MQ Data Conversion

Tim Zielke

Introduction and Agenda

My Background:

- I have been in IT for 21 years with Hewitt Associates/Aon/Alight
- First 13 years mainly on the mainframe COBOL application side
- Last 8 years as a CICS/MQ systems programmer supporting both mainframe and distributed MQ

Session Agenda:

- Main Pieces of Data Conversion (CCSID, Encoding, and Format)
- Procedural (C) and Object-Oriented (Java) Data Conversion
- Data Conversion Debugging With Tracing and MH06 Trace Tools Supportpac
- MH06 Data Conversion Tracing Tool - Message Parsing

Data Conversion Sources

The data conversion information for much of this session was referenced 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” = 0x8696A7, since f = 0x86, o = 0x9F, x = 0xA7 in EBCDIC.

On Solaris (ASCII), the message string “fox” is represented as “fox” = 0x666F78, since f = 0x66, o = 0x6F, x = 0x78 in ASCII.

Main Pieces of Data Conversion

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

- Encoding – Specifies the numeric encoding of numeric data in the message. The numeric encoding defines the representation used for binary integers, packed-decimal integers, and floating-point numbers.

- CCSID – Coded Character Set Id. The IBM CSSID is a complex technical term that is not simple to define. This is an oversimplified explanation, but think of the CCSID as an unique number (e.g. 819) that defines a table of character symbols (e.g. a, b, c, etc.) to byte encodings (e.g. 0x61, 0x62, 0x63, etc.). For a detailed definition of the IBM CCSID, consult the IBM Character Data Representation Architecture (CDRA).

- Format - The name that the sender of a message uses to indicate to the receiver the nature of the data in the message (e.g. MQFMT_STRING).

Encoding

Encoding specifies the numeric encoding of numeric data in the message. The numeric encoding defines the representation used for binary integers, packed-decimal integers, and floating-point numbers.

Here are some examples of common md.Encoding that you might see in an MQ message:

Example from MQPUT on Linux x86

```
md.Encoding = 546
    MQENC_INTEGER_REVERSED
    MQENC_DECIMAL_REVERSED
    MQENC_FLOAT_IEEE_REVERSED
```

Example from MQPUT on Solaris SPARC

```
md.Encoding = 273
    MQENC_INTEGER_NORMAL
    MQENC_DECIMAL_NORMAL
    MQENC_FLOAT_IEEE_NORMAL
```

Encoding – DLQ Example

We have a message on the dead letter queue on both a Linux x86 (INTEGER_REVERSED) and Solaris SPARC (INTEGER_NORMAL) IBM MQ queue manager. The message was put to the DLQs because the TCZ.TEST1 queue was full in both cases.

MQRC_Q_FULL = 2053 = 0x00000805.

MQDLH Reason field is a 4 byte integer field and is at offset 0x08 in bold red below. We use amqsbcg to browse the messages.

Linux x86 (INTEGER_REVERSED):

```
Encoding : 546

00000000:  444C 4820 0100 0000 0508 0000 5443 5A2E          'DLH ........TCZ.'
00000010:  5445 5354 3120 2020 2020 2020 2020 2020          'TEST1           `
```

Solaris SPARC (INTEGER_NORMAL):

```
Encoding : 273

00000000:  444C 4820 0000 0001 0000 0805 5443 5A2E          'DLH ........TCZ.'
00000010:  5445 5354 3120 2020 2020 2020 2020 2020          'TEST1           `
```

Encoding

Reversed encoding (or Little Endian) is used by the x86 processor. It means the least significant bytes appear in the lower memory locations. The bytes appear “reversed”.

```
Ex. 0x00000805 appeared as 0x05080000 in the amqsbcg output
```

Normal encoding (or Big Endian) is used by most processors (e.g. SPARC). It means the most significant bytes appear in the lower memory locations. The bytes appear “normal”.

```
Ex. 0x00000805 appeared as 0x00000805 in the amqsbcg output
```

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).

For messages where Encoding would matter (e.g. PCF messages), you can usually just let the defaults (e.g. md.Encoding = MQENC_NATIVE) handle the setting of this field.

CCSID – Coded Character Set ID

- From a simplistic/practical stand point, think of the Coded Character Set ID (CCSID) as a unique number that represents an assignment of character symbols to byte encodings (e.g. CCSID 819 assigns the letter A to x'41'). Basically, a code page of symbols to bytes.

- 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 2 bytes (UCS-2) or 4 bytes (surrogate pairs).

- 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 fully supports UTF-16. Before v9, MQ only supported the UCS-2 subset (2 bytes) of UTF-16.

CCSID - Example #1

```
437 (Windows)           - fox = 0x666F78         f=66   o=6F   x=78
819 (ISO-8859-1)        - fox = 0x666F78         f=66   o=6F   x=78
1208 (UTF-8)            - fox = 0x666F78         f=66   o=6F   x=78
1200 (UTF-16/Normal)    - fox = 0x0066006F0078 f=0066 o=006F x=0078
1200 (UTF-16/Reversed)- fox = 0x66006F007800 f=6600 o=6F00 x=7800
37 (mainframe/EBCDIC) - fox = 0x8696A7         f=86   o=9F   x=A7
```

NOTES:

1) Pure ASCII data (7 bits or 0x00 - 0x7F) does not need to convert between most ASCII based code pages (e.g. 437, 819, 1208).
2) CCSIDs do not always match and some need data conversion. 437 -> 37 and 819 -> 1200 are examples above.
3) Endianness (Normal vs. Reversed) in 1200 requires data conversion even with the same CCSID. md.Encoding (273 for Normal and 546 for Reversed) is coming into play here with the data conversion.

CCSID - Example #2

```
437 (Windows)           - niño=0x6E69A46F          n=6E   i=69   ñ=A4   o=6F
819 (ISO-8859-1)        - niño=0x6E69F16F          n=6E   i=69   ñ=F1   o=6F
1208 (UTF-8)            - niño=0x6E69C3B16F        n=6E   i=69   ñ=C3B1 o=6F
1200 (UTF-16/Normal)    - niño=0x006E006900F1006F n=006E i=0069 ñ=00F1 o=006F
1200 (UTF-16/Reversed)- niño=0x6E006900F1006F00 n=6E00 i=6900 ñ=F100 o=6F00
37 (mainframe/EBCDIC) - niño=0x95894996          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 0x00 - 0x7F 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. There is a need for data conversion between all of these code pages.
3) CCSIDs 437, 819, 37 are single byte code pages, so only one byte is needed to encode the ñ character.
4) UTF-8 (CCSID 1208) requires two bytes to encode the non-ASCII ñ character. Non-ASCII characters in UTF-8 are uniquely encoded in multi-byte (2, 3, or 4 byte) encodings. UTF-8 has no endianness property like UTF-16, and this is a reason why UTF-8 is more popular in usage than UTF-16.

md.Format

md.Format 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_STR (resolves to MQSTR) means the data is in a character or string format. One of the more common md.Formats.

- MQFMT_NONE (resolves to spaces) 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).

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 procedural applications is recommended.

Other Data Conversion Notes

- With local (bindings) connections, the md.CodedCharSetId value of 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.

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, Encoding, and Format before PUT.
memcopy(md.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);
md.CodedCharSetId = 1208;
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 1208 before PUT!
 &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 encoding (UTF-16), you can ask the IBM MQ Java code to convert the string to another CCSID on the Put.

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

NOTE: If you do NOT set the msg.characterSet, it will use a default (e.g. 819/ISO-8859-1). This default value will then be used to convert msg.writeString(str1) from UTF-16 to this default value (e.g. 819/ISO-8859-1). If you are not explicitly setting msg.characterSet, it is being set for you!

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 MQ 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 MQ Java converts from EBCDIC to UTF-16
queue.get(rcvMessage, gmo);
int length = rcvMessage.getDataLength();
String msgText = rcvMessage.readStringOfByteLength(length);
```

NOTE: My recommendation is to NOT set the MQGMO_CONVERT option for MQ Java, as it can cause double conversion (e.g. 37/EBCDIC to 819/ISO-8859-1 on GET with convert and then from 819/ISO-8859-1 to 1200/UTF-16 in MQ Java readString methods) which is less straightforward and efficient.

Example of a Potential Java GET Issue

We had this issue with one of our MQ Java programs. The code worked fine when only 819/ISO-8859-1 messages were being used. However, when we started using 1208/UTF-8 messages that included Chinese characters, the Chinese characters in the msgText were garbled.

```
mQueue.get(retrievedMessage, gmo);
int len = retrievedMessage.getDataLength();
byte[] bmsgText = new byte[len];
retrievedMessage.readFully(bmsgText);
msgText = new String(bmsgText);
```

- Byte arrays have no encoding. They are just raw bytes.
- How does Java know how to convert a byte array (bmsgText) with no encoding to a Java String msgText (UTF-16)?

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 encoding of JVM default charset
InputStreamReader isr = new InputStreamReader(fis);

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

What about JMS?

- As a reminder, we have been talking about the IBM MQ Classes for Java (base Java). The IBM MQ Classes for JMS (Java Message Service) is a different API than base Java, but the general data conversion principles we have discussed for base Java still apply for JMS.

- It is beyond the scope of this session to go into a detailed explanation of JMS data conversion. The section “JMS message conversion approaches” in the MQ manual is a good place to start to better understand the details of JMS data conversion.

- If you work with JMS, just remember the subtle but important point that a Java String has an encoding of UTF-16. If you are building a Java String from a file, taking a Java String and writing it to an MQ message, etc., there is probably underlying data conversion happening in this process.

Java/JMS String Conversion Change at v8

- From IBM MQ Version 8.0, some of the default behavior regarding character string conversion with Java and JMS clients has changed.
- Before Version 8.0, any untranslatable or malformed data found would result in those characters being replaced by the unmappable character replacement, usually a ? character.
- From Version 8.0, the default behavior changed to report unmappable characters like this by throwing an exception. It is still possible to have the characters replaced, by setting a property to modify the behavior.
- Bottom line, MQ Java/JMS applications may behave differently when you upgrade the jars they are using from v7 (or lower) to MQ v8 (or higher).

Example - String Conversion Change at v8

An MQ Java application will read in a file with the following text and write it as a string to a queue. The file text that is read in will be stored in a Java String (UTF-16). The CCSID of the message on the write string will be set to 819 (ISO-8859-1). Therefore, there will be data conversion on the write string. Note also there is a non-mappable byte for 819 of 0x94 that is at the end of the file text.

```
00000000:  7175 6963 6B20 6272 6F77 6E20 666F 7894          'quick brown fox.'
```

Using MQ 7.5 Java jars, the write string works and you get the following message on the queue. The unmappable byte 0x94 has been changed to a “?” (? is 0x3F in ASCII).

```
00000000:  7175 6963 6B20 6272 6F77 6E20 666F 783F          'quick brown fox?'
```

When you run the same MQ Java program using the MQ v8 jars, the following error is encountered when trying to write the string:

```
An IOException occured whilst writing to the message buffer:
java.nio.charset.UnmappableCharacterException: Input length = 1
java.nio.charset.UnmappableCharacterException: Input length = 1
        at java.nio.charset.CoderResult.throwException(CoderResult.java:278)
        at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:798)
        at com.ibm.mq.jmqi.system.JmqiCodepage.stringToBytes(JmqiCodepage.java:759)
        at com.ibm.mq.MQMessage.writeString(MQMessage.java:2750)
        at MQSamplePut.main(MQSamplePut.java:291)
```

Java/JMS String Conversion Change at v8

- If you want the MQ Java application in the previous slide to work at v8 like it did at v7, you can pass in the following Java system property at the Java command line:

```
java -Dcom.ibm.mq.cfg.jmqi.UnmappableCharacterAction=REPLACE MQSamplePut
```

- However, note that the ability to use this Java system property to revert back to the v7 data conversion approach did not originally work when MQ v8 was released, and was corrected to work at 8.0.0.6.

Data Conversion Debugging Tips

- Get down to the byte level! Have the programmer add code if necessary to print out the data in bytes for debugging purposes.
- 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 trace much more readable by converting the hex dumps of MQ data structures (e.g. MQMD, MQGMO, etc.) to human readable fields. Mqtrcfrmt also has a message search feature to find messages in trace that have a certain piece of text.
- New for MH06 v1.0.8, a Java mqtrcfrmt.jar is now provided (built at Java 1.8). The previous Linux x86, Solaris SPARC, and Windows mqtrcfrmt C executables are still provided, but are now deprecated.

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/ISO-8859-1). When the application receives the messages, they are still in EBCDIC. You ask the developer are they setting the MQGMO_CONVERT option on the GET, and they say yes. The following trace approach can be used to further investigate the issue.

1) Use strmqtrc to collect an API trace of this mqget1 application on Solaris SPARC.

```
> strmqtrc -m qmgr1 -t api -d all -p mqget1
> mqget1 TCZ.TEST1
> endmqtrc -a
> dspmqtrc AMQ1234.0.TRC > AMQ1234.0.FMT
```

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

```
> java -jar mqtrcfrmt.jar –i AMQ1234.0.FMT -t unix
```

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 also 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 the MQGMO_CONVERT option.

```
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 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 819 (ISO-8859-1), 1200 (UTF-16), and 1208 (UTF-8).

- Message parsing could be useful for answering questions like:

1. “Does this 1200 message contain surrogate pairs and where in the message?“
2. “Does this 1208 message follow the rules for UTF-8? If not, where in the message does it have malformed data?”
3. “Does the 819 message contain non-ASCII bytes and where in the message?“.

1208 – UTF-8

- 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. 0x00 - 0x7F). 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 (0xC2 - 0xDF) and a second byte of 10xxxxxx (0x80 - 0xBF).
2) Three byte encodings must have a first byte of 1110xxxx (0xE0 - 0xEF) and a second and third byte of 10xxxxxx (0x80 - 0xBF).
3) Four byte encodings must have a first byte of 11110xxx (0xF0 - 0xF4) and a second, third, and fourth byte of 10xxxxxx (0x80 - 0xBF).

niño = 0x6E69C3B16F, since n=6E, i=69, ñ=C3B1, o=6F

CCSID Mislabeled Example

A string message with the data “niño”, would be encoded as follows in 819/ISO-8859-1:

```
n=0x6E i=0x69 ñ=0xF1 o=0x6F
```

What if an MQ program PUT this message of 0x6E69F16F on a queue but incorrectly labeled the CCSID of the message as 1208? Can we “message parse” the bytes as if they were 1208 to see if the message is valid?

```
n=0x6E <- ASCII byte. Good

i=0x69 <- ASCII byte. Good

ñ=0xF1 <- Leading byte for 3 byte encoding. Bad, only one byte left
in message.

o=0x6F <- ASCII byte. Bad, since should be a trailing byte for 3 byte
encoding.
```

Using Message Parsing in MH06

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

```
# run trace with -d all option to capture message data
> strmqtrc -m QM1 -t api -d all -p mypgmname
> mypgmname
> endmqtrc -a
> dspmqtrc AMQ12345.0.TRC > AMQ12345.0.FMT

# mqtrcfrmt program with -m message parsing option to byte analyze message as 1208
> java -jar mqtrcfrmt.jar -i AMQ12345.0.FMT -t unix -m 1208

# Inside AMQ12345.0.FMT2:
09:56:19.435415    Buffer:
09:56:19.435418      0x0000:  666f78c2 81e18586 f9b0b1b2 666f78c2  |fox.........fox.|
09:56:19.435418      0x0010:  81e18586 f9b0b1b2 666f78c2 81e18586  |........fox.....|
09:56:19.435418      0x0020:  f9b0b1b2                            |....            |

msg-parser UTF-8 Totals: Line:166 Pid:12345.1 Format:MQSTR    CCSID:1208 API:MQGET <<
Byte:36 ASCII:9 MB2:3 MB3:3 MB4:0 Inv:12

msg-parser Byte Analysis: Line:166 3-MB2,5-MB3,8-INV,9-INV,10-INV,11-INV,15-MB2,17-MB3,20-
INV,21-INV,22-INV,23-INV,27-MB2,29-MB3,32-INV,33-INV,34-INV,35-INV,
```

Questions & Answers

Addendum Slides

The following are addendum slides that provide more specifics about code pages and Unicode.

ASCII

- ASCII is a 7 bit (0x00 - 0x7F) character encoding.
- ASCII encodes the English alphabet in upper case (0x41 - 0x5A) and lower case (0x61 - 0x7A).
- ASCII also encodes numeric digits 0-9 (0x30 - 0x39) 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 (0x20 - 0x7E) 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 – ISO/IEC 8859-1

- 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 0x00 - 0xFF. Remember, ASCII is only 0x00 - 0x7F. This one bit extension allows for up to 128 more character mappings in the 0x80 - 0xFF range.
- Bytes 0x00 - 0x1F and 0x7F - 0x9F are actually undefined in ISO-8859-1.
- 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 0xF1 in 819.

niño = 0x6E69F16F, since n=6E, i=69, ñ=F1, o=6F

Unicode

- Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems

- Unicode will map characters to code points. For example, ñ maps to Unicode code point U+00F1. There are over one million code points in Unicode.

- UTF (e.g. UTF-8, UTF-16) stands for Unicode Transformation Format. UTF is a mapping of Unicode code points to a unique sequence of bytes.

  Example:

  ñ = Unicode code point U+00F1

  ñ = x'00F1' in UTF-16

  ñ = x'C3B1' in UTF-8

1200 – UTF-16

- UTF-16 encodes all the Unicode code points in either two byte or four byte encodings.
- Unicode code points U+0000 to U+D7FF and U+E000 to U+FFFF are encoded in two bytes that are numerically equal to the code point. Ex. n=006E
- Unicode code points U+10000 to U+10FFFF are encoded with a pair of two bytes (four bytes total) called surrogate pairs. Unicode code points U+D800 to U+DFFF are reserved to support the UTF-16 surrogate pair encoding design. 0xD800 - 0xDBFF encodes a high surrogate. 0xDC00 - 0xDFFF encodes a low surrogate. Ex. 𠀀 encoded in big endian UTF-16 as 0xD852DF62
- UTF-16 encoded data can also start with a BOM (Byte Order Mark) to denote the endianness of the data. A BOM of 0xFEFF = big endian, and a BOM of 0xFFFE = little endian. IBM MQ recognizes a BOM for CCSID=1200, and a BOM takes precedence over the md.Encoding.

niño (little endian) = 0x6E006900F1006F00, since n=6E00, i=6900, ñ=F100, o=6F00
niño (big endian) = 0x006E006900F1006F, since n=006E, i=0069, ñ=00F1, o=006F
niño (with BOM) = 0xFEFF006E006900F1006F, since n=006E, i=0069, ñ=00F1, o=006F

1208 – UTF-8

- 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. 0x00 - 0x7F). 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 (0xC2 - 0xDF) and a second byte of 10xxxxxx (0x80 - 0xBF).
2) Three byte encodings must have a first byte of 1110xxxx (0xE0 - 0xEF) and a second and third byte of 10xxxxxx (0x80 - 0xBF).
3) Four byte encodings must have a first byte of 11110xxx (0xF0 - 0xF4) and a second, third, and fourth byte of 10xxxxxx (0x80 - 0xBF).

niño = 0x6E69C3B16F, since n=6E, i=69, ñ=C3B1, o=6F