MQGem User Formats

User Guide

13th July 2024

Paul Clarke

MQGem Software Limited
support@mqgem.com

Take Note!

Before using this User's Guide and the product it supports, be sure to read the general information under "Notices".

First Edition, July 2024

This edition applies to Version 9.3.3 of *Live Parsing Editor for IBM MQ Messages*, to Version 9.4.0 of the *IBM MQ Q Utility* and to Version 1.0.1 of *MakeFmt* and to all subsequent releases and modifications until otherwise indicated in new editions.

Notices

Table of Contents

Introduction

Several MQGem products understand how to display MQ message content in the various standard IBM MQ formats such as MQXQH (Transmission header), MQDLH (Dead Letter header) and PCF (Programmable Command Format) as well as a number of industry standards such as XML, EDIFACT and FIX.

However, it is often the case that developers define their own message formats. Naturally our products don't automatically know about these formats but with MQGem User Formats you can provide descriptions of these message formats which then allows MQEdit and the Q Program to display the message in a more meaningful way than just hex. It also allows editing of the message in MQEdit in a much more natural way.

You can also use a User Format definition file to associate a particular formatter with a particular queue or format. For example you can say that a queue, say CUST.DATA, should, by default, use the JSON formatter since messages on that queue are normally in JSON format.

This document fully describes the MQGem User Format and what your user format files should look like in order to allow these programs to use them.

My thanks go to Morag Hughson for considerable assistance with this manual, for numerous usability suggestions, for keeping me sane when nothing ever seems to go right, as well as tirelessly testing many buggy versions of the program!

If you have any questions about MQGem User Formats or perhaps suggestions for additional features then please send an email to support@mqgem.com. Equally if you don't have a licence and would like to try out MQEdit or the Q program then please send an email to the same address.

Chapter 1. User Format Files

A User Format file contains the definitions of the user message formats. It is a simple text file which can be created with any text editor. The file is largely free-format and white-space and comments can be placed almost anywhere it is deemed appropriate. If required this file can include other files so it is not necessary to put all the definitions in a single file and instead the definitions can be organised however is appropriate for your installation. For example, you could have a file per application group.

1.1. Locating the file

Full details of specifying the user format file to MQEdit and the Q program can be found in the "User Format Messages" chapter in the respective product user guide, a brief summary of which is provided here.

MQEdit	By default MQEdit will look for a file called MQEDIT.FMT in the same location as the executable. This can be changed in the User Formats tab in the Preferences dialog where you can specify the path and name of the file.
Q	By default the Q Program will look for a file called mqgem.fmt in the same location as the executable (or in the current directory). This can be changed by setting the environment variable MQGEM_USERFORMAT_FILE to specify the path and name of the file.

1.2. Comments

Any data following a ';' (semicolon) is regarded as a comment. These can be placed anywhere in the file. The comment marks the rest of the line as a comment.

```
; This comment is the entire line
struct Test
{
  int Field; Field comment
}
```

In most programming languages the comment is completely ignored by the parser. However, in MQGem User Formats some comments are treated as part of the data. These are:

- If a comment immediately follows the structure name
- If a comment immediately follows a field definition
- If a comment immediately follows an enumeration definition

1.3. Including other files

The definition file may, at any time, include additional definition files provided the keyword 'include' is at the top level. For example, you can not use **include** in the middle of a structure or enumeration definition. The included file will be parsed just as though the text were entered in the original file. The syntax is as follows:

```
include "mydefinitions.fmt"
```

The file can be fully qualified with the full path if required. If the full path is not specified then the path of the current file is assumed.

There are many reasons why organising the definitions into different files may be useful but perhaps one of the major reasons is that the definitions may not all be coming from the same person. For example, consider a company which has, say, three application groups. We'll call them X,Y and Z. Each of these application groups is responsible the definition and structure of their MQ messages. One might imagine that they are also responsible for producing an User Format file containing the definitions suitable for browsing and editing the messages. So, we could imagine that the MQ administrator may get three files:

- **MsgFormats_X.fmt**
- **MsgFormats_Y.fmt**
- **MsgFormats_Z.fmt**

Clearly the administrator could try to merge these definitions into a single file. However, this is time consuming and error prone. Instead it makes more sense to define a single fixed file and include the application group files.

One further reason why having each structure in a different file is the helper program *MakeFmt. MakeFmt* is free program supplied by MQGem Software that will convert a COBOL cobybook into a user format definition. This can make it make much easier and quicker to generate a representation of the COBOL structure in the User Format syntax.

For example, the content of the User Format file that MQEdit or the Q program is referencing might be:

```
; Include application group definitions
include "MsgFormats_X.fmt"
include "MsgFormats_Y.fmt"
include "MsgFormats_Z.fmt"

; Additional common definitions
struct XXX
{
  ........
}
```

Now when an application group produces a new set of definitions the administrator merely needs to copy the new file over the top of the old one.

1.4. Refreshing the User Format File

To use the definitions it is not even necessary to end the MQEdit. Just go in to the User Formats tab in the Preferences Dialog and press the “Reload User Formats” button.

Since the Q program is not a long running program, to pick up new contents of the User Formats file, just run the Q program again.

1.5. File Syntax

There are essentially three aspects to the definitions.

1. **Structure Definitions**
   These definitions define the data structures and the fields that are contained in them.
2. **Enumerations**
   These definitions allow you to specify special values for a field and given them a name.
3. **Structure Association**
   These definitions allow you to associate either an MQ format or MQ Queue name (or both) with a particular structure definition.

We will discuss these in turn.

Chapter 2. Structures

These are the definitions which actually layout the format of the user data. Each structure definition must have a unique name. Structures contain one or more field definitions, and can also contain other structure definitions.

The general syntax of the definition is:

```
struct <Name> [ ; Optional Description ]
{
  ; As many field definitions as required
  <Type> [<Type Format>]
         [<Type Name>]
         [<Field Name>]
         ['[' <array value> ']']
         ['(' <enumeration> ')']
         ['=' <conditional>] [':' <option list> ] ';'
  . . . .
}
```

At first glance this may appear complicated but it is actually pretty simple. The general idea is that the definition looks very similar to the syntax we might find in any number of programming languages so it should be fairly natural to most technically minded people. Let's look at the simple example:

```
struct CUSTOMER; Customer Record
{
  char ID[4]      ; Eyecatcher
  int  Version;
  char CusName[50]; Customer name
  hex  AuthToken[16];
}
```

Here we can see a simple structure containing just a few fields looking a little like a 'C' structure definition. However, there are a few key differences. Firstly the field types are different than those defined in 'C' and far more varied. For a complete list of the field types please see 'Field Types' below. Another key difference is syntax and usage of comments. As mentioned in the ' Comments' section a comment can be considered as data. For example the comment “Customer Record” will be used as the description of the structure in the message display. If this comment is not present then the structure name, CUSTOMER, will be used as the structure description. In a similar way the comments, “Eyecatcher” and “Customer Name” will also be taken as the values which should be displayed for the field. The key difference for a field comment is that only the first 14 characters will be used.

So, if we used this structure definition and display a message we will get output like the following:

```
[   74 bytes] Customer Record
Eyecatcher   :'CUST'
Version      :1
Customer name:'MQGem Software Limited                            '
AuthToken    :01234567890123456789012345678901
```

We can see the defined structure is effectively overlayed on to the message data. One of the other aspects that this example demonstrates is one of the uses of arrays. Here we are using arrays on the ID, CusName and AuthToken fields to specify that the field consists of a fixed number of the primitive element types. So, for example, CusName is a fixed string of 50 characters. For a complete description of arrays please see 'Field Arrays' below.

2.1. Structure Descriptions

In addition to the field definitions you can also request description lines be output in the structures. This is done by using a type value of '**desc**'.

The syntax for a description line is as follows:

```
struct <Name> [ ';' < Description Text > ]
{
  desc [':' <option list> ] ';' < Description Text >
}
```

A structure definition can have as many description lines as you wish. The description text itself is optional and so these lines can be used to just space out the fields.

The first blank character of the description text, if present is ignored. So, if you wish to actually indent the description in the output structure by 4 spaces you must include 5 spaces in the definition.

If we added a description line to the structure above we could do it with this definition:

```
struct CUSTOMER; Customer Record
{
  char ID[4]      ; Eyecatcher
  int  Version;
  desc ; Identification Fields
  char CusName[50]; Customer name
  hex  AuthToken[16];
}
```

This would yield an output such as:

```
[   74 bytes] Customer Record
Eyecatcher   :'CUST'
Version      :1
Identification Fields
Customer name:'MQGem Software Limited                            '
AuthToken    :01234567890123456789012345678901
```

Clearly many of the field options are not applicable to descriptions lines. However, you can use **low**, **medium** and **high** to control when the description line is displayed.

2.2. Field Types

MQGem User Formats support many field types, far more than you might expect to find in a normal programming language. This allows you to control how the data is overlaid into the structure and how the data is displayed to the user. The full list of data types is given below:

Data Type	Size	Description	Example Output
align16	Variable	Align next field to 16 byte boundary	None
align2	Variable	Align next field to 2 byte boundary	None
align4	Variable	Align next field to 4 byte boundary	None
align8	Variable	Align next field to 8 byte boundary	None
char	1 byte	Character in current codepage	`Field Value  :'X'`
charbp	n bytes	Blank padded character in current codepage	`Field Value  :'X   '`
desc	0	Prints out text line in structure	`This can describe the fields in the structure`
digits(n) **digits(m.n)**	Variable	Signed Decimal character fields For example: **digits(5)** **digits(3.2)** See 'Signed Decimal Character Fields' for more information.	`Field Value  :-12345` `Field Value  :+123.45`
double	8 bytes	8 byte floating point number See 'Floating Point Numbers' for more information	`Field Value  :23.356`
flags	4 bytes	Integer which represents a set of flags	`Field Value  :00000103` `00000001 Flag 1` `00000002 Flag 2` `00000100 Flag 8`
flags16	2 bytes	16bit integer which represents a set of flags	`Field Value  :03` `01 Flag 1` `02 Flag 2`
flags8	1 byte	8bit integer which represents a set of flags	`Field Value  :0103` `0001 Flag 1` `0002 Flag 2` `0100 Flag 8`
float	4 bytes	4 byte floating point number See 'Floating Point Numbers' for more information	`Field Value  :23.356`
hex	1 byte	Hexadecimal byte	`Field Value  :7C`
int	4 bytes	32bit signed integer	`Field Value  :37`
int64	8 bytes	64bit signed integer	`Field Value  :2323423423423423423`
int64h	8 bytes	64bit signed integer, with following hex	`Field Value  :2323423423423423423 [0x'203E754B27F9A3BF']`
int8	1 byte	8bit signed integer	`Field Value  :37`
int8h	1 byte	8bit signed integer, with following hex	`Field Value  :37 [0x'25']`
inth	4 bytes	32bit signed integer, with following hex	`Field Value  :37 [0x'25']`

opunch(n)
opunch(m.n)
	Variable	Over-punched signed Decimal
		For example:
		opunch(5)
		opunch(3.2)
		See 'Over-punched Signed Decimal Fields' for more information.
			`Field Value  :-12345`
			`Field Value  :+123.45`

packed(*n*)	Variable	Signed Packed Decimal
		For example:
		packed(6)
		packed(5.3)
		See Packed Decimal Fields for more information.
			`Field Value  :-123`
			`Field Value  :12.340`

short	2 bytes	16bit signed integer	`Field Value  :37`
shorth	2 bytes	16bit signed integer, with following hex	`Field Value  :37 [0x'25']`
struct	Any	Embedded structure	As per structure content
time32	4 bytes	4 byte time value Number of second since 00:00:00 January 1st 1970	`Field Value  :14:23:17 05/08/1997 (Local)`
time64	8 bytes	8 byte time value Number of second since 00:00:00 January 1st 1970	`Field Value  :14:23:17 05/08/1997 (Local)`
uint	4 bytes	32bit unsigned integer	`Field Value  :37`
uint64	8 bytes	64bit unsigned integer	`Field Value  :2323423423423423423`
uint64h	8 bytes	64bit unsigned integer, with following hex	`Field Value  :2323423423423423423 [0x'203E754B27F9A3BF']`
uint8	1 byte	8bit unsigned integer	`Field Value  :37`
uint8h	1 byte	8bit unsigned integer, with following hex	`Field Value  :37 [0x'25']`
uinth	4 bytes	32bit unsigned integer, with following hex	`Field Value  :37 [0x'25']`

upacked(*n*)	Variable	Unsigned Packed Decimal
		For example:
		upacked(6)
		upacked(5.3)
		See Packed Decimal Fields for more information.
			`Field Value  :123`
			`Field Value  :12.340`

ushort	2 bytes	16bit unsigned integer	`Field Value  :37`
ushorth	2 bytes	16bit unsigned integer, with following hex	`Field Value  :37 [0x'25']`

2.2.1. Over-punched Signed Decimal Fields

Over-punched Signed Decimal fields are numeric fields where each digit of the number is stored in a single character. The physical size of such a field depends on how many digits it needs to support. The leading or trailing byte (depending on the definition) will contain some bits that represent the sign. For full details of the format of Over-punched ASCII fields please read Appendix A: Over-punched ASCII format on page 22.

The definition of this field can take two forms and both forms have some qualifiers defined below.

1. **opunch(a,*qualifier*) – an integer value**
2. **opunch(a.b,*qualifier*) – a real number value**

where:-

- **a** gives the number of digits before the decimal point or number of digits in an integer.
- **b** gives the number of digits after the decimal point.
- ***qualifier*** can take one (or more) of the values in the table in 2.3. Type Format on page 9. Multiple values are separated with the '|' sign.

For example, opunch(5) would allow numbers in the range -99999 to 99999 and opunch(3.2) would allow numbers such as +123.40

2.2.2. Packed Decimal Fields

Packed Decimal Fields are numeric fields popularized, and used heavily, by the COBOL programming language. Packed Decimal fields are numeric fields where each digit of the number is stored in a single nibble. So, unlike a data type, such as int, where the physical size of the bytes is fixed the size of a packed field depends on how many digits it needs to support[1]. The definition of the packed or unpacked field can take two forms.

1. **packed(a) – an integer value**
   where **a** gives the number of digits to support.
   For example, packed(5) would allow numbers in the range -99999 to 99999.

   The editor will not display leading zeros of the number.
2. **packed(a.b) – a real number value**
   where **a** gives the number of digits before the decimal point
   where **b** gives the number of digits after the decimal point
   For example, packed(3.2) would allow numbers such as 123.40

   The editor will not show leading zeros but it will show trailing zeros as above.

2.2.3. Signed Decimal Character Fields

Signed Decimal Character fields are numeric fields where each digit of the umber is stored in a single character. The physical size of such a field depends on how many digits it needs to support and additionally it has one separate character to store the sign symbol (a '+' plus or a '-' minus sign). The definition of this field can take two forms and both forms have some qualifiers defined below.

1. **digits(a,*qualifier*) – an integer value**
2. **digits(a.b,*qualifier*) – a real number value**

where:-

- **a** gives the number of digits before the decimal point or number of digits in an integer.
- **b** gives the number of digits after the decimal point.
- ***qualifier*** can take one (or more) of the values in the table in 2.3. Type Format on page 9. Multiple values are separated with the '|' sign.

For example, digits(5) would allow numbers in the range -99999 to +99999 and digits(3.2) would allow numbers such as +123.40

[1]COBOL is heavily used in back-end systems such as z/OS running on s390 hardware and most information seems to pertain to that system. So much so that we could not find reliable information about how packed decimal fields are stored on, say, an Intel machine. As such User Formats will always treat packed decimal fields as big endian format. If anyone can provide us with the nibble layout of a packed decimal field on other platforms then we'd be happy to also support those.

2.2.4. COBOL equivalents

If you are using packed fields in your messages then there is a good chance that you are also using COBOL. The following table shows COBOL data types and their equivalent User Format type.

COBOL Data Type	MQEdit User Format Data Type
`PIC S9(n) COMP-3`	`packed(n)`
`PIC 9(n) COMP-3`	`upacked(n)`
`PIC S9(n)V99 COMP-3`	`packed(n.2)`
`PIC S9(n)V9(m) COMP-3`	`packed(n.m)`
`PIC S9V99 COMP-3`	`packed(1.2)`
`PIC S9(n) SIGN LEADING`	`opunch(n, signleft)`
`PIC S9(n) SIGN TRAILING`	`opunch(n, signright)`
`PIC S9(n) SIGN LEADING SEPARATE`	`digits(n, signleft)`
`PIC S9(n) SIGN TRAILING SEPARATE`	`digits(n, signright)`
`PIC S9(n)V99`	`opunch(n.2)`
`PIC S9(n)V9(m)`	`opunch(n.m)`
`PIC S9V99 SIGN TRAILING SEPARATE`	`digits(1.2)`

2.3. Type Format

This field can modify the way in which a particular field type is processed and displayed. It is only applicable to certain types of fields.

The definitions are:

Data Type	Type Format	Description	Example Definition
double	<Precision>	How many significant digits are output. Default:16	`double(13) Value;`
float	<Precision>	How many significant digits are output. Default: 7	`float(5) Value;`
time32	Comma separated list of options		
	UTC	Display time in UTC rather than Local time	`time32(UTC)           StartTime;`
	MMDDYYYY	Display date in MM/DD/YYYY rather than the default DD/MM/YYYY	`time32(UTC,MMDDYYYY) StartTime;`
time64	Comma separated list of options		
	UTC	Display time in UTC rather than Local time	`time32(UTC)           StartTime;`
	MMDDYYYY	Display date in MM/DD/YYYY rather than the default DD/MM/YYYY	`time32(UTC,MMDDYYYY) StartTime;`

char
packed
upacked
	Please see Field Description for a list of possible values.	This feature allows you to provide information about the content of the field which is then conveyed to the user. For example you can indicate that this is a date field. The user would then be presented with additional information to allow them to more easily understand the field value.	`char(YYMMDD)          Birthday[6];`
			`packed(8,DDMMYYYY)    OrderDate;`

packed
upacked
opunch
digits
	lead	Display leading zeros for the number	`packed(8,lead,DDMMYYYY)  OrderDate;`
			`opunch(3.2,lead)         Price;`
			`digits(5,lead)           Total;`

opunch
digits
	signleft	For an `opunch` type this means the number contains an over-punched sign on the LHS.
		For a `digits` type this means the number uses a separate extra character to store the sign on the LHS.
			`opunch(3.2,signleft)   Price;`
			`digits(5,signleft)     Total;`

opunch
digits
	signright	For an `opunch` type this means the number contains an over-punched sign on the RHS.
		For a `digits` type this means the number uses a separate extra character to store the sign on the RHS.
			`opunch(3.2,signright)   Price;`
			`digits(5,signright)     Total;`

opunch
digits
	plus	Display the plus sign when the number is positive (the minus sign is always shown when the number is negative).	`opunch(3.2,plus)       Price;`
			`digits(5,plus)         Total;`

2.4. Field Description

There are times where just looking at a field value may not be sufficient to easily interpret the data. Consider a date field where the value is '010203'. Does this mean 1st February 2003 or perhaps 3rd February 2001. Our American friends may see it as 2nd January 2003.

What we need is some way to indicate that the field is actually DDMMYY and this is where a field description comes in. For character and packed fields you can optionally follow the type with a description string. A description string can consist of multiple elements, each one describing a character or digit of the field value.

Element	Meaning
DD	The two digit day of month (1-31)
hh	Hours
JJJ	Julian Day – day number from 1st January
mm	Minutes
MM	Month Number (1-12)
ss	Seconds
yy	A two digit year which is not interpreted – displayed as '84
YY	A two digit year where values less than 50 are considered 20YY. Values greater than 50 are 19YY For example 34 would be displayed as year 2034.
YYYY	A four digit year

The field value and its description will be used to interpret the data and display the realised string on the right-hand side of the field. So, let's look at a couple of simple example field definitions. Suppose we had this set of definitions:

```
struct datestruct
{
  char                  Name[50];
  packed(6,DDMMYY)      BirthDate;
  packed(5,YYJJJ)       JoinDate;
  char(YYYY/MM/DD)      Renewal[10];
}
queue DATES
{
  struct datestruct;
}
```

Clearly this is just an example and one would hope that no software designer wishing to show his face in polite society would ever define a structure with three date fields all of different formats, But here we are....this is the level of education we have today!

Anyway, such a structure, if ever designed would be displayed as the following:

```
[   67 bytes] datestruct
Name         :'A. N. Other                                  '
BirthDate    :10414                 1st April 2014
JoinDate     :32305                 31st October 2032
Renewal      :'2033/10/31'          31st October 2033
End of structure 'datestruct' (67 bytes)
```

There are a couple of things worth noting.

Firstly, note that the convention of not showing leading zeroes on a numerical field leads to a slightly odd look for the actual data field. We have a six digit date displayed as a five digit number. You can fix this using the 'lead' type format. However, the description on the right helps us confirm that the data entry is correct.

Secondly, note that Julian dates are very difficult to read on their own. The JoinDate value of ***32305*** is not at all obvious as 31st October unless you have a mind like a steel trap.

Now, suppose the value of the fields were incorrect? Well, we'd see something like this:

```
[   67 bytes] datestruct
Name          :'A. N. Other                                       '
BirthDate     :11414                  Invalid
JoinDate      :32375                  Invalid
Renewal       :'2033-10-31'           Invalid
End of structure 'datestruct' (67 bytes)
```

See if you can determine why each of these fields are now invalid. Displaying 'Invalid' can be a very useful way of indicating to the user that some of the message contains incorrect data.

2.5. Type Name

Type Name is only applicable for **struct** types.

```
struct MyStruct
```

The name is mandatory and gives the name of the structure definition that should be used for this field. This structure definition must already be defined. The structure name can optionally be followed by a field name.

2.6. Field Name

The name of the actual field. This field is mandatory for all field types except for the following types:

Data Type	Required
align2 **align4** **align8** **align16**	Should not be specified
struct	Can be optionally specified
desc	Should not be specified

2.7. Field Arrays

Field arrays are a way of showing that a particular field repeats. There are two types of array.

2.7.1. Fixed Arrays

A fixed array, as it name suggests, has a fixed size. This is often used for character or hex fields but can be used for any data type. We have already seen arrays used in this way:

```
char CusName[50]; Customer name
```

Here we have defined that the customer name field is a fixed size of 50 characters.

However, suppose we want an array of names. Well, we can just add another array dimension.

```
char CusNames[50][10]; Customers names
```

This definition defines a 10 element array of 50 character names.

In a similar fashion we can define hex fields.

```
hex AuthenticationCode[32];
hex AuthenticationCodeArray[32][5];
```

2.7.2. Variable Arrays

A variable array is an array which can change size as the message content itself changes. This concept is only applicable to **char, hex** and **struct** type fields. There are essentially two forms. Firstly where the size of the array is given by the value of another field. For example, the following:

```
int  CusNameLen         ; Name Length
char CusName[CusNameLen]; Customer name
```

In this case the length of the customer name field is given by the **CusNameLen** field. Or you could do the following:

```
int    NumCusRecords            ; Number of records
struct CusRecords[NumCusRecords]; Customer name
```

where the number of following structures is given by a numeric field. Of course because you can have variable numbers of structures you can effectively have variable numbers of any field, it just depends on the definition of the structure.

Using variable length fields will mean that the offset of any fields following the variable array will change as its length changes. You may want to use a field option such as 'align4' in order to align the following fields. Alternatively you can defined align fields into the structure definition to ensure that following fields are kept on an appropriate boundary.

The second form of variable array applies only to **char** and **hex** type fields which support a special value of '*' which can be used to indicate that the field applies to the rest of the message. Clearly a field using this syntax should be the last field defined in the structure and message.

For example:

```
char EyeCatcher[4];
int  Version;
char Details[*];
```

Here the **Details** field starts at offset 8 into the message and it's length is the remainder of the message.

2.8. Conditional Values

Conditional Values are a way of making the structure selection based on the content of the data. In other words a structure definition will only apply if all the conditional values match. Consider the example:

```
struct CUSTOMER; Customer Record
{
  char ID[4]      ; Eyecatcher
  int  Version;
  int  Type = 1;
  char CusName[50]; Customer name
  hex  AuthToken[16];
}
```

Here we see that a conditional value has been put on the **Type** field. If the underlying data does not have the value '1' in this position then this structure will not apply. In this way you can define multiple structure definitions and let the conditional values drive which structure is in operation.

Conditional values can only be placed on numeric fields, fixed string arrays and fixed hexadecimal arrays.

2.8.1. Conditional Strings

```
struct CUSTOMER; Customer Record
{
  char ID[4] = “CUST”; Eyecatcher
  int  Version;
  int  Type = 1;
  char CusName[50]; Customer name
  hex  AuthToken[16];
}
```

Here we are putting the conditional value on the character array. The conditional string can not be longer than the array field. If it is shorter then only the supplied characters are matched. Note that only characters that apply in both the source and target codepage can be used.

Note that this example has specified two conditional values. Both values must match in order for the structure to be considered a match.

2.8.2. Conditional Hex

```
struct CUSTOMER; Customer Record
{
  char ID[4]      ; Eyecatcher
  int  Version;
  hex  Type[4] = “AE45CFED”;
  char CusName[50]; Customer name
  hex  AuthToken[16];
}
```

Here we are putting the conditional value on the hexadecimal array. The conditional string can not be longer than the array field. If it is shorter then only the supplied values are matched.

2.9. Field Options

Each field can optionally have a list of options values which control the behaviour of the field and its output.

The list of options and their meanings are as follows:

Option	Meaning
align4	Always assign storage to this field in multiples of 4 bytes
ccsid	This field will set the codepage value for the following chained structure
defer	Editing of this field is deferred. The messages byes are not changed until the user presses <ENTER>
enc	This field will set the encoding value for the following chained structure
expert	This field can only be modified in expert mode.
high	This field is always displayed
limval	Limited Values This option is used in conjunction with enumerations. It shows that this field only has a limited number of defined values and that any unrecognised values should be ignored. If this option is not specified then **(Unrecognised)** will be displayed next to to the field
low	This field should be displayed in low detail mode and above This is the default if no detail level option is specified
medium	This field should be displayed in medium detail mode and above
struclen	Identifies the field as the one containing the length of the structure
summary	Identifies this field as one that should be shown in the message list pane.

Chapter 3. Enumerations

Enumerations are useful for a number of reasons.

1. To display a text string description for a 'known' field value
2. When editing messages:-
   a) To list the bit values which should be displayed for the flag fields.
   b) To allow simple data entry of a 'known' value just by double clicking on the value.
   c) To display the list of possible values to the user.

There are actually two types of enumeration definitions and which one to use depends on the type of field.

3.1. Integer Enumeration

This type of enumeration must be used for any type of field which stores an integer number in binary form. In other words the following field types **int**, **short**, **flags**, **uinth** etc.

The format of the enumeration is the same whether you are displaying a single version or a set of flags. The key difference is that when an enumeration is used for a flag field then the enumeration values should be powers of 2. Let's look at a simple numeric enumeration.

```
enum_int Values
{
  1,  “VAL1” ; First Value
  2,  “VAL2” ;
  3,  “VAL3” ; Third Value
  4,  “VAL4” ; Fourth Value
}
```

Each enumeration value consists of three parts. The actual value constant, the constant name and the constant description. Specifying a constant description is optional. So, while the ';' (semicolon) at the end of each value is mandatory the text following the ';' is optional. If no description is specified then the description is assigned the same value as the constant name.

This enumeration can be used in a structure with the following definition:

```
struct EXAMPLE ; Numeric Enumeration Example
{
  int  Field(Values); Main field
}
```

The enumeration must be defined before it is used.

An example of a flag enumeration would be:

```
enum_int ValueFlags
{
  0x00000001,  “Flag1” ; Flag One
  0x00000002,  “Flag2” ; Flag Two
  0x00000004,  “Flag3” ; Flag Three
  0x00000008,  “Flag4” ; Flag Four
}

struct EXAMPLE ; Flag Enumeration Example
{
  flags Field(ValueFlags); Main field
}
```

Any flags in the **Field** value which are not catered for by the enumeration flags will be output as a single additional value.

3.2. Character Enumeration

The second type of enumerations is called character because the 'value' of the enumeration is given by a character value rather than just an integer number. However, this type of enumeration is still used for some numeric fields. The syntax and a use of a character enumeration is very similar to an integer enumeration.

```
enum Countries
{
  "AU",  "Australia";
  "CA",  "Canada";
  "DE",  "Germany";
  "DK",  "Denmark";
  "FR",  "France";
  "NL",  "Netherlands";
  "NZ",  "New Zealand";
  "SE",  "Sweden";
  "UK",  "United Kingdom";
  "US",  "United States";
  "ZA",  "South Africa";
}

struct EXAMPLE ; Character Enumeration Example
{
  char CountryCode[2](Countries);
}
```

In the example above we can see a two character field containing a country code which 'points' to a character enumeration containing the country codes and their text descriptions[2].

Character enumeration isn't only used for character fields however. They are also used for **packed**, **opunch**, **digits**, **float** and **double** fields.

```
enum InterestingNumbers
{
  "0",    "Zero";
  "0.58", "Euler's constant";
  "1.62", "Golden Ratio";
  "2.72", "e – natural log base";
  "3",    "First odd prime";
  "3.14", "pi";
  "13",   "Unlucky for some";
  "27",   "First odd perfect cube";
  "51",   "First uninteresting number!";
  "69",   "Square and Cube give complete set of digits";
}

struct EXAMPLE ; Character Enumeration Example
{
  float Value(InterestingNumbers);
}
```

When using an enumeration for numeric fields the value does not need to match exactly – for example a value of “1.00” would match “1”. As a general rule any trailing or leading numbers are ignored in the comparison. As a consequence it is entirely valid to just specify an integer value for double or packed(7.2) fields.

[2]Apologies to citizens of the hundreds of countries which are not represented here. Clearly in a real implementation the list of countries would be far larger and undoubtedly include yours. This subset was chosen purely for the sake of brevity and because it represents the majority of our customers.

Chapter 4. Structure Association

In order for a defined structure to be used, you need to associate the structure with a particular Queue or Format. Traditionally it is the MQMD FORMAT field which tells MQ what the structure of the MQ message is. So, in an ideal world, you would have a unique FORMAT value for each of your structures. However, this approach has one major drawback. That is that in order to do data conversion you would need data conversion exits for each of your structures.

It is possible, if you are just using character data in your messages, that a format of MQSTR would be sufficient and the context determines how the message should be treated. For example, you could say that all messages on queue 'CUSTOMER' contain only customer record messages.

For this reason you can associate defined structures to particular format values or particular queue names, or both. The syntax is as follows:

```
association list   := format association | queue association | structure list
formatter name     := AUTO | TEXT | JSON | FIX | CSV | XML | EDIFACT

format association := format <format list> [ ';' < Description Text > ]
                      {
                        [ override section ]
                        [ formatter <formatter name>; ]
                        { association list }
                      }

queue association  := queue <queue list>
                      {
                        [ formatter <formatter name>; ]
                        { association list }
                      }

structure list     := <structure list> ;
```

Note that the definition is recursive, for example it is possible to include a queue association inside a queue association.

The format and queue lists can contain the standard wildcards * and ?. The format list also supports the special value **$none** which equates to an empty format field (MQFMT_NONE).

So, an example definition could be:

```
format CUS*,CST* ; Customer Record
{
  struct CUSTOMER;
}

queue CUSTOMER
{
  format $none, MQSTR
  {
    struct CUSTOMER;
  }
}
```

These two simple definitions state that any format beginning with the characters 'CUS' or 'CST' should display the CUSTOMER structure. Equally any message with formats of either MQSTR or NONE on the queue called CUSTOMER should also display the messages as the CUSTOMER structure.

The description “Customer Record” following the ';' on the format line will be displayed to the right of the format field e.g in the message descriptor.

One word of warning about the syntax. The semicolon essentially serves two purposes. It marks the end of the structure list but it also marks the start of a comment. Any characters beyond it on the same line will be ignored. This means that if you try to compress the definitions into a single line, such as the following, it **WILL NOT** work:

```
format CUS*,CST* { struct CUSTOMER; }
```

This is because the terminating '}' bracket will not be seen since it is part of the comment. You should keep your definitions on separate lines, this ensures it is syntactically correct and is also more readable.

4.1. Formatter association

The other form of association you can make is to associate a formatter with a queue or format name. For example, suppose we have a queue, CUST.DATA, that we know always contains messages which are in JSON format. It would make sense to associate that queue with that formatter rather than constantly having to specify it. We can do that simply by the following lines in the user format file.

```
queue CUST.DATA
{
  formatter JSON;
}
```

Chapter 5. Override Section

In an ideal world all MQ messages would contain all the correct identifying information regarding their content. Unfortunately, for a variety of reasons, that is not always be the case. We have seen MQ applications which have not used the correct format, codepage or encoding values. This means that programs like ours may be looking at these fields in the Message Descriptor and making incorrect choices.

Of course, we would always recommend that MQ users go back and fix the applications putting these messages. Sometimes though, that is either not feasible or not possible. What we need is a way to specify that the values in the actual message should be ignored and to display the message as though a different set of values were in force. This is where the override section comes in. Let's look at an example:

```
queue Q1
{
  format *
  {
    override
    {
      CodedCharSetId = 850;
      Encoding       = 0x222;
      Format         = "MQSTR";
    }
  }
}
```

This definition states that every message on queue 'Q1', regardless of the given format value, should be treated as though it is a string format message in codepage 850 and with Intel-reverse encoding. Each of the override values are optional so you can override just one or two of the values if required.

If an override is in force then this will be shown when it the message is displayed. For example:

```
Feedback     :0 (None)
MQEncoding   :0x'111' (Normal)          Override(0x'222')
CCSID        :500 (UK EBCDIC)           Override(850)
Format       :'        ' (None)         Override('MQSTR   ')
Priority     :0 (Lowest)
Persistence  :0 (Not Persistent)
```

Here you can see that the actual message values are shown, and in MQEdit it allows you to edit them, but it also tells you on the right-hand side that it is displaying the messages as though a different set of values were set.

Chapter 6. Floating Point Numbers

Floating point numbers present a lot of challenges for an editor. The main problem is that floating point numbers are inexact and the inexactness is not predictable. Floating point numbers encode an enormous range of numbers in a relatively small number of bytes. As such there are 'gaps'. There are some numbers that just can't be stored exactly. One of the factors which can affect this is the precision by which the number is printed out. For example, by default for **float** types the editor uses a precision of 7. This means that the editor will try to output 7 significant digits for any **float** variable.

For example, consider the number 1.2. In a float variable this is actually stored as 1.20000005. Similarly 1.224 is stored as 1.22399998. As you can see these numbers are very close approximations to the actual numbers but, none the less, are different. If the number is just displayed in a report then the different is unlikely to matter. However, in an editor it very definitely does matter. If you type the string 1.224 and 1.22399998 is displayed you would have a very strange editing experience. Consequently the editor tries to mask these differences by detecting and correcting these rounding errors. However, it means that if you enter a number which looks like a rounding error then the editor will, incorrectly, correct it.

If you find that the editor will not accept a particular floating point number then you can try entering a number which is close.

6.1. Precision

One of the key aspects of floating point numbers is the notion of precision. When a number is output you can say how many significant digits you want to display. However, bear in mind that floating point variable can only support a certain number of significant digits because of the limited number of bits the number is stored in. Again it is not an exact science but roughly speaking a **float** can support up to 7 significant digits and a **double** can support up to 16. Float variables, by virtue of their smaller size, are far more prone to rounding errors. You can control how many significant digits you would like displayed by following the data type with the number of digits you want. For example:

```
float(4)   number1;
double(10) number2;
```

If no precision is specified then the logical maximum values of 7 and 16 respectively are used.

6.2. Display and Entry Format

Not only can floating point numbers store a number with great accuracy but they can also store very large or very small numbers using the exponential form. The exponential form is a fixed format:

```
<number>'e' [ '+' | '-' ] <3-digit exponent>
```

So, the following are all valid examples of floating point numbers using the exponential format:

```
1.2e+022
-4.567e+010
1.345e-024
```

Leading and trailing zeroes are not allowed. The choice about whether normal format or exponential format is used is based on the number itself. The exponential format should be used if the number is:

- equal or greater than 10000000000 (10^{10})
- equal or less than -10000000000 (10^{10})
- smaller than 0.0001 (10^{-4})

6.3. Specification

Floating point numbers (single and double precision) in both IEEE 754 and IBM S/390 Hexadecimal floating point (HFP) formats an be displayed (and edited). The Message Descriptor Encoding field (or Encoding field in any chained headers) is used to determine how any floating point numbers it comes across should be treated:-

- MQENC_FLOAT_IEEE_NORMAL
- MQENC_FLOAT_IEEE_REVERSED
- MQENC_FLOAT_S390

If your message does not correctly describe the floating point format that is in use in the message, see the 'Override Section' description on page 19.

Chapter 7. Developing your User Format Definitions

It is fairly easy to develop your own User Formats.

The suggested process is as follows:

1. **Decide on the message you wish to define a format for and display it.**
   Presumably, for the moment at least it is displayed either as just text or as hex.

2. **Decide where you want to keep your user format file.**

   a) **If you are using MQEdit:**
   Bring up the "User Formats" tab of the Preferences Dialog. Leave this dialog open while you work.

   b) **If you are using the Q program:**
   Set the environment variable MQGEM_USERFORMAT_FILE to point to the user format file.

3. **Display the message with offsets.**
   Displaying offsets is really useful since the editor will show you the offsets of each field in the structure as you define your message format. You can then verify that the offset is indeed what you would expect for the given data types. Offsets can be displayed either in hex or in decimal.

   a) **If you are using MQEdit:**
   This is most easily done by pressing the (+0) tool in the toolbar. Choose hex or decimal in the "General" tab of the Preferences Dialog.

   b) **If you are using the Q program:**
   This can be done by adding the -dt (decimal offsets) or -dT (hexadecimal offsets) to your Q invocation.

4. **Next start a text editor and edit the format definition file**

5. **To start with just define the first field of the structure you want to display.**
   If you have a COBOL copybook then you can get a huge head-start with your strcuture definition by using the *MakeFmt* helper program.

   For example:-

```
struct CUST;
{
  char  CustName[50]; Customer name
}
```

6. **Add an appropriate "Structure Association"**
   This sets this structure definition to be used for this MQ format or Queue. For example, to use a format of "CUSTOMER" we might specify the following:

```
format CUSTOMER
{
  struct CUST;
}
```

7. **If necessary add conditional values to the structure definition.**
   Ideally, conditional values are added at the end of the process, once all fields are defined but it may be necessary to define them now to differentiate this structure from other structures.

8. **Save the file in the text editor**

9. **If you're using MQEdit, press the "Reload User Formats" button in the Preferences Dialog.**

10. **An error message will be displayed if parsing the structure format file encounters any errors.**
    Continue to fix any syntax errors and refresh the display, until parsing is successful.

11. **Now add the definition of the next field of the structure**
    If you don't have any more fields to add then you are done! If you do add a new definition then continue to step 8.

Appendix A: Over-punched ASCII format

While the format for over-punched numbers is well-defined in EBCDIC there are actually two possible formats that could be used for data that is in ASCII. Which format is used depends on the compiler in question.

Modified (same letters as EBCDIC)

This format uses the same character representation as EBCDIC. Thus if an MQ message was converted from EBCDIC to ASCII the same set of letters would mean the same numeric and signed representation, even though the hexadecimal values would bear no relationship to the numeric value.

Digit	EBCDIC Hex-Binary	EBCDIC Char	ASCII Hex	ASCII Char
+0	X'C0' - 1100 0000	{	X'7B' - 0111 1011	{
+1	X'C1' - 1100 0001	A	X'41' - 0100 0001	A
+2	X'C2' - 1100 0010	B	X'42' - 0100 0010	B
...				
-0	X'D0' - 1101 0000	}	X'7D' - 0111 1101	}
-1	X'D1' - 1101 0001	J	X'4A' - 0100 1010	J
-2	X'D2' - 1101 0010	K	X'4B' - 0100 1011	K

PL/I compilers on ASCII systems use this.

Strict

This format uses an alternative hexadecimal value for the negative numbers in ASCII which does bear a relationship to the number. The strict definition for zoned, or over punched numbers is that each digit is represented by a zoned bit value in the left-most four bits (nibble or half-word) and the binary value for the digit in the right-most four bits (nibble or half-word).

Digit	EBCDIC Hex-Binary	EBCDIC Char	ASCII Hex	ASCII Char
+0	X'C0' - 1100 0000	{	X'30' - 0011 0000	0
+1	X'C1' - 1100 0001	A	X'31' - 0011 0001	1
+2	X'C2' - 1100 0010	B	X'32' - 0011 0010	2
...				
-0	X'D0' - 1101 0000	}	X'70' - 0111 0000	p
-1	X'D1' - 1101 0001	J	X'71' - 0111 0001	q
-2	X'D2' - 1101 0010	K	X'72' - 0111 0010	r

Gnu COBOL and MicroFocus COBOL use this.

Which format does MQEdit use?

MQEdit uses the modified format as that seems to make the most sense for data in MQ messages. Our thinking is that if you are using these types of fields, it is because keeping all the data as character data is important to you, and MQ will data convert the character data for you into ASCII, so using the same letter representation would be what you'd likely be using. We are interested in feedback on this decision. Do you have over-punched data in ASCII in your messages? Are you using the modified or strict data format? Do you need MQEdit to support both?

End of document

MQGem Software Limited

www.mqgem.com