MQ PCF Programming

Mark Taylor, IBM Hursley

marke_taylor@uk.ibm.com

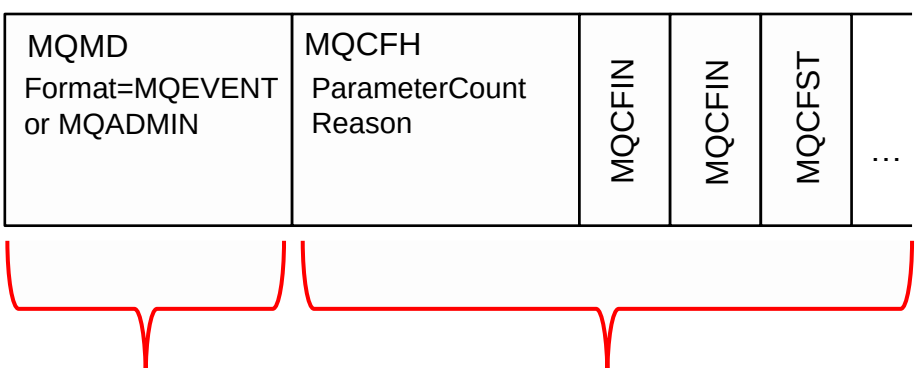
WARNING: PRESENTATION INCLUDES CODE

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



A PCF element

```
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

```
/**************************************************************/
/* MQCFIF Structure -- PCF Integer Filter Parameter           */
/**************************************************************/


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 '.......$.......c'
00000010:  0000 0001 0000 0001 0000 0000 0000 096C '...............l'
00000020:  0000 0002 0000 0014 0000 0010 0000 1F41 '...............A'
00000030:  0000 0004 0000 0004 0000 0020 0000 0BE5 '........... ...å'
00000040:  0000 0333 0000 000C 6D65 7461 796C 6F72 '...3....metaylor'
00000050:  2020 2020 0000 0003 0000 0010 0000 03F3 '    ............ó'
00000060:  0000 0001 0000 0004 0000 0044 0000 0BE7 '...........D...ç'
00000070:  0000 0333 0000 0030 5638 3030 335F 4120 '...3...0V8003_A '
00000080:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000090:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
000000A0:  2020 2020 2020 2020 0000 0003 0000 0010 '        ........'
000000B0:  0000 03FD 0000 005A 0000 0014 0000 0010 '...ý...Z........'
000000C0:  0000 1F42 0000 0004 0000 0004 0000 0018 '...B............'
000000D0:  0000 0BFB 0000 0000 0000 0001 5800 0000 '...û........X...'
000000E0:  0000 0003 0000 0010 0000 03F8 0000 0001 '...........ø....'
000000F0:  0000 0006 0000 0024 0000 0BF9 0000 0000 '.......$...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           '............    '
```

An event message

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

00000000:  0000 0007 0000 0024 0000 0003 0000 0063 '.......$.......c'
00000010:  0000 0001 0000 0001 0000 0000 0000 096C '...............l'
00000020:  0000 0002 0000 0014 0000 0010 0000 1F41 '...............A'
00000030:  0000 0004 0000 0004 0000 0020 0000 0BE5 '........... ...å'
00000040:  0000 0333 0000 000C 6D65 7461 796C 6F72 '...3....metaylor'
00000050:  2020 2020 0000 0003 0000 0010 0000 03F3 '    ............ó'
00000060:  0000 0001 0000 0004 0000 0044 0000 0BE7 '...........D...ç'
00000070:  0000 0333 0000 0030 5638 3030 335F 4120 '...3...0V8003_A '
00000080:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000090:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
000000A0:  2020 2020 2020 2020 0000 0003 0000 0010 '        ........'
000000B0:  0000 03FD 0000 005A 0000 0014 0000 0010 '...ý...Z........'
000000C0:  0000 1F42 0000 0004 0000 0018 0000 0010 '...B............'
000000D0:  0000 0333 0000 0030 0000 0000 0000 0058 '...û........X...'
000000E0:  0000 0003 0000 0010 0000 03F8 0000 0001 '...........ø....'
000000F0:  0000 0006 0000 0024 0000 0BF9 0000 0000 '.......$...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           '............    '
```

PARM (MQCA…)

CCSID (0)

TYPE (cfst)

LEN (1)

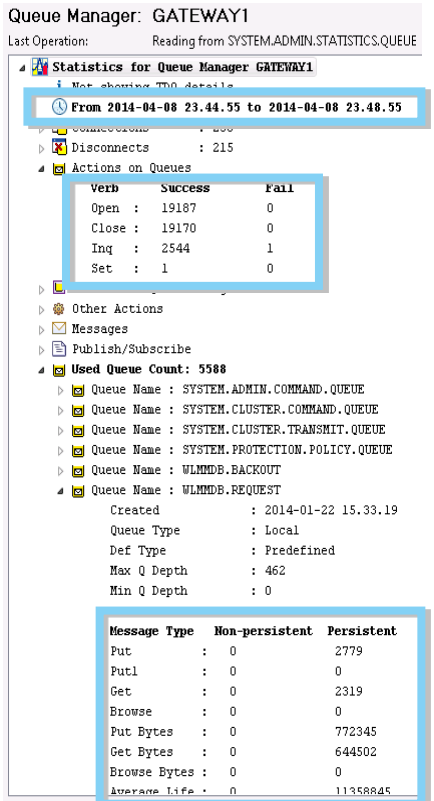
LEN (24)

DATA

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

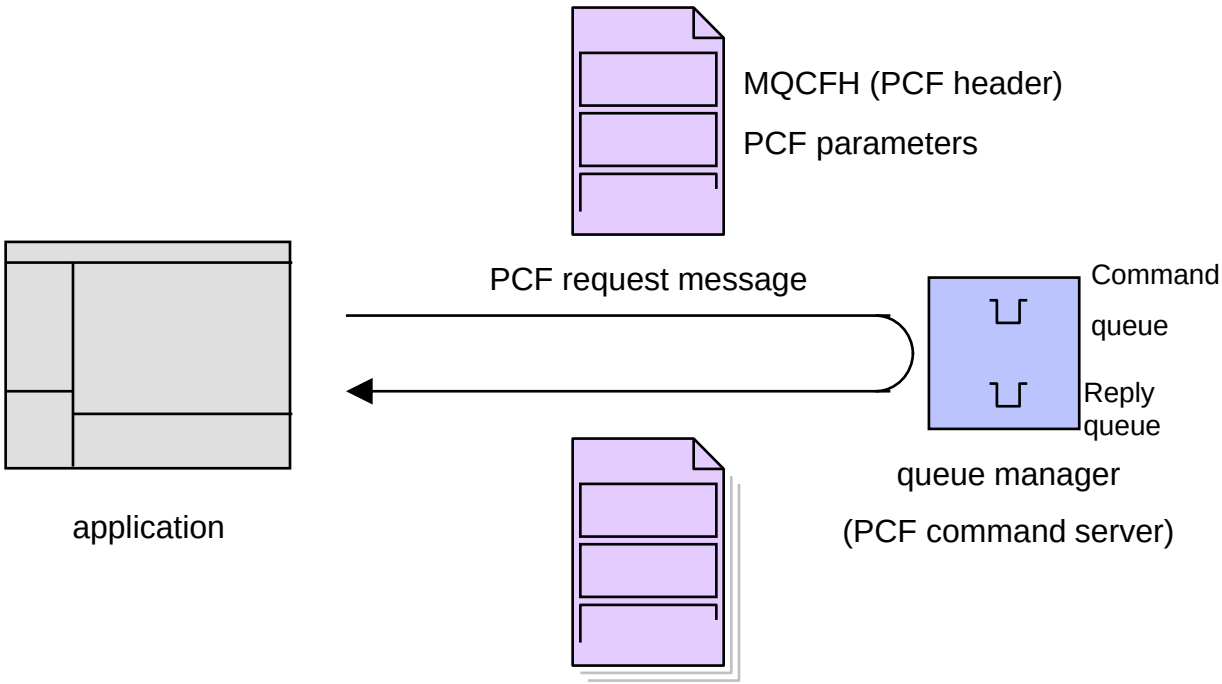


- Time period
- MQ Statistics at QMGR level
- Detailed queue statistics

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



How is z/OS different

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

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

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

Writing PCF programs

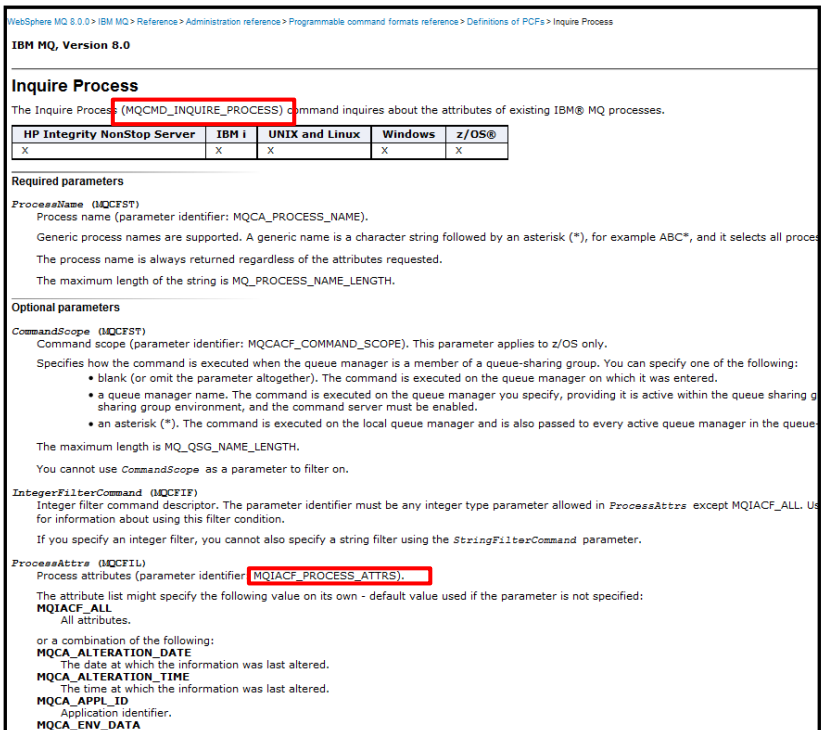
- Most PCF processors are C or Java
  - There is at least one COBOL sample out there too
- Some use .Net classes: that interface is not documented or maintained
  - An historic accident
  - Missing newer function such as z/OS and byte string support
- There is also MQAI originally intended to simplify 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 binary messages
  - 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

Documentation

- 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



WebSphere MQ 8.0.0 > IBM MQ > Reference > Administration reference > Programmable command formats reference > Definitions of PCFs > Inquire Process

IBM MQ, Version 8.0

Inquire Process

The Inquire Process (MQCMD_INQUIRE_PROCESS) command inquires about the attributes of existing IBM® MQ processes.

HP Integrity NonStop Server	IBM i	UNIX and Linux	Windows	z/OS®
X	X	X	X	X

Required parameters

ProcessName (MQCFST)
Process name (parameter identifier: MQCA_PROCESS_NAME).

Generic process names are supported. A generic name is a character string followed by an asterisk (*), for example ABC*, and it selects all proces

The process name is always returned regardless of the attributes requested.

The maximum length of the string is MQ_PROCESS_NAME_LENGTH.

Optional parameters

CommandScope (MQCFST)
Command scope (parameter identifier: MQCACF_COMMAND_SCOPE). This parameter applies to z/OS only.

Specifies how the command is executed when the queue manager is a member of a queue-sharing group. You can specify one of the following:

- blank (or omit the parameter altogether). The command is executed on the queue manager on which it was entered.
- a queue manager name. The command is executed on the queue manager you specify, providing it is active within the queue sharing g sharing group environment, and the command server must be enabled.
- an asterisk (*). The command is executed on the local queue manager and is also passed to every active queue manager in the queue-

The maximum length is MQ_QSG_NAME_LENGTH.

You cannot use *CommandScope* as a parameter to filter on.

IntegerFilterCommand (MQCFIF)
Integer filter command descriptor. The parameter identifier must be any integer type parameter allowed in *ProcessAttrs* except MQIACF_ALL. Us for information about using this filter condition.

If you specify an integer filter, you cannot also specify a string filter using the *StringFilterCommand* parameter.

ProcessAttrs (MQCFIL)
Process attributes (parameter identifier MQIACF_PROCESS_ATTRS).

The attribute list might specify the following value on its own - default value used if the parameter is not specified:
MQIACF_ALL
All attributes.

or a combination of the following:
MQCA_ALTERATION_DATE
The date at which the information was last altered.
MQCA_ALTERATION_TIME
The time at which the information was last altered.
MQCA_APPL_ID
Application identifier.
MQCA_ENV_DATA

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
  - Configuration 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

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 – 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

MQ Appliance – System and qmgr monitors

- The MQ Appliance also has a new style of event generation
  - 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

PROGRAMMING PCF IN C

Sending a command message

- Use command reference to decide which command and parameters 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
- 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

Some code

```
/*****************************************************************/
/* 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

```
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

- 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

- Such structures contain length of the data AND length of the structure
- 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;
```

My preferred version

```
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

```
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

More mistakes – all on one line

```
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

```
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

```
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);
}
```

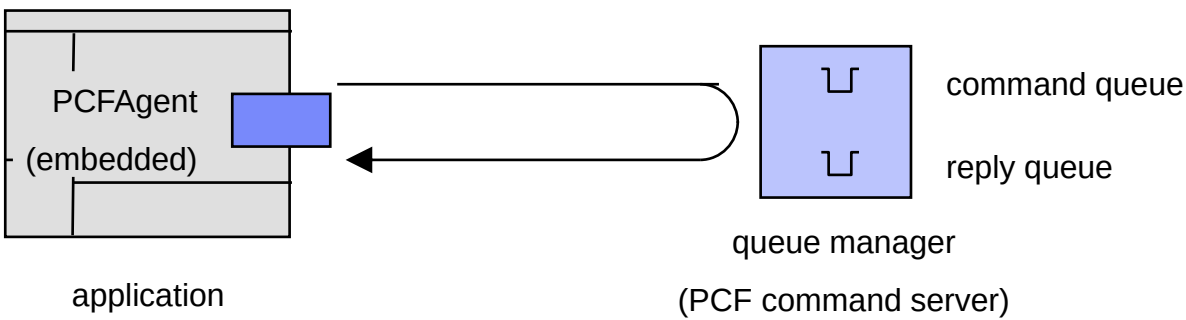
PROGRAMMING PCF IN JAVA

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

```
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.MQCFT_TRACE_ROUTE,
     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;
      …
    }

  }
}
```

Parsing PCF in JMS

```
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);
  ByteArrayInputStream bais = new ByteArrayInputStream(data);
  DataInput dataInput = new DataInputStream(bais);
  PCFMessage pcfMessage = null;

  pcfMessage = new PCFMessage(dataInput);
  …
}
```

Sending a PCF message in JMS

```
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();
}
```

Formatting MQI constants

- 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_AUTHORIZED"
- 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 what filter to apply for which attribute
  - p.getParameter() returns 1
  - Call MQConstants.lookup(1,"MQIA.*") to get "MQIA_APPL_TYPE"
  - But then have to **know** to use filter "MQAT_.*" to get Unix or ZOS
  - 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)

C or Java?

- Often have no choice – language is dictated by larger environment
- From my own programs
  - Eclipse/Explorer plugins forced Java
  - Desire for MQCB forced C
- Java hides many complexities
  - String handling is better
  - Multiple z/OS command responses are automatically combined
  - MQConstants.lookup() equivalent does not exist in C
  - 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 failed

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 appliance-generated events (FP3)
  - amqsailq is a sample using MQAI Bags
  - samples/pcf contains Java programs such as StartChannel, ListQueueNames
- SupportPacs
  - MS0S Java (source at https://github.com/ibm-messaging/mq-mqsc-editor-plugin)
  - MO01 C event formatter
  - MS12 COBOL event formatter

Any questions?