MQ Data Conversion and MH06

Tim Zielke

Introduction and Agenda

My Background:

- I have been in IT for 19 years with Hewitt Associates/Aon
- First 13 years mainly on the mainframe COBOL application side
- Last 6 years as a CICS/MQ systems programmer

Session Agenda:

- Main Pieces of Data Conversion (CCSID and Encoding)
- What Gets Converted and How
- Procedural (C) and Object-Oriented (Java) Data Conversion Examples
- Data Conversion Debugging With Tracing and MH06
- MH06 Data Conversion Tracing Tool - Message Parsing

Data Conversion Sources

The data conversion information for much of this session was gleaned from the following two sources:

1) “Data Conversion Under WebSphere MQ” - IBM document that covers MQ data conversion in great detail.

2) IBM MQ Manual

What is Data Conversion?

Data conversion is basically the process that MQ uses to ensure data is accurately reflected on a given system. Not all systems “talk the same language”. Just as a book needs to be translated between an English and Spanish reader, data sometimes needs to be converted between different systems (z/OS vs. Solaris) to be understood correctly.

Example:
On z/OS (EBCDIC), the message string “fox” is represented as “fox” = x’8696A7’, since f = 86, o = 9F, x = A7 in EBCDIC.

On Solaris (ASCII), the message string “fox” is represented as “fox” = x’666F78’, since f = 66, o = 6F, x = 78 in ASCII.

Main Pieces of Data Conversion

The following message descriptor fields are the main pieces that are used in data conversion.

- CCSID – Coded Character Set Id or code pages that assign glyphs (e.g. A) to decimal values (e.g. x’41’)

- Encoding – The method that a platform (specifically a CPU) uses to represent numeric data. The x86 processor is little endian, and the SPARC processor is big endian.

CCSID – Code Page

- The Coded Character Set ID (CCSID) or Code Page is a table of assigning glyphs to a number (e.g. the letter A is assigned to decimal 65 or x'41' in ASCII).

- Common CCSIDs:

a) 437 - ASCII single byte code page used mainly under OS/2, DOS, and Microsoft Windows console (OEM) windows.
b) 819 - ISO 8859-1 standard Western European ASCII single byte code page. Used commonly on Unix.
c) 037 - EBCDIC single byte code page on z/OS mainframe. “US English” EBCDIC code page.
d) 1208 - UTF-8. An encoding of Unicode which is variable in length from 1 to 4 bytes.
e) 1200 - UTF-16. An encoding of Unicode which uses a 16 bit integer (2 bytes) to represent up to 64 thousand characters (UCS-2), and a further million using 2 bytes from the ‘surrogate’ range. Surrogate pairs have 4 bytes.

NOTE:

- Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
- MQ v9 supports UTF-16. Before, MQ only supported the UCS-2 subset (2 bytes) of UTF-16.

CCSID - Example #1

437 (ASCII) - “fox” = x’666F78’,	since f = 66,	o = 6F,	x = 78
819 (ASCII) - “fox” = x’666F78’,	since f = 66,	o = 6F,	x = 78
1208 (ASCII) - “fox” = x’666F78’,	since f = 66,	o = 6F,	x = 78
1200 (ASCII) - “fox” = x’0066006F0078’,	since f = 0066,	o = 006F,	x = 0078
37 (EBCDIC) - “fox” = x’8696A7’,	since f = 86,	o = 9F,	x = A7

NOTES:

1) Pure ASCII data (7 bits or x’00-x’7F’) does not need to convert between most ASCII based code pages.
2) Code pages do not always match. You can not take the “fox” message that is encoded in ASCII on Unix, and expect a program on z/OS (EBCDIC) to be able to read it unconverted. It first needs to be converted from something like 819 (ASCII) to 37 (EBCDIC).

CCSID - Example #2

437 (ASCII)	- “niño” = x’6E69A46F’,	since n=6E,	i=69,	ñ=A4,	o=6F
819 (ASCII)	- “niño” = x’6E69F16F’,	since n=6E,	i=69,	ñ=F1,	o=6F
1208 (ASCII)	- “niño” = x’6E69C3B16F’,	since n=6E,	i=69,	ñ=C3B1,	o=6F
1200 (ASCII)	- “niño” = x’006E006900F1006F’,	since n=006E,	i=0069,	ñ=00F1,	o=006F
37 (EBCDIC)	- “niño” = x’95894996’,	since n=95,	i=89,	ñ=49,	o=96

NOTES:

1) ñ is a non-ASCII character since its byte representation does not fall between the 7 bit range of x’00-x’7F’ for ASCII based code pages. Remember 37 (EBCDIC) is not ASCII based, so ñ=49 does not count!
2) None of these code pages completely match for niño, since non-ASCII bytes are in use. The need for data conversion is more in play with non-ASCII bytes in an ASCII based code page.

Encoding

Encoding is the method that the platform (CPU) uses to represent numeric data.

- Little Endian is used by the x86 processor. It means the least significant digits appear in the lower memory locations. The number 437 = X’000001BF’ would appear as X‘BF010000’ in a memory dump. It would appear “reversed”.
- Big Endian is used by most processors (e.g. SPARC). It means the most significant digits appear in the lower memory locations. The number 437 = X’000001BF’ would appear as X’000001BF’ in a memory dump. It would appear “normal”.

NOTES:

1) For the most part, you do not have to be concerned with Encoding for data conversion. For example, MQFMT_STRING messages would almost never involve Encoding for data conversion (1200/UTF-16 would be a rare exception).
2) For messages where Encoding would matter (e.g. PCF messages), you can usually just let the defaults handle the setting of this field.

Encoding Example

On SPARC (big endian) processor:
1200 (ASCII) - “fox” = x’0066006F0078’, since f = 0066, o = 006F, x = 0078

On x86 (little endian) processor:
1200 (ASCII) - “fox” = x’66006F007800’, since f = 6600, o = 6F00, x = 7800

NOTES:

1) The encoding of the processor is coming into play. Since these “fox” characters in 1200 (UTF-16) are each represented by a 2 byte numeric number, the different encoding schemes (big endian vs. little endian) of the processors are changing the order of the two bytes that represent a character.
2) Encoding is technical and complicated. However, it is still good to be aware of its existence and implications.

Other MQ Fields Considerations

- Format (e.g. MQFMT_STRING) is part of the message descriptor and is a name that the sender of a message uses to indicate to the receiver the nature of the data in the message. MQFMT_NONE means the nature of the data is undefined. MQ will not convert a message with this format, even if it is requested by the receiving application (e.g. MQGMO_CONVERT option).

- BufferLength and message data length must not be zero for MQGET with MQGMO_CONVERT, for application message data to be converted.

```
MQGET(Hcon,        /* connection handle   */
      Hobj,        /* object handle       */
      &md,         /* message descriptor  */
      &gmo,        /* get message options */
      buflen,      /* buffer length       */  <-----
      buffer,      /* message buffer      */
      &messlen,    /* message length      */  <-----
      &CompCode,   /* completion code     */
      &Reason);    /* reason code         */
```

What Gets Converted, and How

- (Automatically by MQ) - MQ will negotiate the proper CCSID and Encoding that will be used for channel communication. This may cause implicit data conversion on things like channel control blocks (e.g. TSH – Transmission Segment Header) and the message descriptor portion (e.g. Expiry, Priority, etc.) of messages.

- (Determined by Admin/Programmer) – The user portion of the of the message (i.e. message data).

1) The message data conversion can happen at the SENDER channel end with the CONVERT(YES) channel attribute. The message data is then converted to the CCSID and Encoding of the target queue manager. This data conversion approach is not recommended.
2) The message data conversion can also happen on the MQGET with the MQGMO_CONVERT option. This data conversion approach of "Receiver makes good" or converting on the MQGET end by the application is recommended.

C PUT of String Message

For the C language, the MQMD.CodedCharSetId and MQMD.Encoding need to accurately reflect the MQFMT_STRING message data in an MQPUT. The programmer is responsible to make sure these fields are accurately set before the MQPUT.

```
# explicitly set CCSID and Encoding before PUT
# if local connection, MQCCSI_Q_MGR resolves to queue manager CCSID
md.CodedCharSetId = MQCCSI_Q_MGR;
md.Encoding       = MQENC_NATIVE;

MQPUT
 (Hcon,       /* conn handle */
  Hobj,       /* obj handle  */
  &md,        /* msg desc    */
  &pmo,       /* put msg opt */
  messlen,    /* msg length  */
  buffer,     /* msg buffer  */ <- data must be in qmgr CCSID!
  &CompCode,  /* comp code   */
  &Reason);   /* reason code */
```

Java PUT of String Message

For using IBM MQ Classes for Java, a Java String is encoded in UTF-16. Since the String has an inherit CCSID, you can ask Java to convert the string to another CCSID on the Put.

```
String str1 = "blah";  <- Contains string data in UTF-16

int openOptions = MQConstants.MQOO_OUTPUT;
MQPutMessageOptions pmo = new MQPutMessageOptions();
MQMessage msg = new MQMessage();
msg.format = MQConstants.MQFMT_STRING;

msg.characterSet = 37; <- 37 is EBCDIC

msg.writeString(str1);  <- Converts str1 to EBCDIC
queue.put(msg, pmo);  <- Message is now on the queue as EBCDIC!
```

C GET of String Message

For the C language, the MQFMT_STRING message will be converted on a GET if the MQGMO_CONVERT option is specified and the input CodedCharSetId or Encoding on the GET does not match the CodedCharSetId or Encoding of the message.

```
# explicitly set CCSID and Encoding before GET
# if local connection, MQCCSI_Q_MGR resolves to queue manager CCSID
gmo.Options        = MQGMO_CONVERT;
md.CodedCharSetId = MQCCSI_Q_MGR;
md.Encoding        = MQENC_NATIVE;

MQGET(Hcon,       /* conn handle */
      Hobj,       /* obj handle  */
      &md,        /* msg desc    */
      &gmo,       /* get msg opt */
      buflen,     /* buffer len  */
      buffer,     /* msg buffer  */
      &messlen,   /* msg length  */
      &CompCode,  /* comp code   */
      &Reason);   /* reason code */
```

Java GET of String Message

For using IBM MQ Classes for Java, you do not need to set the MQGMO_CONVERT option, as Java can convert the message for you. For the below example, we will assume our message is in EBCDIC (CCSID 37) on the queue.

```
int openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF;
MQQueue queue = qMgr.accessQueue(qName, openOptions);
MQMessage rcvMessage = new MQMessage();

//No MQGMO_CONVERT specified
MQGetMessageOptions gmo = new MQGetMessageOptions();

//Unconverted GET and then Java converts from EBCDIC to UTF-16
//when the Java String is built
queue.get(rcvMessage, gmo);
int length = rcvMessage.getDataLength();
String msgText = rcvMessage.readStringOfByteLength(length);
```

Other Java Considerations

- Using byte arrays instead of Strings to store message data will result in Java MQ programming working more like the C procedural approach. This has the potential benefit of improving performance due to less data conversion, but also can add programming complexity.

- For an MQ Java program that wants to read data from a file (UTF-8) into a String (UTF-16) and put that data to a message, note that there is a potential data conversion from reading the data from the file.

```
File f = new File(“myfile.txt”);
FileInputStream fis = new FileInputStream(f);

Case #1 – Java assumes file is in the CCSID of JVM default charset
InputStreamReader isr = new InputStreamReader(fis);

Case #2 – Java is explicitly told file in the CCSID of UTF-8
Charset charset = Charset.forName(“UTF-8”);
InputStreamReader isr = new InputStreamReader(fis, charset);
```

Client and Other Considerations

- With local (bindings) connections, MQCCSI_Q_MGR refers to the CCSID of the queue manager on a PUT/GET. However, in a client application, MQCCSI_Q_MGR refers to the CCSID of the application locale, not the CCSID of the remote queue manager.

- Data conversion for the client normally happens on the queue manager side. However, certain configurations (e.g. AMS) can cause data conversion to happen on the client side.

- Data conversion happens outside of the queue manager. For a bindings (local) connection, the data conversion happens in the application process. For a client connection, it usually happens in the amqrmppa (channel pooling process) process. As a note, the Application Activity Trace can not report data conversion on an MQGET, since the data conversion is happening outside of the queue manager.

Data Conversion Debugging Tools

- strmqtrc distributed tracing will capture the message after it has been converted. It will show you both the input and output CodedCharSetId and Encoding on an MQPUT or MQGET, and other key fields like the Format of the message. It can also trace the message contents providing both a hex and text dump of the message. For z/OS, the API trace can provide similar information.

- MH06 Trace Tools supportpac provides a tool called mqtrcfrmt that can make a strmqtrc or z/OS API trace much more readable by converting the hex dumps of MQ data structures (e.g. MQMD, MQGMO, etc.) to human readable fields. Very helpful for reading traces! Linux x86, Solaris SPARC, and Windows executables are provided. NOTE: mqtrcfrmt has only been tested with tracing on these platforms.

- Java tracing can also be used for Java applications. For using IBM MQ Classes for Java, remember that the MQMessage characterSet variable plays a role in the data conversion.

- mqcpcnvt is a free program available on the Capitalware web site. It allows you to see how message data at the byte level converts between different code pages. For example, how would the ® symbol (x’AE’ in 819) convert from 819 to 1208 (x’C2AE’ in 1208).

Example using strmqtrc and MH06

Example: A developer reports that their local C application named mqget1 is having issues getting EBCDIC (37) messages from a queue and converting them to ASCII (819).   When the application receives the messages, they are still in EBCDIC. The following trace approach can be used to further investigate the issue.

1) Use strmqtrc to collect an API trace of this mqprog1 application.

```
> strmqtrc -m qmgr1 -t api -d all -p mqget1
```

2) On Unix platforms, use dspmqtrc to format the trace.

```
> dspmqtrc AMQ1234.0.TRC > AMQ1234.0.FMT
```

3) Use mqtrcfrmt in MH06 to provide more formatting of trace

```
> mqtrcfrmt AMQ1234.0.FMT AMQ1234.0.FMT2
```

strmqtrc GMO without MH06

```
08:51:47.841950  22861.1  MQGET >>
.
.
08:51:47.841995  22861.1  Getmsgopts:
08:51:47.841999  22861.1    0x0000:  474d4f20 00000001 00002002 00000000  |GMO ......`.....|
08:51:47.841999  22861.1    0x0010:  00000000 00000000 54435a2e 54455354  |........TCZ.TEST|
08:51:47.841999  22861.1    0x0020:  31202020 20202020 20202020 20202020  |1               |
08:51:47.841999  22861.1    0x0030:  20202020 20202020 20202020 20202020  |                |
08:51:47.841999  22861.1    0x0040:  20202020 20202020                    |                |
```

strmqtrc GMO with MH06

```
08:51:47.841950  22861.1  MQGET >>
.
.
08:51:47.841995  22861.1  Getmsgopts:
08:51:47.841999  22861.1    0x0000:  474d4f20 00000001 00002002 00000000  |GMO ......`.....|
08:51:47.841999  22861.1    0x0010:  00000000 00000000 54435a2e 54455354  |........TCZ.TEST|
08:51:47.841999  22861.1    0x0020:  31202020 20202020 20202020 20202020  |1               |
08:51:47.841999  22861.1    0x0030:  20202020 20202020 20202020 20202020  |                |
08:51:47.841999  22861.1    0x0040:  20202020 20202020                    |                |
                 22861.1       Getmsgopts expanded (all fields):
                 22861.1       StrucId (CHAR4)              : 'GMO '
                 22861.1                                      x'474d4f20'
                 22861.1       Version (MQLONG)             : 1
                 22861.1                                      x'00000001'
                 22861.1       MQGMO.Options= (MQLONG)      : 8194
                 22861.1                                      x'00002002'
                 22861.1          Options=MQGMO_SYNCPOINT
                 22861.1          Options=MQGMO_FAIL_IF_QUIESCING
                 22861.1       WaitInterval (MQLONG)        : 0
                 22861.1                                      x'00000000'
                 22861.1       Signal1 (MQLONG)             : 0
                 22861.1                                      x'00000000'
                 22861.1       Signal2 (MQLONG)             : 0
                 22861.1                                      x'00000000'
                 22861.1       ResolvedQName (MQCHAR48)     : 'TCZ.TEST1'
                 22861.1                                      x'54435a2e5445535431'
```

strmqtrc and MH06 example

The following pieces are found in the trace for the MQGET call. We can see both the inputs and outputs coming from this MQGET. From this trace data, we can see that the local C application is missing something in their MQGET call. What is missing?

```
MQGET >>    (Inputs being passed into the MQGET)
Encoding (MQLONG)                : 273 (MQENC_INTEGER_NORMAL)
CodedCharSetId (MQLONG)          : 819
MQGMO.Options   (MQLONG)         : MQGMO_SYNCPOINT,MQGMO_FAIL_IF_QUIESCING

MQGET <<    (Outputs being passed back from the MQGET)
Encoding (MQLONG)                : 273 (MQENC_INTEGER_NORMAL)
CodedCharSetId (MQLONG)          : 37
Format (CHAR8)                   : MQSTR (MQFMT_STRING)
Compcode                         : 0
Reason                           : 0
```

NOTES:

- There is a learning curve with using tracing. However, if you get comfortable with tracing and the MH06 supportpac, you can have a powerful debugging tool that allows you to “look under the hood” of your applications and quickly get to the bottom of application issues.

MH06 Message Parsing Tool

- Message parsing will analyze a message in a strmqtrc or activity trace as if it was a certain CCSID, and then provide a breakdown of that message based on that assumption. The CCSID values supported for message parsing are 1208 (UTF-8) and 1200 (UTF-16).

- Message parsing could be useful for answering questions like:

1. “Does this message contain surrogate pairs and where?“
2. “Has the message been incorrectly marked as 819 when it has a 1208 make up?” or “Has the message been incorrectly marked as 1208?“
3. “Does the message contain non-ASCII bytes and where?“.

- Before we cover message parsing in more detail, it is helpful to have an understanding of ASCII, and extended ASCII code pages like 819 and 1208.

ASCII

- ASCII is a 7 bit (x’00’ - x’7F’) character encoding.
- ASCII encodes the English alphabet in upper case (x’41 - x’5A’) and lower case (x’61’ - x’7A’).
- ASCII also encodes numeric digits 0-9 (x’30 - x’39’) and various other punctuation and control characters.
- Many code pages (ex. 437, 819, 1208) are ASCII based, and have the alphabetic, numeric, and punctuation part of ASCII (x’20’ - x’7E’) in common. In other words, if your MQ message is string and ASCII based, it’s data will not need to convert between most ASCII based code pages.

819 - Single Byte ASCII Based Code Page

- CCSID 819 or ISO/IEC 8859-1 is a single byte ASCII based code page that extends the 7 bit ASCII by adding in an eighth bit for more character encodings.

- This allows for character encodings from x’00’ - x’FF’. Remember, ASCII is only x’00’ - x’7F’. This one bit extension allows for up to 128 more character mappings in the x’80’ - x’FF’ range.

- 819 or ISO/IEC 8859-1 is a Latin alphabet code page, and is generally intended for Western European languages. For example, the character ñ would be x’F1’ in 819.

1208 - Multi-Byte ASCII Based Code Page

- CCSID 1208 or UTF-8 is a multi-byte ASCII based code page. It encodes all the Unicode code points in one to four bytes, and is the most commonly used code page for the Internet.

- Single byte UTF-8 encodings will only be ASCII values (i.e. x’00’ - x’7F’). Also, ASCII encodings will never appear in a multi-byte encoding.

- Multi-byte UTF-8 encodings have a distinct make up:

1) Two byte encodings must have a first byte of 110xxxxx (x’C2’ - x’DF’) and a second byte of 10xxxxxx (x’80’ - x’BF’).
2) Three byte encodings must have a first byte of 1110xxxx (x’E0’ - x’EF’) and a second and third byte of 10xxxxxx (x’80’ - x’BF’).
3) Four byte encodings must have a first byte of 11110xxx (x’Fx’ range) and a second, third, and fourth byte of 10xxxxxx (x’80’ - x’BF’).

819 and 1208 example

A string message with the data “niño”, would be encoded as follows:

819 (single byte extended ASCII code page):
n=x‘6E’ i=x’69’ ñ=x’F1’ o=‘6F’

1208 (multi-byte ASCII UTF-8 code page):
n=x‘6E’ i=x’69’ ñ=x’C3B1’ o=‘6F’

Observations:

1) If an MQ program took the first message x’6E69F16F’ and incorrectly labeled the CCSID as 1208, we could apply the 1208 encoding rules to the bytes of the message and find that the third byte is invalid.
2) If an MQ program took the second message x’6E69C3B16F’ and incorrectly labeled the CCSID as 819, we could apply the 1208 encoding rules to the bytes of the message and find that it is an exact match to 1208.

Using Message Parsing in MH06

- mqtrcfrmt program has the message parsing functionality. It can be used against a message that is traced in either strmqtrc or the amqsact activity trace program.

```
# run trace with -d all option to capture message data
> strmqtrc -m QM1 -t api -d all -p mypgmname

# For some platforms, use dspmqtrc to format the trace
> dspmqtrc AMQ12345.0.TRC > AMQ12345.0.FMT

# mqtrcfrmt program with -m message parsing option to byte analyze message as 1208
> mqtrcfrmt.linux AMQ12345.0.FMT AMQ12345.0.FMT2 -m 1208

Inside AMQ12345.0.FMT2:
08:06:08.803334    23401.1     Buffer:
08:06:08.803338    23401.1       0x0000:  6e69c3b1 6f                           |ni..o            |
msg-parser  UTF-8 Totals: Line:366 Pid:23401.1 Format:MQSTR    CCSID:1208 API:MQGET <<
Byte:5 ASCII:3 MB2:1 MB3:0 MB4:0 Inv:0
msg-parser Byte Analysis: Line:366 2-MB2,
```

Message Parsing - Real Life Use Case

Background:
We had an MQ Server that was running on Solaris SPARC and was being ported to Linux x86. The old queue manager on Solaris had a default CCSID(819), and the new queue manager on Linux had a default CCSID(1208). There was a Windows application that would PUT string messages with a CCSID(819) and the data was positional in nature. There was a getting C application with local bindings that ran on the Solaris server that would GET with a CONVERT the messages. The input CCSID on the GET with CONVERT was MQCCSI_Q_MGR.

Question:
What CCSID does the getting C application convert the string message to on Solaris? When ported to Linux? Why?

Message Parsing:
I used message parsing to analyze the putting Windows application message data to see if there was a potential data conversion issue here with this MQ port from Solaris to Linux. strmqtrc tracing was used to capture the application message data, and then the message data was message parsed as 1208.

Message Parsing - Real Life Use Case

strmqtrc trace data (formatted to fit slide) with 1208 message parsing:

```
10:54:31.130641  Buffer:
10:54:31.130645    0x0000:  32303135 2f31322f 31332030 303a3030  |2015/12/13 00:00|
10:54:31.130645    0x0010:  3a3030fd 32303135 2f31322f 31342030  |:00.2015/12/14 0|
10:54:31.130645    0x0020:  323a3532 3a3337fd 31313738 34463242  |2:52:37.11784F2B|
10:54:31.130645    0x0030:  37384245 39334431 38363235 37454536  |78BE93D186257EE6|
10:54:31.130645    0x0040:  30303639 33433442 fd323031 352f3130  |00693C4B.2015/10|
10:54:31.130645    0x0050:  2f323220 31343a30 393a3237 fd323031  |/22 14:09:27.201|
10:54:31.130645    0x0060:  352f3132 2f313420 30323a31 383a3335  |5/12/14 02:18:35|
10:54:31.130649  Compcode      : Output Parm
msg-parser  UTF-8 Totals: Line:131 Pid:15983.1 Format:MQSTR    CCSID:819 API:MQPUT >>
Byte:112 ASCII:108 MB2:0 MB3:0 MB4:0 Inv:4
msg-parser Byte Analysis: Line:131 13-INV,27-INV,48-INV,5c-INV,
```

Analysis:

- Byte x'FD' is being used as a delimiter.
- Since x'FD' is not ASCII, it will convert to a multi-byte character (x'C3BD') in 1208.
- When this C getting application is ported from Solaris (819) to Linux (1208), this delimiter will grow from 1 to 2 bytes, causing the message data to shift from a positional standpoint.
- Problems!

Message Parsing - Real Life Use Case

Solution:
I made this application data conversion issue known to the developer, and they changed their application to do the GET with an input CCSID(819). This guarantees that the application gets the data in a single byte code page that it was expecting.

Application:

- Message parsing is helpful for analyzing MQ applications that are being ported between different servers.
- Message parsing is helpful for reviewing new MQ applications.
- Message parsing is helpful for analyzing “poison” messages.
- Reminder: Message parsing can be done with a 1200 (UTF-16) or 1208 (UTF-8) assumption.

MH06 - msg2File

If you insert a special tag “msg2File-” above a “Buffer:” line in a strmqtrc trace or a “Message Data:” line in an activity trace, the mqtrcfrmt program will write the bytes of the message to a file (max length of file name is 20) whose name follows the “msg2File-“ tag.

For example, if you add this msg2File line before a message Buffer in strmqtrc:

```
14:49:27.664265 29003.1 CONN:1400006 msg2File-file1
14:49:27.664265 29003.1 CONN:1400006 Buffer:
14:49:27.664269 29003.1 CONN:1400006 0x0000: 0066006F 0078D801 DC37 |................|
```

then a file called file1 will be written out in your current directory that contains the bytes of the message in the Buffer.

As a convenience, a Java (1.5 compiled) MQFile2Msg.class executable is provided to be able to take a file like the one that msg2File will produce and PUT it back to a queue.

This functionality allows you to capture and reuse messages without having to stop running MQ applications.

Questions & Answers