MQ PCF Programming

Mark Taylor, IBM Hursley
marke_taylor@uk.ibm.com
What is PCF

• Programmable Command Format

• A "self-describing" MQ message used for administrative operations

• Your programs can send commands and get responses using PCF
  – Equivalent to "DISPLAY QSTATUS" or "ALTER CHANNEL"

• MQ emits events in PCF format
  – "Queue is getting full"
  – An application issued an MQGET with these parameters
A simple PCF message

- MQMD Format tells that what follows is PCF
- Body of message contains MQCFH structure followed by elements
  - Normally no user data beyond the elements
- Elements (eg MQCFIN) have a type, length and value
  - Basic types for integer, integer array, string, string array, byte string
  - Additional types for groups and filtered inquiries

| MQMD Format=MQEVENT, MQADMIN or MQPCF | MQCFH Parameter Count Reason | MQCFIN | MQCFIN | MQCFST | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
typedef struct tagMQCFST MQCFST;
typedef MQCFST MQPOINTER PMQCFST;

struct tagMQCFST {
    MQLONG    Type;   /* Structure type */
    MQLONG    StrucLength;   /* Structure length */
    MQLONG    Parameter;   /* Parameter identifier */
    MQLONG    CodedCharSetId; /* Coded character set identifier */
    MQLONG    StringLength;  /* Length of string */
    MQCHAR    String[1];    /* String value -- first character */
};
A PCF element for filtering

```c
typedef struct tagMQCFIF MQCFIF;
typedef MQCFIF MQPOINTER PMQCFIF;

struct tagMQCFIF {
    MQLONG Type;        /* Structure type */
    MQLONG StrucLength; /* Structure length */
    MQLONG Parameter;   /* Parameter identifier */
    MQLONG Operator;    /* Operator identifier */
    MQLONG FilterValue; /* Filter value */
};
```
An event message

**** Message length - 300 of 300 bytes ***

00000000: 0000 0007 0000 0024 0000 0003 0000 0063 '.......$. .......c'
00000010: 0000 0001 0000 0001 0000 0000 0000 096C '................l'
00000020: 0000 0002 0000 0014 0000 0010 0000 1F41 '................A'
00000030: 0000 0004 0000 0004 0000 0020 0000 0BE5 '................å'
00000040: 0000 0333 0000 000C 6D65 7461 796C 6F72 '...3...metaylor'
00000050: 2020 2020 0000 0003 0000 0010 0000 03F3 '............ó'
00000060: 0000 0001 0000 0004 0000 0044 0000 0BE7 '............D. ç'
00000070: 0000 0333 0000 0030 5638 3030 335F 4120 '...3...0V8003_A '
000000A0: 2020 2020 2020 2020 0000 0003 0000 0010 '                '
000000B0: 0000 03FD 0000 005A 0000 0014 0000 0010 '...ý...Z........'
000000C0: 0000 1F42 0000 0004 0000 0004 0000 0018 '...B............'
000000D0: 0000 0BFB 0000 0000 0000 0001 5800 0000 '...û........X...
000000E0: 0000 0003 0000 0010 0000 03F8 0000 0001 '.............φ'
000000F0: 0000 0006 0000 0024 0000 0BF9 0000 0000 '...........$.ù'
00000100: 0000 0001 0000 0008 6D65 7461 796C 6F72 '......metaylor'
00000110: 0000 0000 0000 0005 0000 0018 0000 045C '.............\'
00000120: 0000 0002 0000 000B 0000 0009

© Copyright IBM Corporation 2016
An event message

**** Message length - 300 of 300 bytes ***

00000000: 0000 0007 0000 0024 0000 0003 0000 0063 ........$...c'
00000010: 0000 0001 0000 0001 0000 0000 0000 096C '................l'
00000020: 0000 0002 0000 0014 0000 0010 0000 1F41 '................A'
00000030: 0000 0004 0000 0004 0000 0020 0000 0BE5 '................â'
00000040: 0000 0333 0000 000C 6D65 7461 796C 6F72 '...3...metaylor'
00000050: 2020 2020 0000 0003 0000 0010 0000 03F3 '...............ó'
00000060: 0000 0006 0000 0004 0000 0044 0000 0BE7 '...............D...ç'
00000070: 0000 0333 0000 0030 5638 3030 335F 4120 '...3...0V8003_A'
000000A0: 2020 2020 2020 2020 0000 0003 0000 0010 '...........'
000000B0: 0000 03FD 0000 005A 0000 0014 0000 0010 '............ý...Z...'
000000C0: 0000 1F42 0000 0004 TYPE (cfst) LEN (24) DATA
000000D0: 0000 0003 0000 0010 0000 03F8 0000 0001 '...........ø'
000000E0: 0000 0006 0000 0024 0000 0BF9 0000 0000 '.........$...ù...'
000000F0: 0000 0001 0000 0008 6D65 7461 796C 6F72 '...metaylor'
00000100: 0000 0000 0000 0005 0000 0018 0000 045C '................\'
00000110: 0000 0002 0000 000B 0000 0009
An event message decoded

Event Type : Command Event
Reason : Command MQSC
Event created : 2015/06/03 13:28:20.51 GMT
Correlation ID : 414D512056383030335F412020202020556F00F120001E05

COMMAND CONTEXT
Event User Id : metaylor
Event Origin : Console
Event Queue Mgr : V8003_A
Command : Set Auth Rec

COMMAND DATA
Auth Profile Name : X
Object Type : Queue
Principal Entity Names : metaylor
Auth Add Auths : Output
: Input
A set of statistics events decoded in MS0P

- Time period
- MQ Statistics at QMGR level
- Detailed queue statistics
Why PCF

• MQSC intended for human consumption
  – Parsable by eye, less easy in programs
  – For example, `DESCR('This is 'an' object description with quote & paren(')`
  – No guaranteed ordering in runmqsc, two-column output

• PCF intended for programs
  – Can tell exactly what the parameter is for, its length and value
  – But cannot easily be scripted

• Approximately one-one mapping between MQSC commands and PCF
  – MQSC DISPLAY == PCF Inquire
  – MQSC ALTER == PCF Change

• Remember that PCF invented before formats like JSON or XML
  – And there are many MQ apps that are built on PCF
Basic pattern for commands

- **Application**
- **MQCFH (PCF header)**
- **PCF parameters**
- **PCF request message**
- **Queue manager** (PCF command server)
- **Reply queue**
- **Command queue**
- **PCF response messages** (1 or more)
How is z/OS different

- z/OS started to support real PCF for commands from V6
- But command server requires extended-format PCF messages

<table>
<thead>
<tr>
<th>Other platforms</th>
<th>z/OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCFH in requests has type MQCFT_COMMAND</td>
<td>MQCFH in requests has type MQCFT_COMMAND_XR</td>
</tr>
<tr>
<td>Single type of response message (MQCFT_RESPONSE)</td>
<td>Up to three types of response message (MQCFT_XR_MSG, MQCFT_XR_ITEM, MQCFT_XR_SUMMARY)</td>
</tr>
<tr>
<td>MQCFH Control field indicates last response</td>
<td>MQCFH Control field indicates last response in group</td>
</tr>
<tr>
<td>Single logical collection of responses</td>
<td>Responses may be grouped into multiple logical groups (grouping information embedded in response parameters)</td>
</tr>
</tbody>
</table>
z/OS CMDSCOPE

- One command on z/OS may be fanned out to multiple qmgrs in QSG

- Initial response shows how many queue managers
- Followed by separate set of responses from each queue manager
- Followed by a final set from the original processing qmgr

- Can see a lot of this if you use client-mode runmqsc to z/OS system
Transactions and serialisation

- There is no point in putting PCF commands inside a transaction
  - Command server will always process them individually (GET – NO SYNC)
  - No way for command server to back out updates

- On Distributed, cmd server completes command before continuing
  - All replies generated etc before reading next command from input queue

- On z/OS, commands may overlap
  - Put two commands, responses may come in any order
Writing PCF programs

- Most PCF processors are C or Java
  - There is at least one COBOL sample out there too

- Some use .Net classes: that interface is not documented or maintained
  - An historic accident
  - Missing newer function such as z/OS and byte string support

- See also github.com/ibm-messaging/mq-golang for a source-included Go interface to MQ and PCF
  - Used to feed data to a Grafana dashboard

- There is also MQAI originally intended to simplify programs but it too does not contain all function
  - Has concept of "bags" – put in elements, pull out elements
  - In particular, extensions to support z/OS formats are missing
  - And is not available on z/OS
"Escape" PCF

- One trivial use of PCF is to wrapper an MQSC command
  - MQCMD_ESCAPE

- Allows use of the MQSC text instead of building messages with individual parameters
  - Can build your own "runmqsc" command-line interface using this

- Main problem with its use is that the response comes back in text too
  - Cannot be easily parsed

- Before V6, remote MQSC to z/OS was done without PCF wrapper
  - Simply put the command as a string to the command queue
    - MQMD.Format = "MQSTR"
    - Still works of course
The KnowledgeCentre is essential reference

For each command it shows required and optional parameters

But has been known to sometimes be wrong!

- Particularly for reqd and optional distinctions

Will see debug assistance later
Processing events

• Events appear on well-known queues (unless redefined)
• Generally independent
• CFH contains reason for the event followed by additional information
  – eg MQRC_QUEUE_FULL

• Command and Configuration events may both be generated for the same operation
  – Config events have BEFORE and AFTER sets of all attributes of the object
  – All events generated for the same command have the same CorrelId
  – Command Events contain two PCF groups: who did it; what did they do
IBM MQ

MQEPH – Embedded PCF Header

- Designed to allow applications to add tracking information as they process messages
  - Shows relationships between messages
- The only time user data may follow PCF elements

- But noone seems to use these so I won't talk more about them
  - Other than in their canned format for traceroute (dspmqrte) messages
MQ Appliance and MQ V9 – Application Activity Events

- Application activity events record MQI calls
  - Some vendor products use API exits to record similar data
- Normally configured by editing mqat.ini
  - Not possible on the appliance: also no exits
- Alternative is to subscribe to topics
  - Dynamic enable/disable
  - Allowing multiple consumers

- Basic topic is "$SYS/MQ/INFO/QMGR/<qmgr name>/ActivityTrace"
  - Can then add "/ApplName/amqsputc.exe"
  - Or "/ChannelName/SYSTEM.DEF.SVRCONN"
- Events in same format as on other platforms

- https://www.ibm.com/developerworks/community/blogs/messaging/entry/Tracing_applications_with_the_MQ_Appliance
IBM MQ

MQ Appliance and MQ V9 – System and qmgr monitors

• The MQ Appliance introduced a new style of event generation
  – Also available in MQ V9
  – Allows multiple consumers of the same information
  – Information may change – no guarantees that data in release N will also be generated in release N+1
  – Always non-persistent to avoid capability changes causing migration issues

• A monitoring application subscribes to a well-known (meta-)topic
  – MQSUB("$SYS/MQ/INFO/QMGR/<qmgrname>/Monitor/METADATA")
  – Publications then respond with which real information is available
  – And the monitor can then choose to subscribe to specific topics
  – Possible topics include CPU, Disk and MQI statistics

• Each publication is still in PCF format
Feeding a dashboard

MQ PROMETHEUS CAPTURE

Queue Activity

CPU

Log write latency

File system

© Copyright IBM Corporation 2016
PROGRAMMING PCF IN C
Sending a command message

- Decide which command and parms you want to issue
- In one buffer, add the MQCFH followed by required parameters followed by optional parameters

- Open the SYSTEM.ADMIN.COMMAND.QUEUE for output
  - Add remote qmgr name to send to other queue managers
- Open a reply queue for input
  - Often a TDQ but not a good idea if you are doing this frequently

- Set MQMD.Format to MQADMIN
- Set MQMD.MsgType to MQMT_REQUEST
- Set MQMD.ReplyToQ

- Put the message
/******************************************************************/
/* Put the command to the COMMAND.QUEUE                           */
******************************************************************/
memcpy(&md, &DefaultMD, sizeof(MQMD));
memcpy(md.Format, MQFMT_ADMIN, (size_t)MQ_FORMAT_LENGTH);
strncpy(md.ReplyToQ, ResolvedReplyQ, MQ_Q_NAME_LENGTH);
md.MsgType = MQMT_REQUEST;
md.Report = MQRO_COPY_MSG_ID_TO_CORREL_ID;

memcpy(&pmo, &DefaultPMO, sizeof(MQPMO));
pmo.Options |= MQPMO_NO_SYNCPOINT;
pmo.Options |= MQPMO_NEW_MSG_ID;
pmo.Options |= MQPMO_NEW_CORREL_ID;

MQPUT(Hcon,                /* connection handle               */
    HCmdQ,               /* object handle                   */
    &md,                 /* message descriptor              */
    &pmo,                /* default options (datagram)      */
    offset,              /* message length                  */
    buffer,              /* message buffer                  */
    &CompCode,           /* completion code                 */
    &Reason);            /* reason code                     */
Working with responses

- Wait for replies, parsing each until returned CFH.Control contains MQCFC_LAST
- CFH.Reason may be an MQRC or MQRCCF error code for failures

- CFH.ParameterCount shows how many elements are in this message
  - An entire group containing multiple elements CFH only counts as 1 here
  - The group element shows how many elements are contained in it

- Walk through the elements, casting pointers to appropriate datatype before fully decoding each one
  - One convenience – can inspect an element to discover its type before really knowing as the type is always at the same offset (0) in the structure
Starting to decode a response message

```c
offset = sizeof(MQCFH);
for (counter = 0;
     counter < pCfh->ParameterCount;
     counter++)
{
    MQLONG Type;
    MQLONG StrucLength;

    /* All elements have the Type and StrucLength in the same offsets */
    Type = *(MQLONG *)&(buffer[offset]);
    StrucLength = *(MQLONG *)&(buffer[offset + sizeof(MQLONG)]);

    switch (Type)
    {
    case MQCFT_BYTE_STRING:
        pCfbs = (MQCFBS *)&(buffer[offset]);
        ...
```

* Need to adjust this if processing groups
Alignment and lengths - Types

- All structures must begin on a 32-bit boundary, the size of an MQLONG
- Some elements are naturally of the correct size (MQCFIN)
- But others require rounding up and padding

- HOWEVER no guarantee that 64-bit values begin on 64-bit boundaries
  - Can be an issue on some platform architectures (particularly Sparc)
    
    ```
    MQINT64 v = mqcfin64->Value;  /* May fail with SIGBUS */
    ```
  - Have to access via memcpy
    
    ```
    MQINT64 v;
    memcpy(&v, &(mqcfin64->Value), sizeof(MQINT64));
    ```

- Structures contain length of structure AND (if needed) length of data
Alignment and lengths - Arrays

- MQCFSL structure contains an array of strings
  - All strings must be the same length
  - All strings must start on a 4-byte boundary

- Array PCF structures are defined with a single entry
  - PCF created before (most) C compilers supported empty arrays in structures
    ```
    struct { type name[0];} or struct { type name[];}
    ```

- So if using malloc, may want to use something like
  ```
  malloc(sizeof(MQCFIL) + (N-1)*sizeof(MQLONG))
  ```
Space Padding

- When working with MQ objects (eg queue names) it's always best to fully pad with spaces to 48 (or 20) characters
  - Some commands seem to be OK with NULL terminators
  - Some commands seem to be OK with shorter parameters

- But I've seen occasional problems unless I did it this way

- At least these object name lengths are already rounded to 4 bytes

- Other string attributes do not need to be set to their maximum length

- String lengths do not include NULL terminators
Some broken code

- The next code fragment was sent from a developer trying to use the trace-route facility
  - I've cut it down from the fuller program he sent, but the main point is there

- The coder had converted it from a Java sample I had sent him

- Problem description was "it's not working"

- Can you see what was wrong
What's wrong here?

PMQCFIN pPCFInteger = <start of a buffer>
...

pPCFInteger = pPCFInteger + MQCFIN_STRUC_LENGTH;
pPCFInteger->Type = MQCFT_INTEGER;
pPCFInteger->StrucLength = MQCFIN_STRUC_LENGTH;
pPCFInteger->Parameter = MQIACF_MAX_ACTIVITIES;
pPCFInteger->Value = 1000;

pPCFInteger = pPCFInteger + MQCFIN_STRUC_LENGTH;
pPCFInteger->Type = MQCFT_INTEGER;
pPCFInteger->StrucLength = MQCFIN_STRUC_LENGTH;
pPCFInteger->Parameter = MQIACF_ROUTE_ACCUMULATION;
pPCFInteger->Value = MQROUTE_ACCUMULATE_NONE;
PMQCFIN pPCFInteger;
char *p = <start of buffer>
...
pPCFInteger = (PMQCFIN)p;
pPCFInteger->Type = MQCFT_INTEGER;
pPCFInteger->StrucLength = MQCFIN_STRUC_LENGTH;
pPCFInteger->Parameter = MQIACF_MAX_ACTIVITIES;
pPCFInteger->Value = 1000;
pCFH->ParameterCount++;
p += MQCFIN_STRUC_LENGTH;

pPCFInteger = (PMQCFIN)p;
pPCFInteger->Type = MQCFT_INTEGER;
pPCFInteger->StrucLength = MQCFIN_STRUC_LENGTH;
pPCFInteger->Parameter = MQIACF_ROUTE_ACCUMULATION;
pPCFInteger->Value = MQROUTE_ACCUMULATE_NONE;
pCFH->ParameterCount++;
p += MQCFIN_STRUC_LENGTH;
Why do it my way?

• An alternative to the original "bad" code would be

```c
pPCFInteger = pPCFInteger + 1;
```

• But my style works when more datatypes are involved
  – Easier to work out how much to increment

• Experience shows it's easy to move blocks around and add parameters

• These blocks often encapsulated in macros for even easier reuse
  – Especially when (as in the MQ source code) there is lots of this going on
char *ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE";
PMQCFST cfst = <start of buffer>
PMQCFIN cfin;
...

memcpy(cfst->String, objectName, strlen(ObjectName));
DataLength = sizeof(*cfst) + strlen(ObjectName);

cfst->StrucLength = DataLength;

cfin = (char *)cfst + DataLength;
...

More mistakes – all on one line - corrected

```c
char *ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE";
PMQCFST cfst = <start of buffer>
PMQCFIN cfin;
...

#define RoundUp4(n) (((n)+3) & ~0x03)
memcpy(cfst->String,ObjectName,strlen(ObjectName));
ElemLength = RoundUp4(MQCFST_STRUC_LENGTH_FIXED +
                      strlen(ObjectName));

cfst->StrucLength = ElemLength;
cfst->DataLength = strlen(ObjectName);

cfin = (char *)cfst + ElemLength;
...
```
Using the MQAI – List queues and their depth

```c
mqCreateBag(MQCBO_ADMIN_BAG, &adminBag, &CC, &RC);
mqCreateBag(MQCBO_ADMIN_BAG, &responseBag, &CC, &RC);
mqAddString(adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, "+", &CC, &RC);
mqAddInteger(adminBag, MQIA_Q_TYPE, MQQT_LOCAL, &CC, &RC);
mqAddInquiry(adminBag, MQIA_CURRENT_Q_DEPTH, &CC, &RC);

mqExecute(hConn, MQCMD_INQUIRE_Q, MQHB_NONE, adminBag,
          responseBag, MQHO_NONE, MQHO_NONE, &CC, &RC);

mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags, &CC, &RC);

for (i=0; i<numberOfBags; i++) {
  mqInquireBag(responseBag, MQHA_BAG_HANDLE, i, &qAttrsBag, &CC, &RC);
  mqInquireString(qAttrsBag, MQCA_Q_NAME, 0, MQ_Q_NAME_LENGTH, qName,
                   &qNameLength, NULL, &CC, &RC);
  mqInquireInteger(qAttrsBag, MQIA_CURRENT_Q_DEPTH, MQIND_NONE, &qDepth,
                   &CC, &RC);
  mqTrim(MQ_Q_NAME_LENGTH, qName, qName, &CC, &RC);
  printf("%4ld %-48s\n", qDepth, qName);
}
```
Element classes

• MQCFH, MQCFIN, MQCFST, MQCFIL, MQCFSL, MQCFBS…
  – Classes representing MQ PCF structure elements
  – Share basic operations defined in PCFHeader interface
    • Methods to read and write message content: initialize(), write(), size()
    • PCFParameter interface adds getValue(), setValue(), getParameter() etc.
    • PCFParameter.nextParameter() parses PCF content in MQMessages

• Packages com.ibm.mq.pcf, com.ibm.mq.headers.pcf both define these classes
  – Pick one and stick with it (just in case)
  – I use the "headers" packages
  – Never attempt to use anything from the "jmqi" packages

• Generally used with MQ basic Java classes, not JMS
  – Though it is possible to use JMS
PCFAgent class

- PCFAgent manages interaction with queue manager
  - Maintains connection handle, request and reply queue handles
  - Puts PCF request messages on command queue, gathers responses

- Automatically detects target platform and version during connection
  - Sets MQCFH values in request accordingly, and gathers responses using z/OS or traditional scheme
PCFMessage and PCFMessageAgent

• PCFMessageAgent extends PCFAgent
  – Represents request and response messages using instances of PCFMessage, rather than raw MQCFH/PCFParameter instances
  – Not available for JMS programs

• PCFMessage provides convenience methods for adding/retrieving PCF parameters, automatically updating MQCFH fields
Sending a command message

```java
String[] names;

PCFMessageAgent agent = new PCFMessageAgent("localhost", 1414, "CLIENT");
PCFMessage request = new PCFMessage (CMQCFC.MQCMD_INQUIRE_Q_NAMES);
request.addParameter (CMQC.MQCA_Q_NAME, "*");
request.addParameter (CMQC.MQIA_Q_TYPE, MQC.MQQT_LOCAL);

PCFMessage [] responses = agent.send (request);
names = (String[])responses[0].getParameterValue(CMQCFC.MQCACF_Q_NAMES);

for (int i = 0; i < names.length; i++) {
    System.out.println("Queue: " + names [i]);
}
```
Another way to add elements to a message

MQMessage msg = new MQMessage();
msg.replyToQueueName = replyQueue.getName();
msg.replyToQueueManagerName = qMgr.getName();
msg.messageType = CMQC.MQMT_REQUEST;
msg.characterSet = CMQC.MQCCSI_DEFAULT;
msg.persistence = CMQC.MQPER_NOT_PERSISTENT;
msg.report = CMQC.MQRO_ACTIVITY | CMQC.MQRO_DISCARD_MSG;
msg.format = CMQC.MQFMT_ADMIN;

MQCFH.write(msg, CMQCFC.MQCMD_TRACE_ROUTE, 1, CMQCFC.MQCFH_VERSION_3);
MQCFGR.write((Object) msg, CMQCFC.MQGACF_TRACE_ROUTE, 8); // number in grp
MQCFIN.write((Object) msg, CMQCFC.MQIACF_ROUTE_DETAIL, CMQCFC.MQROUTE_DETAIL_MEDIUM);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_RECORDED_ACTIVITIES, 0);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_UNRECORDED_ACTIVITIES, 0);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_DISCONTINUITY_COUNT, 0);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_MAX_ACTIVITIES, 1000);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_ROUTE_ACCUMULATION, CMQCFC.MQROUTE_ACCUMULATE_NONE);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_ROUTE_FORWARDING, CMQCFC.MQROUTE_FORWARD_ALL);
MQCFIN.write((Object) msg, CMQCFC.MQIACF_ROUTE_DELIVERY, CMQCFC.MQROUTE_DELIVER_NO);

targetDestination.put(msg, pmo);
Parsing an event message

MQMessage msg;

queue.get(msg,gmo);

MQCFH cfh = new MQCFH(msg);
int cnt = cfh.getParameterCount();
int cmd = cfh.getCommand();

PCFParameter p;
try {
    int valInt;

    while ((p = PCFParameter.nextParameter(msg)) != null) {
        switch (p.getType()) {
            case CMQCFC.MQCFT_INTEGER :
                valInt = ((Integer) p.getValue()).intValue(); /* older java!*/
                break;
            ...
        }
    }
}
```
if (jmsMessage.getStringProperty("JMS_IBM_Format").trim().equals("MQADMIN")) {
    parsePCF(jmsMessage);
}

parsePCF(BytesMessage jmsMessage) throws JMSException {
    long msgLength = jmsMessage.getBodyLength();
    byte[] data = new byte[(int) msgLength];
    jmsMessage.readBytes(data);
    MQMessage mqMsg = new MQMessage();
    mqMsg.write(data);
    // Make sure we can work on little-endian systems – the simple use of
    // DataInputStream(ByteArrayInputStream(data)) doesn’t always work
    mqMsg.encoding = jmsMessage.getIntProperty("JMS_IBM_Encoding");
    mqMsg.format = jmsMessage.getStringProperty("JMS_IBM_Format");
    mqMsg.seek(0);
    PCFMessage pcfMessage = new PCFMessage(mqMsg);
    ...
}
```
void producerSendMessage(final PCFMessage pcfMessage, MessageProducer producer, Queue responseQueue) throws JMSException, IOException {

    BytesMessage message = new BytesMessage();
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutput dataOutput = new DataOutputStream(baos);

    pcfMessage.write(dataOutput);
    baos.flush();
    message.writeBytes(baos.toByteArray());

    message.setStringProperty(WMQConstants.JMS_IBM_MQMD_FORMAT, "MQADMIN");
    message.setStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET, Charset.defaultCharset().name());

    producer.send(message);
    baos.close();
}
C or Java or something else?

- Often have no choice – language is dictated by larger environment

- From my own programs
  - Eclipse/Explorer plugins forced Java
  - Desire for MQCB forced C
  - Working with Prometheus database strongly encouraged Go

- Java hides many complexities
  - String handling is better
  - Multiple z/OS command responses are automatically combined
  - Enumerators assist with parsing but I had problems with MQCFGR enumeration

- C probably easier to debug when it does go wrong
  - Details are not hidden
  - But it is more likely to go wrong!
SOLVING PROBLEMS
Incorrect parameters

• Validation of commands is done by command server  
  – Parameter checks, lengths etc

• MQRC/MQRCCF in the response message CFH shows problems

• Parameters may need to be provided in a particular order  
  – Always put the required parameters first

• Example: Not setting the ObjectType parameter in SetAuthRec  
  – So command had AuthProfileName, ObjectType, List of Names, Authorities  
  – Returned MQRCCF_CFSL_PARM_ID_ERROR  
  – It was not expecting to see a CFSL at that point in the parsing

• Example: Not sending the right length message  
  – MQPUT(bufferlength/2) …  
  – MQRCCF_MSG_LENGTH_ERROR
Looking at a trace

- Sometimes found it helpful to look at command server trace
- Process name is amqpcsea
  - strmqtrc -m <qmgr> -i <pid of command server>

- Look for function pcmUnpackMsgParms
  - Just above it will be something like pcm<Commandname>

- It is followed by information about each parameter it could decode
  - So you can see how far through your PCF command it got before failing
One nice feature of MQ Java is the MQConstants.lookup() method

- Returns the string definitions for a value with a filter
  - MQConstants.lookup(2035,"MQRC.*") returns "MQRC_NOT_AUTHORIZE\d"
- I used this a LOT in the MS0P event formatter
  - And wrappered it in the mqidecode program also in MS0P
- But you still need to know the filter to apply for each attribute
  - p.getParameter() returns 1; p.getParameterValue() returns 6
  - Call MQConstants.lookup(1,"MQIA.*") to get "MQIA_APPL\d_TYPE"
  - But you have then to know that ApplicationType corresponds to MQI constants beginning MQAT_
    - MQConstants.lookup(6,"MQAT.*") to get "MQAT_AIX/MQAT_UNIX"
    - There is nothing in the MQI that maps these value sets to attributes
- Have to manually create this attribute-to-filter map for your code
  - And deal with duplicate values (MQAT_MVS, MQAT_390, MQAT_ZOS)
Event formatting C sample in V8.0.0.4

• No sample previously shipped to format all "standard" events
  – Authorisation, queue full, service interval, command/config etc
  – Other samples are available for acct/stats, activity reports
  – Several SupportPacs but product only has out-of-date source code in the KC

• The **amqsevt** program formats events into readable English-ish text
  – Option to stay with full MQI constant name instead of making it look nice
  – Uses MQCB to read from multiple event queues. No polling required
  – Can connect as client to any remote queue manager including z/OS
  – Source code included

• Includes C header file to help convert MQI numbers to strings
  – Many developers have MQI strerror-like functions that typically need manual updates for new versions – this .h is automatically updated (300+ new verbs!)
  – Similar to Java MQConstants.lookup() capability for all sets of constants

```c
printf("Error is %s\n", MQRC_STR(2035));
```
Examples

**** Message #1 (320 Bytes) on Queue SYSTEM.ADMIN.QMGR.EVENT ****
Event Type : Queue Mgr Event [44]
Reason : Unknown Alias Base Queue [2082]
Event created : 2015/07/07 10:54:51.17 GMT
  Queue Mgr Name : V8003_A
  Queue Name : EVT.NO.BASE.QUEUE
  Base Object Name : EVT.NOT.DEFINED
  Appl Type : Unix
  Appl Name : amqsput
  Base Type : Queue

**** Message #4 (300 Bytes) on Queue SYSTEM.ADMIN.QMGR.EVENT ****
Event Type : Queue Mgr Event[44]
Reason : Not Authorized [2035]
Event created : 2015/07/07 10:54:51.30 GMT
  Queue Mgr Name : V8003_A
  Reason Qualifier : Open Not Authorized
  Queue Name : EVT.NO.PUT
  Open Options : 0x00002010 [ fiq out ]
  User Identifier : db2inst1
  Appl Type : Unix
  Appl Name : amqsput
Decoding attributes

switch (cfin->Parameter) {
    ...
    case MQIACH_LISTENER_CONTROL:
        fn = MQSVC_CONTROL_STR;
        break;
    case MQIA_CHINIT_TRACE_AUTO_START:
        fn = MQTRAXSTR_STR;
        break;
    case MQIA_CHLAUTH_RECORDS:
        fn = MQCHLA_STR;
        break;
    case MQIA_CLUSTER_PUB_ROUTE:
        fn = MQCLROUTE_STR;
        break;
    case MQIA_CLWL_USEQ :
        fn = MQCLWL_STR;
        break;
    ...
}
printf("%s %s\n", MQIA_STR(cfin->Parameter), fn(cfin->Value));
Example – Decoding attributes

**** Message #20 (3204 Bytes) on Queue SYSTEM.ADMIN.CONFIG.EVENT ****
Event Type : Config Event [43]
Reason : Config Change Object [2368]
Object state : Before Change
Event created : 2015/09/29 12:39:55.72 GMT
Event User Id : metaylor
Event Origin : Console
Event Queue Mgr : V8004_A
Object Type : Queue Mgr
Queue Mgr Name : V8004_A
Dead Letter Queue Name : SYSTEM.DEAD.LETTER.QUEUE
Def Xmit Queue Name :
Cert Label :
Conn Auth : SYSTEM.DEFAULT.AUTHINFO.IDPWOS
Command Input Queue Name : SYSTEM.ADMIN.COMMAND.QUEUE
Alteration Date : 2015-09-29
Alteration Time : 13.39.54
Queue Mgr Identifier : V8004_A_2015-09-24_10.07.13
Trigger Interval : 999999999
Max Handles : 256
Max Uncommitted Msgs : 10000
Authority Event : Enabled
Inhibit Event : Disabled
Samples

• Product includes several PCF-processing samples
  – amqsstop is a fairly simple issue-command-process-response example
  – amqslog dedicated to Logger events
  – amqsmon, amqsact more complex formatters
  – amqsclm inquires and sets queue attributes
  – amqsrua for the dynamic topic-based events

  – amqsailq is a sample using MQAI Bags
  – samples/pcf contains Java programs such as StartChannel, ListQueueNames

• SupportPacs etc
  – MS0S Java (source at https://github.com/ibm-messaging/mq-mqsc-editor-plugin)
  – MO01 C event formatter
  – MS12 COBOL event formatter
  – Go and PCF at https://github.com/ibm-messaging/mq-golang
Summary

- PCF is the best way to write programs to manage and monitor MQ

- Once you've done it a few times, it becomes easy and natural

- Lots of information to help you start
Any questions?