How to do a manual byte swapping from the output of IBM MQ amqsbcg in Windows and Linux x86 to find out the DLQ reason code

https://www.ibm.com/support/pages/node/7151034

Date last updated: 10-May-2024

Angel Rivera
IBM MQ Support
https://www.ibm.com/products/mq/support
Find all the support you need for IBM MQ

+++ Objective +++

I know, the title is very long but the title tries to provide the most common reason why IBM MQ Administrators may want to do a manual byte swapping when working with Intel x64 64-bit systems such as Windows and Linux x86.

The topic for this technote applies to several different scenarios when working with Intel-based architectures when using IBM MQ. But the most common scenario is this:
You are dealing with undelivered messages found in the Dead Letter Queue (DLQ) of a queue manager and you want to find out the reason why a message was sent to the DLQ.

++ Summary

This tutorial provides all the steps to help you do a byte swap of the following 4 bytes:
22 08 00 00

The following visual aid can be help you to do all the steps.
Before a byte swap: 22 08 00 00

```
22 08 00 00  => 00 00 08 22
         ++------++
       ++ --------- ++
    ++ --------------- ++
++ --------------------- ++
```

After a byte swap: 00 00 08 22
It can be represented in Hexadecimal as: 0x00000822

+ Related articles

a) The following article has a good explanation on using the MQ Explorer and amqsbcg to look at the reason code.

https://www.ibm.com/support/pages/node/6589941
How to find reason code for message that was sent to the IBM MQ Dead Letter Queue - DLQ

b) Another scenario when the swapping of bytes might be necessary is when an MQ JMS client application uses the JMS Delivery Delay feature and the delayed message is temporarily stored in the SYSTEM.DDELAY.LOCAL.QUEUE and the message has a header DDH that indicates the delay:

https://www.ibm.com/support/pages/node/7151033
Calculating the delay time for a message in SYSTEM.DDELAY.LOCAL.QUEUE in an IBM MQ queue manager

+++ Background +++

"Byte Swapping" usually means reversing the order of the 2 bytes within each aligned pair. Such as, converting all the 2-byte words between big/little endian.

I work for IBM MQ Distributed Support and many of our customers use Intel-based hardware architectures, which are "Little Endian" (see bellow for more details).
Thus, the audience for this document are MQ Administrators who use Intel-based systems, such as Windows and Linux x86.

Quick question: How do you know if your Linux is using a x86 architecture?
Answer:
You can use the following command.
If you see "x86_64" in the output, then you are using x86.

$ **uname -a**
Linux riggioni1.fyre.ibm.com 4.18.0-477.27.1.el8_8.x86_64 #1 SMP Thu Aug 31 10:29:22 EDT 2023 x86_64 x86_64 x86_64 GNU/Linux

Hardware architectures that use POWER or SPARC (used by AIX, HPe Non Stop Server NSS, Solaris SPARC and HP-UX), or EBDCID (z/OS, IBMi) use the "Bit Endian" method to represent Bytes in a hex dump, when reading from LEFT to RIGHT, the MOST significant bytes are shown FIRST.

While the hardware architectures based on Intel x86 use the "Little Endian" method to represent Bytes in a hex dump, when reading from LEFT to RIGHT, the MOST significant bytes are shown LAST. This is very COUNTERINTUITIVE!!!

https://en.wikipedia.org/wiki/Endianness#Byte_addressing
Endianness
"
In computing, endianness is the order or sequence of bytes of a word of digital data in computer memory or data communication which is identified by describing the impact of the "first" bytes, meaning at the smallest address or sent first.
Endianness is primarily expressed as big-endian (BE) or little-endian (LE):
- A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest.
- A little-endian system, in contrast, stores the least-significant byte at the smallest address.
"

In particular, the following section has one of the best explanations that I have seen about this topic:

.

Byte addressing[edit]

See also: Byte addressing

When memory bytes are printed sequentially from left to right (e.g. in a hex dump), little-endian representation of integers has the significance increasing from left to right. In other words, it appears backwards when visualized, which can be counter-intuitive.

This behavior arises, for example, in FourCC or similar techniques that involve packing characters into an integer, so that it becomes a sequences of specific characters in memory. Let's define the notation `'John'` as simply the result of writing the characters in hexadecimal ASCII and appending `0x` to the front, and analogously for shorter sequences (a C multicharacter literal):

```
   '  J  o  h  n  '
hex  4A 6F 68 6E
---------------
   -> 0x4A6F686E
```

On big-endian machines, the value appears left-to-right, coinciding with the correct string order for reading the result:

increasing addresses →

...	$4A_h$	$6F_h$	68_h	$6E_h$	...
...	'J'	'o'	'h'	'n'	...

But on a little-endian machine, one would see:

increasing addresses →

...	$6E_h$	68_h	$6F_h$	$4A_h$	...
...	'n'	'h'	'o'	'J'	...

+++ Answer +++

The following technote provides the answer:
https://www.ibm.com/support/pages/node/6589941
How to find reason code for message that was sent to the IBM MQ Dead Letter Queue DLQ

In particular, you use the command:
amqsbcg DLQ QMgrName > DLQ-out.txt

When the queue manager sends a message to the DLQ, a Dead Letter Header (DLH) will be added to the message, and it is shown at the top of the section for the Payload when using the amqsbcg command.

In the output file from amqsbcg, take a look at the FIRST line of the payload (after the line for "length") contains the part of the DLH that has the reason code.

Notice the value for the 9-12 bytes (5th and 6th pair from the output).

For example:

```
**** Message ****
length - 697 bytes
00000000: 444C 4820 0100 0000 2208 0000 5143 315F 'DLH ....."...QC1_'
                              **** ****
                              5th  6th  pair of bytes
```

To better visualize these bytes, let's isolate them from the other bytes in the 1st line and then let's add a space to clearly mark each individual byte:

```
value                         22 08 00 00
byte position                  9 10 11 12
```

In AIX systems, you could use the "mqrc" command to find out the meaning of these 4 bytes. You have to compose the bytes all together and without space and then add a "0x" at the beginning to indicate that it is hexadecimal.

```
value                         22 08 00 00
```

Remove spaces:
```
22080000
```

Add the prefix: 0x
```
0x22080000
```

Then provide this hexadecimal number to the following MQ utility:
```
mqrc 0x22080000
```

But it will give you an error message:
```
No matching return codes
```

Why this error message?
In Windows and Linux Intel, you need to take into account the swapping of the bytes.

For some folks, the swapping is a trivial manner, and they can do it on their head.
But for other folks (like me!) it is a bit harder, and thus, I wanted to share my method, showing every step:

1: let's show again the individual 4 bytes, exactly in the order in which they appear in the 1st line from the payload/content section of the amqsbcg command:

```
value          22 08 00 00
byte position   9 10 11 12
```

2: I like to draw an arrow to indicate a "transformation" in my mind:

```
value          22 08 00 00  ===>
byte position   9 10 11 12
```

3: Start with the byte in the LAST position (position 12th) and copy it immediately after the arrow:

```
                     Start
                     Here
                       ||
value          22 08 00 00  ===>  00
byte position   9 10 11 12        12
                        ** ======>**
```

4: Then proceed to handle the byte in the 11th position and copy to the right, beyond the byte that was copied in step 3 above:

```
value          22 08 00 00  ===>  00 00
byte position   9 10 11 12        12 11
                     ** ==========>**
```

5: Do the same for the byte in the 10th position and copy to the right, beyond the byte that was copied in step 4 above:

```
value          22 08 00 00  ===>  00 00 08
byte position   9 10 11 12        12 11 10
                  ** ================>**
```

6: Finally, do the byte in the 9th position:

```
value          22 08 00 00  ===>  00 00 08 22
byte position   9 10 11 12        12 11 10  9
               ** ========================>**
```

7: This is the result about the byte swap:

```
value                             00 00 08 22
byte position                     12 11 10  9
```

8: Let's remove spaces

```
00000822
```

9: Add the prefix: 0x

```
0x00000822
```

10: Use the MQ utility mqrc:

```
mqrc 0x00000822
```

2082 0x00000822 MQRC_UNKNOWN_ALIAS_BASE_Q

++ INCORRECT swappings

+ Do not swap each number for the byte!

```
value          21 08 ===> 8 0 1 2  INCORRECT!!
                   + -----+
                  + --------+
                 + ----------+
                + ----------------+
```

+ Do not swap the pairs of bytes (amqsbcg shows 2 bytes as a pair)

```
value          2208 0000 ===> 0000 2208  INCORRECT!!
                    ++++ ---- ++++
               ++++ -------------- ++++
```

+++ end +++