4,MSGCLASS=J
//* MQ Message counts.
//STEP1    EXEC ASV6,OPTIONS='ERRORABEND NOMACROGEN NOSOURCE NOSOURCE2'
//MSGSTAT DD DSN=MQ.PROD.MSGSTAT,DISP=SHR
//NEW      DD DSN=MQ.PROD.MSGSTAT.NEW,DISP=(NEW,CATLG,DELETE),
```

```
//            SPACE=(CYL,(5Ø,5Ø),RLSE),AVGREC=U,
DCB=(RECFM=FS,DSORG=PS,LRECL=27648,BLKSIZE=27648)
//IN       DD DSN=MQ.PROD.MSGS,DISP=SHR
//SYSIN    DD *
  *  PURPOSE    : Message counts across channels                  ;
  %macro chan(file);
   data tis;
   infile in(&file) LENGTH=LEN;
   input @1 LINE £VARYING2ØØ.LEN;
   length channel £4Ø.;
   format date date. time time.;
   drop line;
   if line=:'Trans Total' then delete;
   trans=input(scan(line,1,' '),8.);
   total=input(scan(line,2,' '),8.);
   date=input(scan(line,3,' '),ddmmyy1Ø.);
   time=input(scan(line,4,' '),time8.);
   channel=scan(line,5,' ');
   if date=today()-1;
   label trans='Tran*Count'
        total='Total*Trans Count*Today To Date'
        date='Interval*End*Date'
        time='Interval*End Time'
        channel='Channel*Name';
   run;
   proc append data=tis base=tismch;
   run;
%mend;
%chan(chan1);  /* list each channel history file to process */
%chan(chan2);
proc sort data=tismch;
by channel date time;
run;
data new.dismch;   /* keep 35 days history of interval data */
update msgstat.dismch tismch; /* use update so job can be rerun */
by channel date time;
if intck('day',date,today())<35;
run;
* create daily totals file ;
proc summary data=tismch;
by channel date;
var trans;
output out=tdsmch(drop=_type_ _freq_) sum=;
run;
data new.ddsmch;  /* keep 12 months history of interval data */
update msgstat.ddsmch tdsmch; /* use update so job can be rerun */
by channel date;
if intck('month',date,today())<12;
run;
//*CLEAN UP
//CLEANUP  EXEC  PGM=IDCAMS
```

```
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
  DELETE   MQ.PROD.MSGSTAT
  IF LASTCC = Ø THEN DO
    ALTER  MQ.PROD.MSGSTAT.NEW    -
           NEWNAME(MQ.PROD.MSGSTAT)
  END
//*
```

The second job produces graphs in gif format. The graphs show the daily history of the total message count.

```
//MQMCDT15 JOB ,'ISOS TST MW RAB',CLASS=4,MSGCLASS=X
//*  PURPOSE    : MQ Channel message count graphs
//B1      EXEC  ASV6,OPTIONS='NOERRORABEND'
//MSGSTAT  DD DSN=MQ.PROD.MSGSTATS,DISP=SHR
//GIFS     DD DSN=MQ.PROD.WEBFILE,DISP=(NEW,CATLG,DELETE),
//         DCB=(RECFM=VB,LRECL=132,BLKSIZE=Ø),
//         SPACE=(TRK,(5Ø,5Ø,1ØØ))
//SYSIN    DD *
goptions
  device=gif
  gsfmode=replace
  gsflen=128
  cback=white
  ctext=black
  ctitle=black
  nodisplay
;
axis1
  c=blue
  w=3
  label = none
  value = (font=swiss h=1.5)
;
axis2
  c=blue
  w=3
  label = (font=swiss h=2 a=-9Ø r=9Ø ‘Number of Messages’)
  value = (font=swiss h=1.5)
;
symbol1 i=j l=1 w=3  c=blue  v=none;
symbol2 i=j l=2 w=3  c=red  v=none;
%macro gifs(channel,FILEd);
  * daily peak graphs ;
  title;
  title1 font=swiss h=3 "Daily Message Count History";
  title2 font=swiss h=3 "Channel &channel";
  * dynamic file allocation to a PDS  ;
  filename &filed "mq.prod.webfile(&filed)" disp=shr;
  goptions gsfname=&filed;
```

```
  proc gplot data=msgstat.ddsmch;
  where channel="&channel";
  plot trans*date/haxis=axis1 vaxis=axis2 overlay;
  run;
  quit;
%mend;
*list channels to graph ;
%gifs(QP01.TO.QP02, gp1p2);
%gifs(QP02.TO.QP01, gp2p1);
```

GRAPHIC OUTPUT

Figure 1 shows the regular peaks and troughs of the weekday and weekend workload. The large peak at the start of July is due to a new application going live.

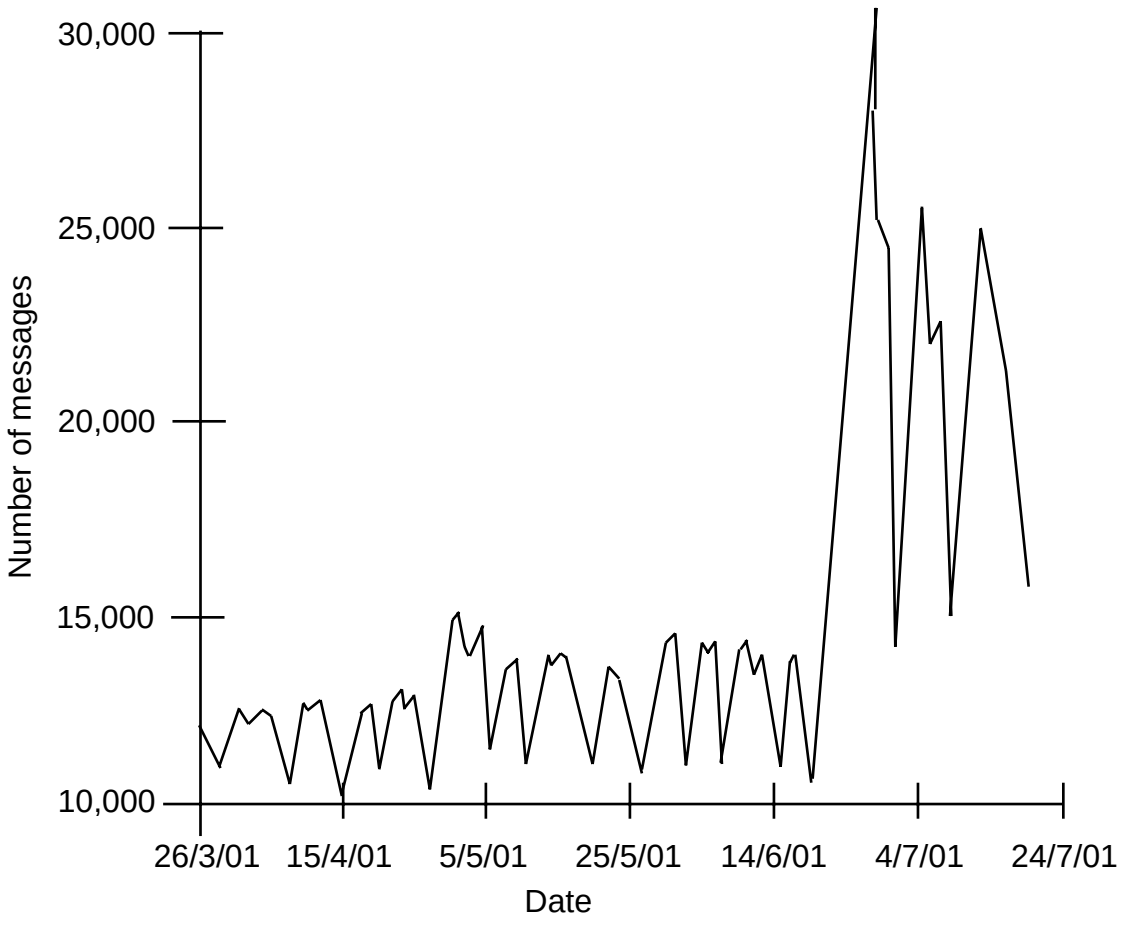

Figure 1: Daily message count history Channel QP01.TO.QP02l

PUBLISHING

This graph can now be published on your Web site. This is a simple process using FTP.

This job is very similar to the previous FTP job but it does a PUT instead of a GET.

```
//MQMCDT2Ø JOB ,'ISOS TST MW RAB',CLASS=4,MSGCLASS=X
//*  PURPOSE    : MQ Channel message count graphs FTP
///FTP     EXEC PGM=FTP,REGION=2Ø48K,PARM='SYSD 21 (TIMEOUT 2Ø'
//STEPLIB  DD DSN=SYS1.SEZATCP,DISP=SHR
//SYSTCPD  DD DSN=TP.VOTCPBP.PARMLIB(TCPDATA),DISP=SHR
//SYSFTPD  DD DUMMY
//NETRC    DD DUMMY
//SYSMDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//OUTPUT   DD SYSOUT=*
//INPUT    DD *
userid
password
quote site umask Ø22
binary
put 'mq.prod.webfile(gp1p2)'  /internal/mq/channel/gp1p2.gif
put 'mq.prod.webfile(gp2p1)'  /internal/mq/channel/gp2p1.gif
```

SAMPLE HTML CODE

This html displays a page with two small graphs, *gp1p2.gif* and *gp2p1.gif*. Clicking on either of these gifs brings up a full-screen version of it.

MAIN PAGE

```
<html>
<head>
<title>PRODUCTION MQ MESSAGE COUNT STATS </title>
</head>
<p align="center"><font color=black><b>CHANNEL MESSAGE COUNT STATS</
b><br></font></p>
<a href=p1p2.htm><img vspace=Ø border=2 src=gp1p2.gif width=48%
height=4Ø%></a>
<a href=p2p1.htm><img vspace=Ø border=2 src=gp2p1.gif width=48%
height=4Ø%></a>
<BR><BR>
</b>
</p>
```

P1P2.HTM PAGE

```
<html>
<head>
<title>PRODUCTION MQ MESSAGE COUNT STATS </title>
</head>
 <p align="center"><font color=black><b>CHANNEL QPØ1.TO.QPØ2 STATS</
b><br></font></p>
<img vspace=Ø border=2 src=gp1p2.gif width=9Ø% height=9Ø%>
p2p1.htm page
<html>
<head>
<title>PRODUCTION MQ MESSAGE COUNT STATS </title>
</head>
<p align="center"><font color=black><b>CHANNEL QPØ2.TO.QPØ1 STATS</
b><br></font></p>
<img vspace=Ø border=2 src=gp2p1.gif width=9Ø% height=9Ø%>
```

Rab McGill
Senior Technician (UK)

What's new in MQSeries Integrator V2.0.2?

Earlier this year, IBM announced an update to MQSeries Integrator (MQSI) V2 with the release of V2.0.2. The formal announcement can be viewed at *http://www.ibm.com/software/ts/mqseries/integrator/v202*.

In this article I want to look at the most notable changes in a little more detail than was covered in the announcement. To describe each of them in depth would take a considerable amount of space and that is not the function of articles such as this: my intent is to highlight issues of particular interest and, hopefully, inspire you to find out more for yourselves. The ultimate aim, of course, is to have you use the product!

AVAILABILITY ON HP-UX AND IBM ISERIES 400

Previous versions of MQSI V2 had support for Windows NT, AIX, and Solaris. This list has now been extended to include the HP-UX and IBM iSeries 400 platforms. As with the other implementations of MQSI V2, the Control Centre and Configuration Manager must run on Windows NT V4 or Windows 2000.

In order to run MQSI V2.0.2 on an HP server you will need year 2000-compliant Hewlett Packard HP-9000 hardware, HP-UX V11.00 with year 2000-fixes, and IBM MQSeries for HP-UX Version 5.1 or later.

Databases supported on this platform are IBM DB2 Universal Database Version 6.1 with ODBC or Version 7.1, Microsoft SQLServer 6.5, 7.0, and 2000, and Sybase Version 12. However, transactional support (coordinated message processing and database updates in a single unit of work) is only available with DB2.

Support on the iSeries 400 is provided through the use of an Integrated xSeries Server rather than a native iSeries implementation. The Integrated xSeries Server for iSeries provides an Intel processor and PC memory, which are contained on a motherboard located within the iSeries 400 server. This allows MQSI for Windows NT V2.0.2 to run on the Intel processor. The Integrated xSeries Server contains an Intel Pentium III 700 MHz and up to 4 GB of main memory. In order to use this Integrated xSeries Server you will need OS/400 V4R5 in one of the following iSeries servers: 270, 820, 830, 840, SB2, or SB3. (See *http://www.as400.ibm.com/windowsintegration/hdwspec.htm* for more details on the Integrated xSeries Server.)

NATIONAL LANGUAGE SUPPORT

MQSI V2.0.2 has extended the number of supported national languages to ten. The user interface and message catalogues support the following languages: Brazilian Portuguese, French, German, Italian, Japanese, Korean, Simplified Chinese, Spanish, Traditional Chinese, and US English. There is a restriction, however, with some of the New Era of Networks' support, which is for US English only. This applies to the Rules and Formatter support nodes and graphical interfaces.

MQSI V2.0.2 is able to process and construct messages in any of the code pages for which MQSeries supports conversion to and from Unicode, on all operating systems.

MQSERIES EVERYPLACE AND SCADA PROTOCOL SUPPORT

Prior to V2.0.2 the only supported input and output message format was that of MQSeries. With V2.0.2 that has changed, and it is now possible to process MQSeries Everyplace and the SCADA Device protocol messages within MQSI V2.

MQSeries Everyplace and the SCADA protocols are specifically designed for use with pervasive computing devices. MQSeries Everyplace is typically used on small handheld devices and PDAs. SCADA is used on small footprint devices which are usually remote and unattended. Such devices typically monitor flow rates or temperatures, for example.

Because there is now support for these new message types there are four new IBM-supplied input and output nodes. These are:

- The MQeInput and MQeOutput nodes to provide support for MQSeries Everyplace.
- The SCADAInput and SCADAOutput nodes for the SCADA protocol.

There have also been modifications to the Publication node to add support for MQSeries Everyplace and the SCADA protocol.

The MQeInput node represents an MQSeries to MQSeries Everyplace bridge. The MQeOutput node allows messages to be written to an MQSeries Everyplace queue.

For those interested in using the MQSeries Everyplace support there is Supportpac ID03 (*http://www.ibm.com/software/ts/mqseries/txppacs/id03.html*), which contains very useful information on how to implement MQSI V2 and MQSeries Everyplace communication. The Supportpac shows how MQSeries Everyplace and MQSeries Integrator work together. It also provides sample code to configure MQSeries Everyplace to MQSeries Integrator communication, as well as a description of how to use that code.

The SCADAInput node represents a SCADA input port. It receives a message from a client connection using the MQIdsp protocol in the format recognized by MQSI V2 and establishes a processing environment for the message.

Although the SCADAOutput node can be used to write an output message it is not normally used for this purpose. The Publication node is used instead.

One restriction to note is that all message flows containing SCADA nodes must be deployed to the same execution group.

MQSeries Everyplace and SCADA applications have significant differences when compared with a typical MQSeries application. It is advisable to read the available documentation before starting to work with either of these new message formats. The *Introduction and Planning Manual* has a useful introduction and a section on special considerations when using this new support.

MESSAGE FLOW DEBUGGING TOOL

In the past, facilities to debug message flows have been limited to user trace and trace nodes. V2.0.2 has added significant new functionality in this area, with a message flow debugger. The debugger is selected by using the debugger option on the Message Flows view in the Control Centre. Using the debugger, it is possible to define breakpoints with a message flow.

The debugger will only work on assigned message flows so you must have a broker defined and running. Using the debugger screen in the Control Centre it is possible to start and stop debugging as well as set options. The breakpoints are set and removed on the message flow panel. There is a SmartGuide to help with the setting of debugger options.

When a breakpoint is encountered, control is returned to the Control Centre user and it is then possible to inspect or modify the message contents. This greatly improves the process of debugging message flows. It provides the ability to see incorrect conditions with filter nodes or see incorrect logic in compute nodes, for example.

There are two modes of operation possible with the debugger. The first is to trace the course of execution of a single message. The second is to set a breakpoint in a particular message flow to catch messages going down a particular path, perhaps an error path for when you do not understand why messages were going along that path, for example. Having set the breakpoint you would process messages. When a message was sent along the path with the breakpoint set, control would be returned to you and you would then be able to examine the message and see why it was going along that error path. The book *MQSeries Integrator Using the Control Centre* has a chapter dedicated to the debugger, which provides valuable information on how to use this new feature.

NEW ERA OF NETWORKS VERSION 5.2

Support for New Era of Networks Rules and Formatter Version 5.2 has been added to MQSI V2.0.2. This brings three new nodes (NEONRulesEvaluation, NEONTransform, NEONMap) and a new parser (NEONMSG).

The new support significantly enhances the function previously available. A statement recently appeared in some publications stating that support for the old nodes (NEONRules and NEONFormatter) and domain parser (NEON) was to be deprecated and discontinued. This is incorrect, as the product 'readme' (available after product installation) makes clear. They are still within the product, and usable.

All of the improvements to Rules and Formatter V5.2 are available within MQSI V2.0.2.

You must follow the migration and administration steps detailed in the MQSeries Integrator V2.0.2 documentation before using either the existing function (NEONRules and NEONFormatter nodes and NEON domain message parser) or the new function (NEONRulesEvaluation, NEONTransform, and NEONMap nodes, and NEONMSG domain parser).

By providing both the existing and new functions, the intention is to enable you to run existing New Era of Networks nodes 'as is', and also gain exposure to the new nodes and function that they bring. The new function is not simply a new implementation of the existing function; it provides additional functionality.

At the core of the new technology is the NEONMSG domain parser, which is the new domain parser for messages defined in the Rules and Formats database. The old domain remains, but is really there for compatibility. The NEONMSG parser has the ability to translate wire-format messages into an MQSeries Integrator message tree. There is also the ability to translate the message tree into a wire-format message. The message tree produced by NEONMSG has a different structure from the NEON parser, so look closely at the documentation. There are also some restrictions that you need to be aware of, so, again, read the manuals closely.

NEONMSG can only parse messages defined as input formats in the Rules and Formats database, so it does not cover all message formats.

It identifies the input format by using the Message Type property on the MQInput node or the <Type> element in a message content descriptor (<mcd>) folder in an RFH2 header.

The NEONRulesEvaluation node is intended as a direct replacement for the old NEONRules nodes. The attributes of the old and new nodes are the same. However, because of some of the changes in the new NEONRules and NEONFormatter function, some difference in behaviour for particular Subscription Actions will be seen.

The NEONTransform node is intended as a replacement for the old NEONFormatter node. It is capable of reproducing the same function as the old node; however, slightly different usage may be required in some situations because of a change to the Rules and Formatter support in MQSI V2.0.2.

In a NEONTransform node a two-stage process takes place. In the first stage, the message data is mapped from the fields of the input format to the fields of a specified target format according to the default mapping or the specified Map Name and Map Version. In the second stage, the node applies the output operations which were specified for each field in the target format (that is as defined in the NEONFormatter User Interface). In stage one, a field would be parsed and potentially reordered; in stage two, it might be translated to upper case (UPPER CASE), for example.

The NEONMap node is identical to the NEONTransform node, but it performs only the mapping stage of a reformat; it would not perform the translation to upper case as described in the example for the NEONTransform node.

SUPPORT FOR MQSERIES 5.2

MQSI V2.0.2 supports the use of MQSeries 5.2. This is good news. It will allow you to pick up the latest MQSeries performance improvements. Laboratory tests have shown significant gains in performance with Version 5.2 compared with Version 5.1, particularly for persistent messages. Logging was rewritten in Version 5.2, and, as a result, syncpoint processing is significantly more efficient. With Version 5.1, only one application at a time was able to commit updates. This meant that the maximum commit rate was largely

constrained by the speed of the disk on which the log was located. With Version 5.2 it is possible for multiple applications to commit updates at the same time. This has led to a significant increase in message throughput rates. Version 5.2 has been available since October 2000.

PERFORMANCE IMPROVEMENTS

There are a number of performance improvements with V2.0.2. These are in the areas of parsing(MQ header parsing), string processing, and Field Handling. These changes are internal to the product and you do not need to do anything to activate the benefits other than use V2.0.2. Laboratory tests have shown improvements of between 10% and 60% for Filter Nodes, Compute Nodes, the MRM, and Publish/Subscribe. The benefits you observe will always vary of course, because of differences in the processing taking place and the environment.

Message throughput using the Message Repository Manager (MRM) is also significantly improved with V2.0.2 compared with V2.0.1. There was a problem with V2.0.1 MRM performance as a result of which an efix was produced. This was called efix 'MSDW01' and had to be applied on top of CSD1 for MQSI V2.0.1. The efix is now rolled into CSD2 for V2.0.1. The benefits of the efix were most significant when converting to or from CWF format messages. The MRM fix is included in V2.0.2. If there are any V2.0.1 installations using the MRM that do not have either the efix or CSD2 installed there is a strong recommendation to update your system and benefit from the performance improvement.

DOCUMENTATION

There have been a number of changes to the product documentation, intended to extend its scope and the way in which it is structured.

Using The Control Centre has been restructured. The book is essentially split into two parts. The tasks that you need to perform, along with details on their execution, are in the first part. The second part describes the concepts that lie behind the tasks. There is a new section on New Era of Networks Rules and Formatter support. The Control Centre tour has been moved to the *Introduction and Planning Book*.

There is a new book called *ESQL Reference*. Much of the material about the use of ESQL, which was in *Using The Control Centre,* has been moved to this new book. The book gives an overview of ESQL, describes key concepts, and shows the syntax of field definitions and ESQL statements. Complex SELECTs, ROWs, and LISTs are discussed, as is querying external databases. The *Appendices* provide additional information, such as the details of ESQL components (Special characters, datatypes, etc), examples (the format of the example message used in much of the book for illustration purposes and use of trace to view message structure), and also details of the MQSeries message header parsers (MQRFH, MQIIH etc).

The *Introduction and Planning Book* contains a number of changes to include information on the new features in this product version. This includes coverage of MQSeries Integrator for HP-UX Version 2.0.2 and a discussion of the capabilities of the new debugger function available in the Control Centre.

The main changes in the *Programming Guide* are coverage of the Plug-in SmartGuide for the defining of new Plug-in nodes and a description of the MQSeries Integrator SCADA device protocol.

The *Administration Guide* has been updated to deal with national language support, a discussion of commands to manipulate message sets, the message flow debugger, and migration issues when upgrading from previous versions of the product. Information on the New Era of Networks Rules and Formatter nodes has been moved to the *Administration Guide* from the *Installation Guide,* and is provided for each of the supported platforms. The information has also been updated to include the new functionality in the New Era of Networks Rules and Formatter support.

TRANSACTIONAL DATABASE SUPPORT

Transactional database support on Solaris is now extended to include Oracle 8.1.6 or 8.1.7. Previously, only DB2 supported transactional processing on Solaris.

Tim Dunn
Software Engineer, IBM Hursley (UK)

MQ news

Strategic Thought has released the first of its new Active Integrator gateway products, MQ/Tuxedo, providing an XML message-based interface between BEA Tuxedo and IBM WebSphere MQ, allowing applications in either environment to initiate communications via the gateway. The company claims that the product enables communication between the two systems without needing to rewrite code or configure the gateway for each message.

Active Integrator – MQ/Tuxedo is currently available on NT and Solaris, has been tested on AIX, and can be provided on other platforms running Tuxedo and MQ.

A second Active Integrator product will also be released shortly: MQ/File Adapter, which offers the ability to transport files from point-to-point using WebSphere MQ.

For further information contact:
Strategic Thought, 4 Queens Road, The Old Town Hall, Wimbledon, London SW19 8YA, UK.
Phone + 44 (0) 208 410 4000
Fax + 44 (0) 208 410 4030
Web:http://www.strategicthought.co.uk

* * *

Xephon's annual *MQ Update 2001* event runs 12-13 December at the Radisson SAS Portman Hotel in London. This two-day Conference provides a thorough analysis of recent product developments in the MQ environment and provides essential pointers on how to maximize performance within the enterprise.

For further information contact:
Xephon, 27-35 London Road, Newbury, Berks, RG14 1JL, UK.
Tel: + 44 (0) 1635 33823
Fax: + 44 (0) 1635 38345
Web:http://www.xephon.com/events

* * *

IONA has announced the release of Enterprise Integrator V3.1. A standards-based enterprise integration solution, Enterprise Integrator V3.1 offers new Web services and business-to-business integration features through interoperability with the IONA XMLBus and native IONA B2B Integrator adapters.

The company claims that these new features and advanced adapters for JMS, CORBA, and MQSeries allow companies to leverage and better utilize multiple messaging infrastructures.

For further information contact:
IONA, The IONA Building, Shelbourne Road, Ballsbridge, Dublin 4, Ireland.
Tel: + 353 1 637 2000
Fax: + 353 1 637 2888
Web: http://www.iona.com

IONA, 200 West Street 4th Floor, Waltham, MA024451, USA.
Tel: + 781 902 8000
Fax: + 781 902 8001

* * *