MQSCX

Extended MQSC Utility

User Guide

Version 9.3.2

13th November 2023

MQGem Software Limited

www.mqgem.com

Notices

The following paragraph does not apply in any country where such provisions are inconsistent with local law.

MQGEM SOFTWARE LIMITED PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

While every effort has been made to ensure the accuracy of the information contained in this document no guarantees can be made. Similarly the applicability of information in this document may well depend on the customers operating environment. If you feel that that there are inaccuracies in this document please raise your concerns by sending an email to support@mqgem.com.

MQGem Software Limited may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

- IBM
- IBM MQ
- MVS
- z/OS

The following terms are trademarks of the Microsoft Corporation in the United States and/or other countries:

- Windows

The following terms are trademarks of the Open Group:

- Unix

Twenty Seventh Edition, November 2023
This edition applies to Version 9.3.2 of Extended MQSC Utility and to all subsequent releases and modifications until otherwise indicated in new editions.

(c) Copyright MQGem Software Limited 2012,2023. All rights reserved.

Table of Contents

1 INTRODUCTION....9

1.1 Feedback....9

2 MAIN CHANGES FROM PREVIOUS VERSION....10

3 GETTING STARTED....11

3.1 Installation....11
3.1.1 Windows....11
3.1.2 Unix....11
3.1.3 z/OS....11
3.1.3.1 z/OS Unix Installation....12
3.2 Licensing....12
3.3 Issuing your first commands....13
3.4 Screens....15
3.5 Command recall....15

4 EXTENDED FILTERING....16

4.1 Wildcards....16
4.2 FIND() feature....16
4.3 WHERE() clause....18
4.3.1 Abbreviations....20
4.3.2 Attribute presence....21
4.3.3 Variables....21

5 MODIFYING THE DISPLAY OUTPUT....22

5.1 =SORT and =SORTD....22
5.2 Attribute Modifiers....22
5.3 MAXRESP....22

6 POWER FEATURES....24

6.1 Command retrieval....24
6.2 Command auto-completion....24
6.2.1 Auto Complete in WHERE and SORT clauses....25
6.3 Object name auto completion....25
6.4 Double-click command completion....26
6.5 Command Lists....26
6.6 Repeating Commands....27
6.6.1 Repeating multiple commands....27
6.7 Key assignment....28
6.7.1 Keyboard mapping....28
6.7.2 Key commands....29
6.7.3 Keyboard mapping example....29
6.7.4 Keyboard mapping example (z/OS)....30
6.8 Synonyms....31
6.8.1 A single command....31
6.8.2 A list of commands....31
6.8.3 A list of synonyms....31
6.8.4 Recursion....31
6.9 Copy and Paste....31
6.9.1 Copy....31
6.9.2 Paste....32
6.10 Undo/Redo....32
6.11 Display Totals....32
6.12 Namelist modification....33
6.12.1 =add()....33
6.12.2 =rmv()....33
6.13 Search....34

7 CONTROL LANGUAGE....35

7.1 Getting started with the control language....35
7.2 Variables....36

7.2.1 User Variables ... *36*
7.2.2 Arrays ... *37*
7.2.3 System Variables ... *37*
7.2.4 Constant Variables ... *39*
7.2.5 Response Variables ... *39*
7.3 Variable Scope and Stack Frames ... 40
7.4 Expressions ... 42
7.4.1 Data Types ... *42*
7.4.2 Coercion ... *42*
7.4.3 String Concatenation ... *43*
7.5 Inserting code fragments ... 43
7.6 Substitution commands ... 44
7.6.1 Functions ... *44*
7.7 Running the control commands ... 44
7.8 General syntax ... 47
7.8.1 Continuation ... *47*
7.8.2 Comments ... *47*
7.9 Statements ... 47
7.9.1 break ... *47*
7.9.2 continue ... *47*
7.9.3 foreach(....) clause ... *48*
7.9.4 foritem(....) clause ... *48*
7.9.5 fprint statement ... *49*
7.9.6 goto ... *49*
7.9.7 if(....) clause ... *50*
7.9.8 label ... *50*
7.9.9 leave ... *50*
7.9.10 print statement ... *51*
7.9.11 return ... *52*
7.9.12 var ... *52*
7.9.13 wait() statement ... *53*
7.9.14 while(...) clause ... *53*
7.10 Functions ... 54
7.10.1 Function Basics ... *54*
7.10.2 Function Definitions ... *55*
7.10.3 Function Invocation ... *55*
7.10.3.1 Recursion ... 56
7.10.3.2 Mutual Recursion ... 56
7.10.4 Dynamic Execution ... *57*
7.10.5 Comments ... *58*
7.11 Invoking other programs from your script ... 58
7.11.1 Synchronously ... *59*
7.11.2 Asynchronously ... *59*
7.12 Bootstrap file ... 59
7.13 Examples ... 61
7.13.1 Calculating the age of a queue (age.mqx) ... *61*
7.13.2 Sorting data (sort.mqx) ... *63*
7.13.3 Printing out namelist values to a file (fnamelists.mqx) ... *65*
7.13.4 The fullest queues (top.mqx) ... *67*
7.13.5 Reading files (fgets.mqx) ... *68*
7.13.6 Transmission Queues (xmitqs.mqx) ... *68*
7.13.7 Augmenting the command line using a function (conns.mqx) ... *69*
7.13.8 RESET QSTATS ... *70*
7.13.9 Monitoring (monitor.mqx) ... *71*
7.13.10 Emulating z/OS responses (disdqm.mqx) ... *71*

8 DEBUGGING ... 74

8.1 Debugger ... 74
8.1.1 <Enter> ... *75*
8.1.2 print ... *76*
8.1.3 eval ... *76*
8.1.4 Assignment ... *76*
8.1.5 list (short-form 'l') ... *77*
8.1.6 llist (short-form 'll') ... *77*
8.1.7 where ... *77*

8.1.8 Breakpoints *78*
8.1.8.1 bl 78
8.1.8.2 bp 78
8.1.8.3 bc 78
8.1.9 end *78*
8.1.10 run *78*
8.1.11 runout *78*
8.1.12 sf *79*
8.1.13 Help (short-form ?) *79*
8.1.14 Command alteration *80*

9 MODIFYING THE CLIENT CHANNEL DEFINITION TABLE 81

9.1 Channel Table Location 81
9.1.1 File Path *81*
9.1.2 File Name *81*
9.2 Binary CCDT file specifics 82
9.2.1 Channel Definition Version *82*
9.2.1.1 Useful commands 82
9.2.1.2 Useful functions 83
9.2.2 Authentication Information Objects *84*
9.3 JSON CCDT file specifics 85

10 CCDT MQSC COMMANDS 86

10.1 Channel Type 86
10.2 Generic names 86
10.3 =SORT Clause 86
10.4 =FIND Clause 86
10.5 =MAXRESP Clause 86
10.6 Where Clause 86
10.7 Where Clause on Alter 86
10.7.1 ALTER CHANNEL *87*
10.7.2 Syntax diagram for ALTER CHANNEL *87*
10.7.3 Parameter descriptions for ALTER CHANNEL *87*
10.8 DEFINE CHANNEL 88
10.8.1 Syntax diagram for DEFINE CHANNEL *88*
10.8.2 Parameter descriptions for DEFINE CHANNEL *88*
10.9 DELETE CHANNEL 90
10.9.1 Syntax diagram for DELETE CHANNEL *90*
10.9.2 Parameter descriptions for DELETE CHANNEL *90*
10.10 DISPLAY CHANNEL 91
10.10.1 Syntax diagram for DISPLAY CHANNEL *91*
10.10.2 Parameter descriptions for DISPLAY CHANNEL *91*
10.10.3 Requested Parameters *92*
10.11 MOVE CHANNEL 93
10.11.1 Syntax diagram for MOVE CHANNEL *93*
10.11.2 Parameter descriptions for MOVE CHANNEL *93*

11 MQEV EVENT MONITORING 94

11.1 Licensing 94
11.2 MQSCX running modes 94
11.3 Displaying MQEV data using MQSCX 94

12 CHANGING APPEARANCE 97

12.1 Prompt 97
12.1.1 Conditional Insertion *98*
12.2 Colours 99
12.2.1 Setting the various screen elements *99*
12.2.2 Switching colours off *99*
12.3 Columns 99
12.3.1 Content Columns *100*
12.3.2 Columns based on fixed width portions of the screen *100*
12.4 Separators 100
12.5 Window re-size support 103

13 CONFIGURATION FILE 104

14 FILE MODE....105

14.1 Piping a file in from stdin....105
14.2 Explicitly passing in an input file....105
14.3 Explicitly asking for file mode....105
14.4 Implicitly obtaining file mode....105
14.5 File mode separators....105
14.6 File mode features....106
14.7 Configuration file....106

15 Z/OS FILE NAME FORMAT....107

16 MQSCX PARAMETERS....108

16.1 Initial Command....110
16.2 Channel Definition Fields....110

17 LICENSING....111

17.1 Licence Types....112
17.2 Licence File Location....112
17.2.1 When running MQEV in z/OS UNIX....113
17.2.2 When running MQEV interactively in TSO....113
17.2.3 When running MQEV from JCL....113
17.3 Multiple licences....113
17.4 Changing your licence file....113

18 COMMANDS....114

18.1 Command Help....114
18.2 =cache....114
18.3 =ccdt....115
18.4 =ccdtj....115
18.5 =checkver....115
18.6 =clear....116
18.7 =cls....116
18.8 =colour....116
18.8.1 Colour limitations....117
18.9 =conn....118
18.9.1 User and Password....120
18.9.2 Client Channel Definition Parameters....120
18.9.3 Using your MO71 Configuration File....121
18.10 =count....121
18.11 =disc....122
18.12 =echo....122
18.13 end....123
18.14 =import....123
18.14.1 Nesting imports....124
18.15 =key....125
18.16 =keyname....126
18.17 =merge....127
18.18 =mqev....127
18.19 =mqsc....128
18.20 =output....128
18.21 =purge....128
18.22 quit....128
18.23 =repeat....129
18.24 =set....130
18.25 =show....133
18.26 =syn....134

19 CONNECTING TO THE IBM MQ QUEUE MANAGER....135

19.1 Connecting to a local queue manager....135
19.2 Connecting to a remote queue manager over a client connection....135
19.3 Reply Queue....136
19.3.1 Temporary Reply Queue Prefix....136
19.4 Connection Status....136
19.5 Switching connections....136

20 PROGRAM SETTINGS....137
20.1 Multiple Users....137
21 PERFORMANCE....138
21.1 Performance Test....138
22 NATIONAL LANGUAGE SUPPORT....139
23 SECURITY....140
23.1 Example security commands for Distributed Platforms....140
23.2 Example security commands for z/OS (using RACF)....141
24 RETURN CODE....142
25 PROBLEM DETERMINATION....143
25.1 General frequently asked questions....143
25.1.1 It says 'Please set required location of configuration file.....' ?....143
25.2 MQ Related frequently asked questions....143
25.2.1 What does 'Can not find IBM MQ on this machine' mean ?....143
25.2.2 Why is my connect failing ?....144
25.2.3 Why does MQSCX make two connections to the Queue Manager ?....144
25.2.4 Why do I see 'Command Server or network slow in responding' ?....144
25.3 Keyboard related frequently asked questions....145
25.3.1 Why is there an unnatural delay when I press the <ESCAPE> key ?....145
25.3.2 Why do F1, F2, F3 and F4 not work but the other function keys do ?....145
25.3.3 Why do I get told key or key sequence is unrecognised ?....145
25.3.4 Paste is not working on Linux....145
25.4 Display related frequently asked questions....145
25.4.1 Why is there a delay for a second or two when I resize the screen ?....145
25.4.2 Why are Red and Blue interchanged ?....145
25.4.3 My screen shows everything in the wrong place, why is that ?....145
25.5 Command Script file problems....146
25.5.1 Command script not returning....146
25.5.2 Responses not as expected....146
25.5.3 Don't forget the debugger....146
25.5.4 Debugging functions....146
25.6 Support....146
26 CHANGES IN PREVIOUS VERSIONS....147
26.1 Changes made in Version 9.3.1....147
26.2 Changes made in Version 9.3.0....147
26.3 Changes made in Version 9.2.1....147
26.4 Changes made in Version 9.2.0....147
26.5 Changes made in Version 9.1.0....148
26.6 Changes made in Version 9.0.1....149
26.7 Changes made in Version 9.0.0....149
26.8 Changes made in Version 8.0.1....150
27 MIGRATION FROM PREVIOUS VERSIONS....151
27.1 Migrating from a version prior to Version 9.3.2....151
27.1.1 Backward search....151
27.1.2 MQSCX on Windows is now 64-Bit....151
27.2 Migrating from previous builds of Version 9.3.1....151
27.2.1 JSON CCDT values....151
27.3 Migrating from a version prior to Version 9.3.1....151
27.4 Migrating from a version prior to Version 9.3.0....151
27.5 Migrating from a version prior to Version 9.1.0....151
APPENDIX A. EXPRESSION OPERATORS....152
APPENDIX A.1. WILDCARDS....153
APPENDIX A.2. OPERATOR SYNONYMS....153
APPENDIX B. EXPRESSION FUNCTIONS....154
APPENDIX C. KEY NAMES....160

APPENDIX C.1. EMULATOR KEYS.....160
APPENDIX D. KEY ACTIONS.....161
APPENDIX E. VALID NAMES 162

1 Introduction

The world of IBM MQ contains many ways to configure and monitor your MQ system so the first question you may be asking yourself is 'why do we need another one?'. The answer to that question is fairly straightforward. There are a number of Graphical User Interfaces (GUIs), such as MQ Explorer, but these can often seem complicated, slow and in many ways unwieldy. So a large number of administrators still use RUNMQSC to make quick simple changes to their configuration. Unfortunately RUNMQSC is perhaps a little too simplistic and some of its inherent limitations can become a little frustrating. MQSCX removes these frustrations and opens up a whole new range of possibilities with the addition of control language constructs. Not only can a command file issue a set of commands but can act on the responses. This means that command files can be written which:

- Augment or modify the current set of command responses
- Display the consolidated result of multiple commands
- Perform health checking
- Perform monitoring
- Write a report file of the data from your Queue Manager
- Check objects against your naming conventions.
- etc

In fact the list is virtually endless. MQSCX allows you to write a simple set of control commands to solve any number of situations.

MQSCX can be run on a variety of platforms and can administer all versions of IBM MQ which are in support and which provide PCF and remote MQSC interfaces. In particular this includes the z/OS platform including Queue Sharing Group (QSG) installations.

1.1 Feedback

I am always interested in hearing user views, whether good or bad, and would love to hear your opinions, comments and suggestions. So, if you would like to make a comment either about MQSCX itself or this manual then please do contact me at support@mqgem.com.

So, without further delay let's see what MQSCX has to offer…….

2 Main changes from previous version

If you are already familiar with **MQSCX** then you are probably mainly interested in what's new in this version. So, briefly, here is a summary of the changes for this version.

1. **Improved column support**
   MQSCX can now look at the entire `DISPLAY` command response and display each attribute in their own column. For more information please refer to Chapter 12.3 Columns on page 99.

2. **`=SORT()` available on `DISPLAY` commands**
   For more information please refer to Chapter 5.1 =SORT and =SORTD on page 22.

3. **`=MAXRESP()` available on `DISPLAY` commands**
   For more information please refer to Chapter 5.3 MAXRESP on page 22.

4. **Modifiers to add and remove attribute names from `DISPLAY` output**
   For example a command such as **`DISPLAY QUEUE(*) -TYPE`**
   For more information please refer to Chapter 5.2 Attribute Modifiers on page 22.

5. **Improved QSG `DISPLAY` output with QSGQM available for expressions.**

6. **New function `cmpver(v1,v2)` allowing comparison of version strings**

7. **MQSCX on Windows is now 64-Bit**
   Since it has been announced that IBM MQ is dropping support for 32-Bit programs all **MQSCX** programs are now 64-Bit.

8. **New settings**

`colstyle`	Controls what type of columns are wanted (auto | fixed | content)
`cssort`	Controls whether sorting with `=SORT` should be case sensitive.
`defmaxresp`	Default number of responses shown for `DISPLAY` command
`listval`	Specifies whether list attributes should be displayed in a single or multiple columns.

3 Getting Started

3.1 Installation

If MQSCX has already been installed you can skip this section. By default the program will try to write the configuration file to the same directory that the program itself is in so ideally the user should have write access to this directory. If you wish you can use an environment variable, MQSCXCFG, to specify an alternate directory.

The examples in this manual assume that the program executable is in the path. The examples and screen-shots are from a Windows system but very similar screens will be seen in Unix. The code should run correctly on most versions of Windows and Unix however it is always strongly recommended that you 'try before you buy'. In other words, request a trial licence, play with the program and ensure that it meets you needs. If all is well then you can go ahead with the purchase. If you encounter problems then please contact support@mqgem.com and we'll see if we can help.

3.1.1 Windows

MQSCX is provided as a simple zip file. Once you have downloaded this file you should unzip it into a location which is either in your path or can be explicitly referenced on the command line.

3.1.2 Unix

MQSCX is provided as a simple tar or gzip file. Once you have downloaded this file you should untar the file using one of the following commands depending on the file type (note that the file name depends on the platform).

```
tar -xvzf mqscx.tgz

OR

gzip -d mqscx.tar.gz            (If file is a gz file)
tar -xvf mqscx.tar
```

Move the file to a directory which is either in your path or can be explicitly referenced on the command line.

3.1.3 z/OS

MQSCX is provided as a simple zip file. Once you have downloaded this file you should unzip before following these instructions.

The zip file contains the following:-

- ***MQSCX.SEQ*** – sequential file containing the **MQSCX** program
- ***MQSCX.JCL*** – example JCL for running the program in batch

Once unzipped, transfer the ***MQEV.SEQ*** file to a z/OS system using the following commands.

```
ftp> binary
ftp> quote site recfm=FB lrecl=80 blksize=3120 blocks primary=1000
ftp> put MQSCX.SEQ
```

Once the ***MQSCX.SEQ*** file is successfully FTPed to your z/OS system, from TSO use the following command:

```
receive inds(MQSCX.SEQ)
```

When prompted for a filename, reply

```
DSN(USER.LOAD)
```

The other file in the zip is a text file and can be transferred using **`ascii`** mode in ftp if you plan to use it.

MQSCX can be run on z/OS in BATCH and an example piece of JCL is provided in the zip file as noted above. **MQSCX** can also be run interactively, e.g. from the TSO/E READY prompt, or the ISPF Command Shell (=6). It can also be run in z/OS UNIX (see below).

3.1.3.1 z/OS Unix Installation

If you wish to run **MQSCX** in z/OS UNIX, you can copy the MVS executable module that you have installed in the previous section, to a directory in z/OS UNIX with the following command.

```
TSO OPUT 'GEMUSER.USER.LOAD(MQSCX)' '/u/gemuser/bin/mqscx' BIN
```

3.2 Licensing

The next consideration is to ensure that you have a suitable licence for the program. This will depend on the type of program you have downloaded and installed.

Program Type	Licence required ?
Beta Test	No, although the program will be time limited
Trial Program	No, although the program will be time limited
Full Program	Yes

If a licence file is required then please ensure you have a licence file installed in the appropriate location (usually the same directory as the MQSCX program).

Please see section*17 Licensing* on page 111 for more information about licence files.

3.3 Issuing your first commands

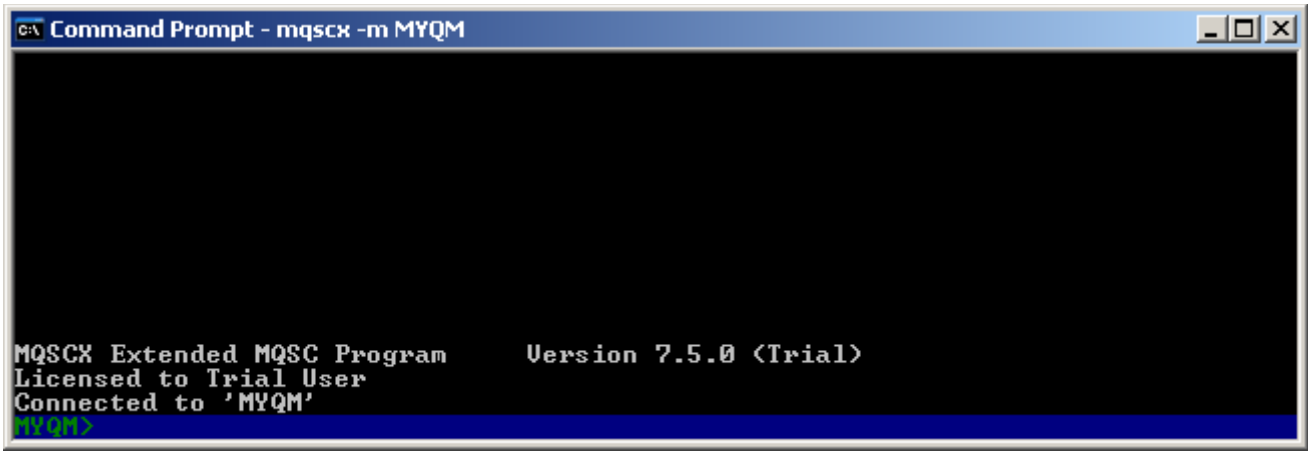
In these first examples we will assume that we have a local Queue Manager which is up and running which we are authorised to connect to. If life is more complicated than that for you then please refer to see section *19 Connecting to the IBM MQ Queue Manager* on page 135 to see how to connect to your Queue Manager.

The first thing we need to do is start **MQSCX** and give it the name of the Queue Manager we wish to connect to.

```
C:\>mqscx -m MYQM
```

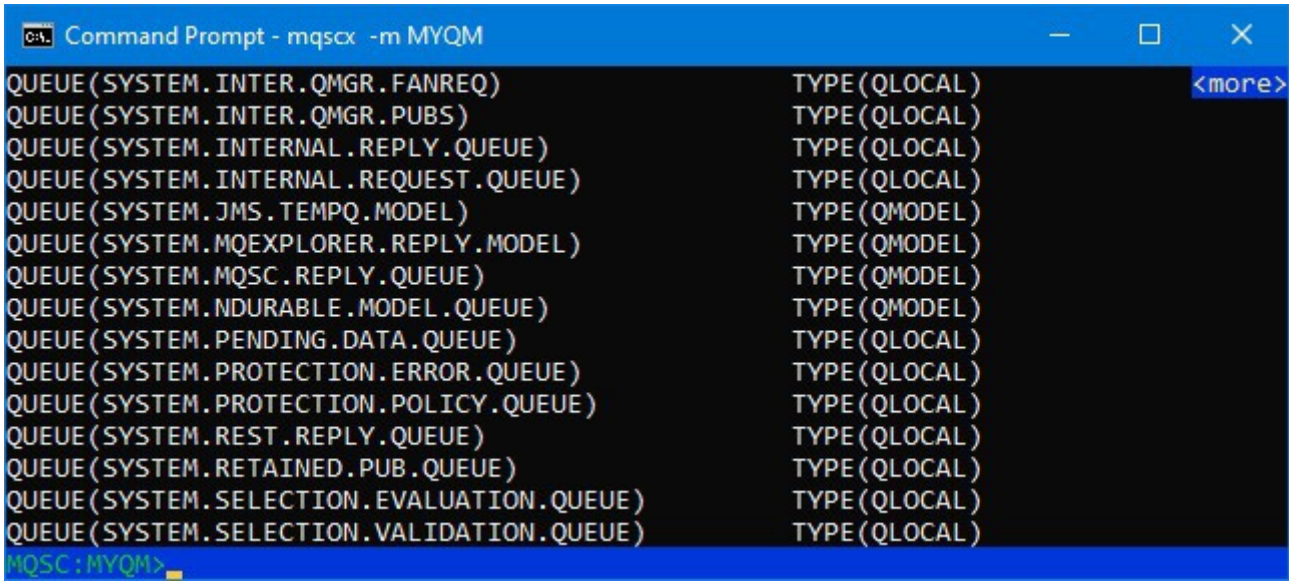
The **-m** parameter is not necessary if you wish to connect to the default Queue Manager.

This will run the program and display a panel something like this……

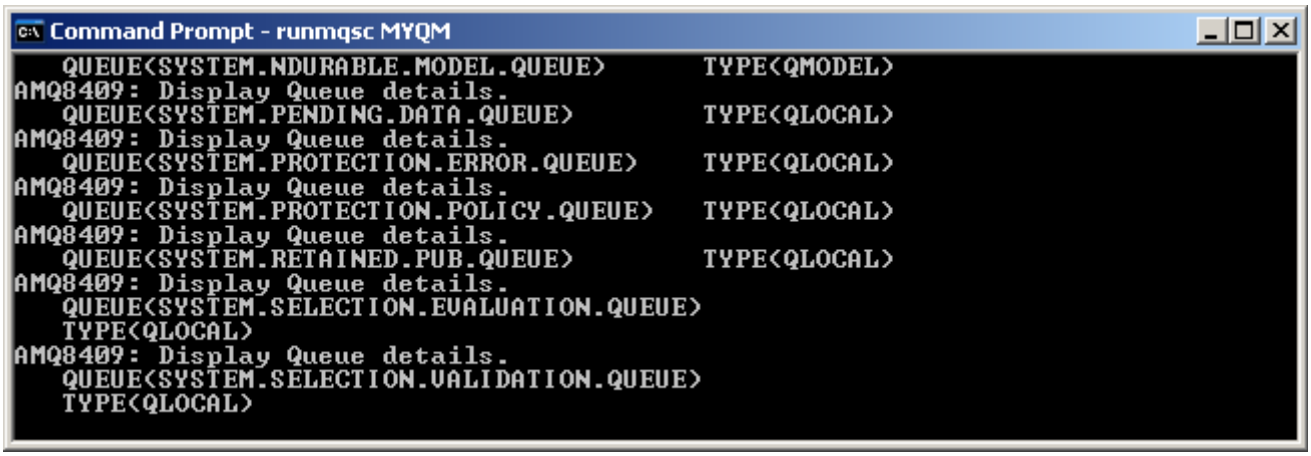


Now, this doesn't look hugely dissimilar to an equivalent RUNMQSC invocation but there are key differences. Firstly you will notice that the command prompt is at the bottom of the screen, it is a nice striking blue colour and the command prompt itself is preceded by the name of the Queue Manager. The prompt and colours can be changed, as we'll see later, but the location of the command prompt is always at the bottom of the screen. As commands are typed the result will be shown scrolling above the prompt but the prompt will stay in a fixed position.

So, let's see what happens when we enter a simple command. Type **DIS Q(SYS*)** and press enter. We now see something like this….



Again we see similarities with **RUNMQSC** output but the display is far from the same. For comparison let's remind ourselves what **RUNMQSC** would have looked like…



You can see that the two screens look fairly different although they are both showing the same information.

So, rather like a 'spot the difference' puzzle, let us explain the differences……

- **Response format**
  As I'm sure you know, **RUNMQSC** formats the output of its responses into two columns provided the data fits. Sometimes two columns is appropriate but often it isn't. By default MQSCX formats the output from the command server into an appropriate number of columns for the width of the output screen. In this case MQSCX is trying to fit the data into four columns however if the data doesn't fit into a single column then it spans into the next column or columns as necessary.

  If you wish you can change this behaviour or indeed ask for any other number of columns in the output. Please see section *12.3 Columns* on page 99 for information of how to do this.

- **Display Details message**
  Before each object **RUNMQSC** outputs a message to explain what it is about to output. These messages can often be distracting; they take up valuable screen real estate and are usually superfluous anyway. So, by default, **MQSCX** doesn't always display them. You can, if you wish, reinstate them by changing a program setting – please see section*12.4 Separators* on page 100 for more information.

- **The <more> indicator**
  The **<more>** indicator at the top right of the screen indicates that there is more data above the displayed output. Effectively it tells you that the output can be scrolled up and down. In **RUNMQSC** looking at previous responses can be awkward, in **MQSCX** it's as simple pressing 'Page Up' and 'Page Down', go on, try it.

- **Command prompt**
  The command prompt, as we've mentioned before, is quite different from the simple prompt you have with **RUNMQSC**. Our next few exercises will see what we can do however the key difference is that you can edit this field using the left, right, delete, backspace and insert keys[1]. Additionally, by default, pressing the ESCAPE key will empty the command line and move the cursor to the start of the command line. Unix users may wish to read the discussion about the ESCAPE key in *Appendix C.:Key Names* on page 160 if they notice a delay when using this key.

Many of the differences between **RUNMQSC** and **MQSCX** output is to make more efficient use of screen real estate. In this simple example we see that in the same size screen **RUNMQSC** displays 7 queue definitions where as **MQSCX** displays 15!

[1] This was mainly a problem for Unix users before, depending on the emulator you were using.

3.4 Screens

The MQSC output screen is actually just one of four display screens, although it is the most important. However, let's introduce you to the other screens. These can be displayed using the **=show** command or, by default, pressing a function key.

Try typing in one or two of the following commands:

Command	Default Key assignment	Description
`=show scrn(help)`	F1[2]	Show the help screen
`=show scrn(mqsc)`	F2	Show the MQSC display
`=show scrn(keys)`	F3	Show the key assignment screen
`=show scrn(sets)`	F4	Show the settings screen.

We shall talk about the various commands you can enter into **MQSCX** later so do not be too concerned about the exact format of the commands or what you can do. All we really want to know at the moment is that there are these four displays and they all give us some useful information. The majority of the time you will be using the MQSC screen but it is handy to know the others are there should you need them.

Now, if you've not already done it come back to the MQSC display and we'll continue looking at what we can do with **MQSCX.**

3.5 Command recall

One of the most frustrating usability features of **RUNMQSC**, certainly on Unix, is its lack of any command retrieval. We've probably all been there, you type in a long **`DEFINE CHANNEL`** command specifying half a dozen attributes, hit enter, only to be told that you had a syntax error and it dawns on you that you missed a quote character. You then have to type it all in again. Wouldn't life be so much simpler if you could retrieve your previous command?

Well, perhaps not surprisingly, with **MQSCX** you can. By default pressing 'up' and 'down' on the keyboard will cycle backwards and forwards respectively through the previous commands. So, have a play with it, type in a few simple MQSC commands and then use 'up' and 'down' to retrieve the commands, edit them and re-issue. There are a few other things you can do with retrieval, please see section *6.1 Command retrieval* on page 24 for more details.

So, that's it for the basic features, can we do anything more? Of course we can! There are plenty more exciting features for us to take a look at such as:

- Command auto-completion
- Object name auto-completion
- Keyboard assignments
- Extended filtering
- Extended editing capabilities
- and more.....

Let's take a look at some of the more powerful features of **MQSCX**.

[2] Some emulators assigned function key F1 for their own use so it may be necessary to disable the interception of F1 or reassign the **`=show scrn(help)`** to an alternative key.

4 Extended Filtering

As we all know the MQSC language has a number of ways that you can filter the displayed output. However, there are some significant restrictions. **MQSCX** alleviates many of these; we'll consider each in turn.

4.1 Wildcards

The native MQSC language supports wildcards in its DISPLAY commands. This is invaluable when you want a single command to return multiple entries. For example:

```
DIS Q(*)              Show all queues
DIS CHS(*)            Show all current channels
```

Native MQSC allows us to go further and specify a root value before the wildcard.

```
DIS Q(SYS*)           Show all queues starting 'SYS'
DIS CHS(XP*)          Show all current channels starting 'XP'
```

Unfortunately that's where the current MQSC language stops. It will only allow a single wildcard character and it must be at the end of the name.

MQSCX extends the MQSC language and allows the user to specify '*' wildcards anywhere in the name. It also allows you to specify a '?' signifying a single character. So, with **MQSCX** the following commands are now possible.

```
DIS Q(*MVS*)          Show all queues containing the string 'MVS'
DIS Q(SYSTEM*PUB*)    Show system queues which contain the string 'PUB'
DIS Q(??)             Show any queues with only a two character name
DIS CHL(*QMX*)        Show any channels with 'QMX' in the name
```

Note that the way in which **MQSCX** achieves this feature is to modify the actual command issued to the command server. If a field contains a wildcard then the value is terminated at that first wildcard. If any responses are received the full wildcard is then applied to the returned values and only those which match the full wildcard specification are shown to the user. You can see this is operation by issuing an invalid command. Try, for example:

```
DIS Q(*PUB*) X
```

You will see that the action command issued to the command server was actually 'DIS Q(*) X'.

4.2 FIND() feature

Wildcards are a very simple way of filtering the data from the host by the object name. But suppose you want to filter by a different attribute? Well, then traditionally you would use the **`WHERE()`** clause. However, as we'll see later the **`WHERE()`** clause is quite complicated. People who have been using MQSC for years often still have to refer to the books to get their **`WHERE()`** clause correct. Primarily to get the syntax right but also to get the name of the returned attribute correct. Let's try a simple example. Suppose you know you have an application, let's say, amqsput.exe and you just wish to display it's connections. What is the command using the where clause?

Give up?

One might imagine that this would work

```
DIS CONN(*) WHERE(APPLTAG LK '*amqsput*')
```

or possibly this…

```
DIS CONN(*) WHERE(APPLTAG CT 'amqsput')
```

Unfortunately what you have to enter is something like this.

```
DIS CONN(*) WHERE(APPLTAG LK 'c:\nttools\amqsput.exe*')
```

But clearly this is far more complicated that we would like. Firstly it requires you to know a weird operator **LK** ('like'). Secondly it is non-intuitive, the **LK** operator matches wildcards but the wildcard must only be at the end of the value. Thirdly you need to know the exact attribute name. And lastly you even have to know the exact path to the program.

All we really want to do is filter out any responses which contain the string 'amqsput' and this is exactly what the **=FIND()** feature does.

Try typing in the command:

```
DIS CONN(*) ALL =FIND(amqsput)
```

How much simpler was that? What's more you can find any string or part of string anywhere in the response.

But you probably have one or two questions about the command right? So, let me see whether I can guess what they are:

- **Why did you specify 'all'?**
  Well, the **=find()** will filter the responses from the command server. However, it can only work with the data returned so you need to ensure that the response contains the field you are looking for. The simplest way of doing this is to specify 'all' which says 'return all attributes. Of course if you are cleverer than me and knew that it was **APPLTAG** then you could have specified. **DIS CONN(*) APPLTAG =FIND(amqsput)** Of course, usually when you're issuing a command of this nature you are doing so to find out information about the object, for example its process id, so returning all the attributes may well be what you want anyway.

- **Why is it =find(…) and not just find(…)?**
  This comes down to a question of future proofing. Currently the MQSC language itself has no **find()** attribute but it may in the future. By prefixing with a '=' I can be fairly certain that it is never going to clash with any extensions to MQSC by the IBM MQ product. You will see later on that I use the same principle for all the **MQSCX** commands.

- **Is it a case sensitive comparison?**
  A good question. Pat yourself on the back if you asked this one. The answer depends on whether the value is specified in quotes or not. Without quotes, as in the example above, it is case insensitive. However, had I specified **DIS CONN(*) ALL =FIND('amqsput')** then it would have been a case sensitive search.

As we said, the **=find()** feature is for doing very simple filtering based on a simple text search. If you need the filter to be a little cleverer then we need to use the **=WHERE()** clause below.

4.3 WHERE() clause

I have already mentioned the **WHERE()** clause and was not overly complimentary about it. It is complicated, has a difficult to follow syntax and has some considerable restrictions. So, why am I mentioning it again? Well, there are times unfortunately where a little complexity is unavoidable. If you want to ask more complicated questions then something a little more powerful than a simple search string is needed.

MQSCX has taken the **WHERE()** filter concept and removed many of those restrictions and enhanced it so that one can now issue many more powerful queries. If you followed along the **=find()** discussion then it won't surprise you to learn that **MQSCX** has implemented the **WHERE()** feature as **=WHERE()**. Again, the '=' sign ensures that it is clear who is doing the filtering. Is it the command server or is it **MQSCX**? Having this separation ensures that any future developments in either filter will be kept separate and they won't clash. That being said the **=WHERE()** filter has modelled itself very closely on the standard **WHERE()** clause. Clearly I could have invented a brand new syntax. However that seemed unnecessary and potentially confusing. Far better, at least for the moment, to build on the current syntax and just try to remove one or two of those little annoyances.

So, perhaps listing those restrictions might be a good place to start. If you are new to the **WHERE()** clause it might be worth you revisiting the MQ manuals to read a description of the syntax and come back when you have the basics. You will have seen that you can do lots of good stuff in a **WHERE()** clause but what might not be so obvious is what you can't do……which is:

- You can't use the primary object name in a **WHERE()** clause
- You can't use any of the sub-filter attributes in a **WHERE()** clause
- A **WHERE()** clause may only contain one operation – you can't link them together with AND and OR
- You can only have a single attribute in a **WHERE()** clause. For example, you can't compare one attribute with another.
- Although it supports wildcard comparisons the wildcards may only be at the end of strings.
- You can only have one **WHERE()** clause on a single **DISPLAY** command
- You can't use indicators[3] in a **WHERE()** clause.

Phew! Quite a list of restrictions and unfortunately most of these would be very nice to have.

But, as you may suspect, help is at hand because using the **=WHERE()** clause you can do all this, and more.
Let's try some real world examples. Provided you know the **WHERE()** syntax you should have no trouble working out what the command is doing. Just to keep it interesting we'll only stick to commands which would have been illegal to issue using the standard **WHERE()** clause, see if you can spot why the command would have been illegal.

```
DISPLAY Q(*) =WHERE(DESCR LK '*PUB*')
```

So, this command will show you all the queues whose descriptions contain the text 'pub'. Note that we can use multiple wildcards in the operand.

```
DISPLAY Q(*) =WHERE(CURDEPTH GT 0 AND QUEUE LK '*CHANNEL*')
```

Show me all the queues which are non-empty and have 'channel' somewhere in their name. This shows that you can link multiple expressions using 'and' and 'or' operators. If you prefer you can use the operators '&' and '|' to represent the 'and' and 'or' operators. Using '&' and '|' can make the expression easier to read and clearly involve less typing, partly because they are shorter but also because they don't require a space character to separate them from the attribute.

You can link expressions as many times as is appropriate. For example consider the following example.

```
DISPLAY CHANNEL(*) =WHERE(MCAUSER | SCYEXIT | SSLCIPH)
```

[3]An indicator is an attribute, usually returned on a status display, which contains two values in list format. Usually a short term and a long term average. Examples include NETTIME on channel status or QTIME on a queue status response.

This command is effectively asking to display all the channels with some type of security setting. However, perhaps a more useful command would be the inverse of this, to show the channels which don't have any security setting. Now, for those of us that remember our high school algebra we know we could write the inverse as

```
DISPLAY CHANNEL(*) =WHERE(!MCAUSER & !SCYEXIT & !SSLCIPH)
```

Here we're using the '!' (logical NOT) operator. This command will give us what we want but there are couple of things to make clear. Firstly, despite what our algebra teacher told us, this command will not return the exact inverse of the previous command. The reason is the Attribute presence rules described on page 21. Essentially a command will only return results for objects for which the attributes apply. This rule has been relaxed for the OR operator but not for the AND operator. Consequently the command above will never return CLNTCONN definitions since the MCAUSER attribute does not apply to CLNTCONNs. The second point to be made is that actually there is, of course, a much easier way to invert a command. Consider the following command:

```
DISPLAY CHANNEL(*) =WHERE(! (MCAUSER | SCYEXIT | SSLCIPH) )
```

This command is not only easier to understand but is guaranteed to return the inverse result for all objects regardless of their type.

Ok, so, what other flavours of =WHERE expressions are supported. Consider the following:

```
DISPLAY Q(*) =WHERE(CURDEPTH >= 0.80*MAXDEPTH)
```

This command compares two attributes of the same object.

It is effectively saying “Show me all the queues which are greater or equal to 80% of their maximum depth”.

This command demonstrates a couple of points worth mentioning

- **Mathematical expressions**
  The **=WHERE()** clause is essentially one big Boolean expression. If the expression evaluates to TRUE (non-zero) for any object then it is displayed. If it evaluates to FALSE (zero) then it is not displayed.

  Within that remit you are free to enter virtually any mathematical expression you like, subject to the supported operators. If the expression evaluates to a non-zero value then the object is displayed. So, for example a filter of **=WHERE(1)** is perfectly valid although somewhat superfluous.

  For a complete list of the **=WHERE** operators please see *Appendix A. Expression Operators* on page 152.

- **Operator synonyms**
  A number of operators have synonyms. In this case the operator **GE** could equally have been used. It is purely a matter of taste whether you have more of an SQL versus a mathematical background.

So, staying on a mathematical theme can we guess what this might do?

```
DISPLAY Q(*) =WHERE(QUEUE < 10)
```

To understand this filter we really just need to know how **MQSCX** handles an expression containing both strings and numbers. The answer is coercion[4]. **MQSCX** will automatically convert any string value to a numerical equivalent if it has an operation concerning both a string and number. The number it converts it to is its length. So, this command is saying, display any queues which have a name of less than 10 characters.

```
DISPLAY Q(*) =WHERE(QUEUE != UPPER(QUEUE))
```

[4]Coercion, in the computing sense, is merely the process of making one data type compatible with another data type.

Now, the first thing I should say is that this is doing a case sensitive comparison between two strings and by default string comparisons in the **=WHERE()** clause are case insensitive. So, before you issue this command you should make **=WHERE()** clauses case sensitive. You can do this with this command:

```
=set cswhere(on)
```

Don't worry too much at this stage about what this is doing just issue the **=set** command and then issue the DISPLAY command above. Of course depending on your system configuration you may or may not see results. What the command is doing is saying 'display only those queues whose name is not the same as the upper case version of their name'. In other words, display any queues which have lower case letters in the name. Commands like these can be useful to ensure compliance with installation standards although I will agree you won't be doing it often.

One of the most notable things about this command is the use of a function in the **=WHERE** clause. There currently aren't many functions available but for a complete list please see *Appendix B. Expression Functions* on page 154.

Anyway, before we continue let's go back to having case insensitive **=WHERE()** clauses

```
=set cswhere(off)
```

The next expression format we want to discuss is enumerated types. These are object attributes which have a fixed set of values, for example, suppose we wish to see the queues which are currently MQGET disabled. We would enter the command:

```
DISPLAY Q(*) =WHERE(GET = DISABLED)
```

Note that you must enter the fields this way round. It would be invalid to enter the command:

```
DISPLAY Q(*) =WHERE(DISABLED = GET)
```

This would generate a variable error since the program would look for a variable called 'disabled'.

Now let's try an example of a **=WHERE()** clause containing an indicator.

```
DISPLAY QSTATUS(*) =WHERE(QTIME.SHORT > 5000000)
```

This example, of course, requires that queue monitoring is on. Essentially an indicator consists of two fields, a long and a short average of the field value of time. These two fields can be accessed individually by using the suffix 'short' or 'long' to the attribute name. If no suffix is specified then the short indicator is used. Note that the short and long suffixes are something I have created to allow you to address the two values individually. IBM MQ itself does not recognise these directly, so you can not, for example, issue the following command:

```
DISPLAY QSTATUS(*) QTIME.SHORT
```

4.3.1 Abbreviations

Until now we have been been good citizens and have always specified the full name of the attribute. However, **MQSCX** allows you to specify only as many characters of the name to ensure it is unique.

As an example let's take the simple command to show only those queues which have messages on them.

The traditional command would be:

```
DISPLAY Q(*) =WHERE(CURDEPTH > 0)
```

but we can use this command and save on typing

```
DIS Q(*) =WHERE(CUR > 0)
```

In actual fact the mathematical ones amongst you will notice that actually since the current depth of a queue can't be negative then this boolean expression is just the same as saying 'cur'. In other words, the expression is TRUE if the curdepth is non-zero. Consequently we can save even more typing and use this command:

```
DIS Q(*) =WHERE(CUR)
```

4.3.2 Attribute presence

Of course it is possible to use an attribute in an expression which doesn't get returned for each object. Consider the expression:

```
DISPLAY Q(*) WHERE(RNAME NE 'xxxxxxxx')
```

Here we have a standard native MQ where clause. One might expect that this expression would show all queues since it is unlikely that you have any queues defined with an **RNAME** value of 'xxxxxxxx'. However, the semantics of the **WHERE** clause is such that the expression always evaluates to FALSE if a variable is not present in the response. In other words, other queue types, such as local queues do not have a substitute value. So, in actual fact, this expression will just show us remote queue definitions.

This mechanism is quite useful since, given the **WHERE** expression, it is likely that the user is only interested in remote queues. If you issue the expression against your own Queue Manager you will likely see all of your remote queue definitions, unless you really do have one which points to a queue called ' xxxxxxxx'!

However, as we have heard in the sections above, the **=WHERE** clause has been extended to include logical OR expressions so this rule has been this relaxed.

Consider the expression:

```
DISPLAY Q(*) =WHERE(RNAME | TARGET)
```

If the **=WHERE()** clause stuck to the original rule then this expression would never return any objects since **RNAME** is an attribute of a remote queue and **TARGET** is an attribute of an alias queue. The two attributes are mutually exclusive since they belong to different object types. However, the rule above has been relaxed and this expression will return a list containing all remote queues which have a value for **RNAME** and all alias queues with a value for **TARGET**.

4.3.3 Variables

An **=WHERE** clause can also use Variables. These are described in more detail later but in essence you defined a variable with a value and then subsequently use that variable in an **=WHERE** clause.

A simple example would be.

```
@chlmask = "'*WIN*'"
dis chl(*) =where(channel == <@chlmask>)
```

This command will show all channels containing the string “WIN”.

5 Modifying the DISPLAY output

In the previous chapter we saw how **MQSCX** helps you filter the DISPLAY output. In this chapter we introduce a few ways in which **MQSCX** allows you to change the way in which the results of a DISPLAY command are shown.

5.1 =SORT and =SORTD

The =SORT clause allows you to ask **MQSCX** to sort the data before it is displayed. For example,

```
DISPLAY QUEUE(*) =SORT(curdepth)
```

By adding =SORT the data will be shown in ascending queue depth order. Note that the value specifying the sort clause is an expression not just an attribute. This means that we can sort the queues according to the 'fullness' of their queue using the command.

```
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth)
```

Note that **MQSCX** will automatically add the attributes, such as CURDEPTH and MAXDEPTH, to the command output itself since clearly it needs these values to perform the sort. This means that the values of these attributes will be displayed on the command. However, you can suppress these if you prefer using the Attribute Modifiers below.

5.2 Attribute Modifiers

Attribute modifiers allow you to either remove or add the displaying of attributes. For example we can extend the previous command:

```
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth) -curdepth,maxdepth
```

This command asks for the display of the CURDEPTH and MAXDEPTH attributes to be suppressed. They are still returned on the command it's just that **MQSCX** will remove them from the display which can reduce screen clutter. You can actually remove them both in a single token using the following command.

```
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth) -#depth
```

The # character indicates that this applies to any attribute containing the following string. So, in this case, any attribute containing the string 'depth' will not be shown. Yet another alternative command is to remove every attribute and then add in only the ones you want to display. There are actually two ways to do this. Firstly using the **-#** we saw before but without a following string. Or secondly to use the special value **-ALL**. So the following two commands are logically equivalent:

```
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth) -#   +queue,type
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth) -ALL +queue,type
```

The output of these commands will be the same as when we just removed the depth attributes but in this case we are more explicit about what we want to see.

5.3 MAXRESP

The =MAXRESP clause is a way of telling **MQSCX** how many responses you are interested in. For example consider the following command.

```
DISPLAY QUEUE(*) =SORTD(curdepth/maxdepth) =MAXRESP(10)
```

Here we are telling **MQSCX** to sort the queue by 'fullness' and then to just display the top 10. In other words **MQSCX** will display the top 10 most full queues. If the =MAXRESP clause is not specified then **MQSCX** will use

the default MAXRESP value which, by default, is **unlimited**. However, you may feel that you wish to limit the amount of data which could be thrown at you. Is there ever any point, for example, in every getting more than 1,000 answers to a DISPLAY command?

You can set the default maxresp value using the command

```
=set defmaxresp(999)
```

The current value is shown in the settings screen.

6 Power Features

So, we've seen the basics of what **MQSCX** can do, now let's take a look at some other powerful features.

6.1 Command retrieval

I mentioned before how powerful command retrieval is but we didn't cover it in much detail. Essentially 'up' and 'down' will cycle backwards and forwards respectively through the previous commands. Not only that but the previous commands are maintained across program invocations so you can recall the commands you were using the last time you ran **MQSCX**. By default **MQSCX** will keep the last 50 commands you entered. If you would like to change this then you can use the **`=set`** command, please see section *18.24 =set* on page 130 for a description of how to do this.

In addition there are two handy mechanisms which can help you find the command you want quicker.

- **Command prefix**
  If the cursor is not in the first position on the command line then the characters before the cursor will be used as a prefix for the search. In other words, only commands starting with the given characters will be retrieved.

  For example, to retrieve the previous 'ALTER' command one merely needs to type 'AL' and press 'up.

  Of course this is presupposing there is an ALTER command to find!

- **Command content**
  Suppose you want the previous command which mentioned a certain string, let's say 'Q1'.

  Well, to do that you merely prefix the string with a '#' character. So, for example type '#q1' and then press up.

The command retrieval feature will remove command duplicates and the command comparison is case insensitive. This means that "DIS Q(*)" and "dis q(*)" are considered the same command, this is fine since these are essentially the same command. However, "dis q('Q1')" and "dis q('q'1')" are not the same command and yet will be treated as so. MQ recommends that users do not have MQ objects with same name but in different cases so it should not cause a problem but it is **MQSCX** behaviour to be aware of.

6.2 Command auto-completion

One of the problems with interactive MQSC is the need to remember a whole host of commands, fields and attributes. Even after years of working with the language it is still very easy to forget the exact spelling of the attributes. If would be handy if one could have the values automatically entered at the press of a button. Well, with **MQSCX** you can….and that button is the tab key.

Let's try it out……let's start off with an empty command line. Now press <tab>. You should see that a command has been entered for you ,"ALTER". Press a few more times. You'll see that the command changes. It is cycling through the commands you can issue. Now, try pressing <shift-tab>. Now you will see we're going through the same list backwards.

So, this is handy but it might take a long time to get to the command you want. Let's suppose it's "DISPLAY". Well, start with a blank command type 'DI' and then press <tab>. You'll see that it immediately fills in 'DISPLAY'. Pressing <tab> again will produce an error message telling you that there isn't another command starting with 'DI'. But that's OK because is was 'DISPLAY' we wanted anyway.

So, now press <space> and once again hit <tab>. It probably won't surprise you by now to see that it fills in a possible next part of the command. In this case 'AUTHINFO()'[5]. Again pressing <tab> a few times will cycle through the possible things you can 'DISPLAY'. Keep pressing <tab> until we come to 'QUEUE()'.

[5]Or ARCHIVE if you are administering a z/OS Queue Manager.

Phew! There are a lot of things you can display aren't there!. Of course you might be thinking by now that it would have been quicker just to type 'QUEUE()' itself. And, of course, you'd be right. Those of you that have been paying attention so far I'm sure will have guessed that the solution, as in the case of command retrieval, is to type the first character or two of what you are looking for. So, let's try it again. Clear the command line. A quick way to do this is just press the <esc> key. The <esc> key will clear the command line and move the cursor to the first position of the command line.

So, now let's now try typing 'di' <tab> <space>'qu'<tab>

There, that was much quicker, wasn't it ?

So, now let's complete the command. Let's say we're trying to issue the command

```
DISPLAY QUEUE(*) DESCR
```

Type '*' in the brackets. Press <end> to go to the end of the command. Type 'DES' and press <tab>.

Now, for such a trivial command this doesn't seem to buy you a great deal. However, how many of us can remember the exact spelling of all the MQ attributes? Sometimes you can only remember it contains certain characters. For example, take the heartbeat attributes. You might know there are two attributes on channels concerned with heartbeat but can't remember just what they are. So, use the trick of using the '#' character and enter the command…

```
DISPLAY CHANNEL(*) #HB_
```

Now press the <tab> key. You'll see that **MQSCX** cycles through the two heartbeat attributes, **BATCHHB** and **HBINT**.

So, as you can see auto completion saves you time in typing and is also a good memory aid for those of us who can't remember the exact spelling of every attribute. However, be aware that auto completion does not guarantee that the command you ended up with is valid. Nor, of course, does it guarantee that the command is the one you should be issuing so care is still needed. Generally speaking only commands and attributes appropriate for the platform you are configuring will be shown but due to the huge number of combinations this can not be guaranteed.[6]

6.2.1 Auto Complete in WHERE and SORT clauses

Pressing <tab> in a **WHERE()**, **=WHERE()**, **=SORT()** and **=SORTD()** clauses will also suggest possible values at the point of the cursor. Note that it is still possible to construct an invalid expression this way but it can be a useful memory aid.

6.3 Object name auto completion

So far our examples have just used '*' as the object name, we haven't been targeting a particular object. But suppose that's what we wish to do, can **MQSCX** help here? Well, naturally it can.

Let's try it. Type in a command such as 'DISPLAY QUEUE()' ideally using the mechanisms you have used earlier. Now, type the first character of the queue name you are after in the brackets and press <tab>. You should see that the first defined queue, alphabetically, starting with the given character is now suggested in the brackets. Pressing <tab> multiple times will cycle through the defined queues.

I suspect you can guess that if you type '#' as the first character then **MQSCX** will show you any queues containing the characters following the '#'. This can be very useful if you can't remember the exact spelling of an object name but you know some of the characters in it.

So, what is happening here? How does this work ? Well, **MQSCX** maintains a cache of the names of most of the MQ object types. The first time you press the <tab> character in an object field a request is sent to the server for all the object names of that type. As you tab through the values **MQSCX** is merely going through this retained list.

[6] If there are particular platforms and commands which you feel should be handled differently then please contact support and I will consider making changes to the program.

This immediately begs the question of when the cache is refreshed of course. Well, the cache is refreshed for two reasons:

- **It ages out**
  The object cache is maintained only for a certain amount of time before it is considered too out of date. In an ideal world the cache would be refreshed each time the user pressed <tab> but that would cause an excessive amount of background work. The amount of work required is different depending on whether the Queue Manager is local or remote. A Queue Manager is considered remote if some form of channel, be it a client or sender/receiver pair, is required to reach it. As a consequence the time outs themselves vary depending on whether it is a local or remote; being 60 and 1800 seconds respectively. You can change these values if required, please see section*18.2 =cache* on page 114 for information on how to do this.

- **It is requested by the user**
  At any time the user can request that the cache is discarded by issuing an '=cache purge' command. This command will purge all object definitions meaning that the next time the user presses <tab> in an object field a new set of definitions will be fetched from the server.

As I'm sure you know MQSC will upper case any object names which don't appear in quote (') signs. As a consequence **MQSCX** will automatically add quote signs around any name containing lower case letters. Similarly auto completion will remove quote signs if they are not needed.

6.4 Double-click command completion

It is quite common to issue one command to display a list of objects and then issue a more detailed command to see all the information about that object. For example issuing 'dis q(*) where(curdepth gt 0)'. Then you might want to issue a 'dis qstatus(X) all' to display details about one of those queues.

MQSCX can reduce the typing here by filling in the queue name from the previous response when you double click on the name. To see this in action issue the command 'dis q(*)'.

Now use auto-completion by typing 'di' <tab> <space> 'qs' <tab>. You should have a command line which looks like:

```
DISPLAY QSTATUS()
```

Now, choose a queue returned by your 'dis q(*)' command and using the mouse double click on the name. You should see the name automatically copied into the 'DISPLAY QSTATUS()' command.

Essentially any time you double click on on the screen the word you select will be copied to the command line. If there are empty brackets, '()', on the command line then the word will be copied between the brackets. If there aren't any empty brackets then it will be copied to the end of the command line.

6.5 Command Lists

Any time you can enter a single command you can actually enter a list of commands. A list of commands is created by joining a number of individual commands with semi-colons. So, for example we could enter the command:

```
dis q(Q1) ; dis q(Q2) ; dis q(Q3)
```

Pressing enter will then run each of the commands one after another and display the output.

You may feel there isn't much advantage in this since it involves just as much typing. However, there are a few advantages:

- If there are a few commands you regularly use together then using a list allows you to recall the list all together rather than recalling each command separately.

- When used together with the *=repeat* command, described later, you can easily repeat multiple commands. This is the simplest way to monitor multiple objects.

- When used with the initial command parameter it can be useful to be able to pass in multiple commands to be run when the program first initialises. For a description of the initial command parameter please refer to section *16.1 Initial Command* on page 110.

6.6 Repeating Commands

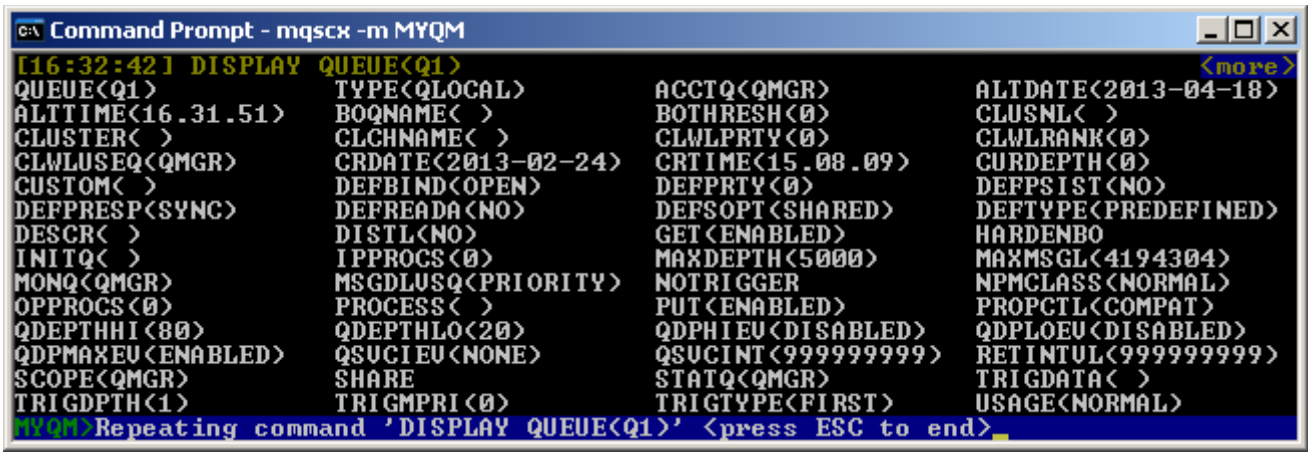
There are times when it would be nice to have the ability to repeatedly issue a command, usually a DISPLAY command, and be able to just watch the output of the command. In this way a very simple monitor can be established. For example, suppose you wanted to look at how the depth of a queue is changing, or look at how many channels are running or watch the time messages spend on a queue. There is no doubt that for complicated monitoring then a sophisticated monitoring tool is the way to go but for a simple demo or as a development aid a repeating MQSC command can be quite useful.

MQSCX provides an **`=repeat`** command. For a complete definition of the command please see section *18.23 =repeat* on page 129 but for now we'll just give a simple example of its use. Let's suppose we just want to monitor the definition of a queue, say 'Q1'.

We issue the command:

```
=repeat cmd(DISPLAY QUEUE(Q1)) delay(1)
```

This simple command requests that **MQSCX** repeatedly issues the 'DISPLAY QUEUE' command waits for one second before re-issuing the command. The result of the command is something like the following:



If you do this yourself you will notice that the timestamp at the top left of the screen is 'ticking' as each command is issued. While in this mode keyboard entry is restricted. The user can not scroll around the display or enter new commands. The only key which is recognized in this mode is <esc> which will cancel the REPEAT command. As a consequence you really need to ensure that the screen is large enough to contain all the output data you wish to monitor.

6.6.1 Repeating multiple commands

One of the first questions many people ask is "Can I repeat multiple commands?". Well, yes you can. There are two ways to do it.

1. **Use a command list**
   Any time you enter a command you can actually enter multiple commands separated by a semi-colon (;) character. So, in the example above if we wished to display two queues we could issue:

   ```
   =repeat cmd(DISPLAY QUEUE(Q1); DISPLAY QUEUE(Q2)) delay(1)
   ```

2. **Using an `=import` command**
   An import command, which is described later, allows you run a sequence of commands read from a file. So, for example, you could issue the following command:

   ```
   =repeat cmd(=import file(mycommands.mqs)) delay(1)
   ```

However, bear in mind that the output of all your commands must fit into a single screen if you wish to see the results since you can not scroll the window while in repeat mode.

6.7 Key assignment

In a number of places in this manual I have said things along the lines of 'press this key and this will happen'. That should indeed be true when you first start using **MQSCX**. However, we all have our favourite ways of doing things and, as a consequence, to a large extent, **MQSCX** allows you to reconfigure what happens when a key is pressed.

There are essentially two main reasons why you might want to do this, and we'll consider those next.

6.7.1 Keyboard mapping

It is amazing how many 'standard' key configurations exist and, as users, we all have our preferences. Even keys that have their own functions, for example 'page up' and 'page down', may have preferable alternatives such as F7 and F8. But what of 'repeat find'? Should it be <Ctrl-f> or <Ctrl-n> or a function key ? The fact of the matter is that whatever will feel natural to some will feel alien to others. The answer therefore seems is to allow users to set whatever mapping they choose.

By way of explanation let's take one of the first keyboard mapping quandaries I had to deal with. What keys should represent the 'previous command' and 'next command' actions? The initial version of **MQSCX** had these actions assigned to <Ctrl-up> and <Ctrl-down> respectively. The keys 'up' and 'down' just moved the cursor up and down. However, users of **MQSCX** agreed that it was more intuitive if just the 'up' and 'down' keys cycled through the commands so the default was changed. So the <Ctrl-up> and <Ctrl-down> keys are used to navigate the cursor. Some users may find it counter intuitive that 'left' and 'right' cursor keys work as expected but you need to press the <Ctrl> button if you want to move the cursor up and down. As one user put it "I want the program to be psychic!" Well, until we invent a way of allowing programs to determine what the user is thinking we'll just have to use mapping.

So, suppose we decided we didn't like these cursor defaults and wished to change them back? Let's use this as an example to show how keys can be reassigned. What would be the commands to issue cause the alternative assignments?

Enter the following commands and observe the change in behaviour.

```
=key(up)         action(cursor up)
=key(down)       action(cursor down)
=key(ctrl-up)    action(previous command)
=key(ctrl-down)  action(next command)
```

After entering these commands you'll notice that now the cursor keys just move the cursor in the appropriate direction and in order to cycle the previous commands you need to press the 'ctrl' key. It is, of course, entirely your preference over which mapping you choose. Just for completeness though, here are the commands you need to set it back to the default values:

```
=key(up)         action(previous command)
=key(down)       action(next command)
=key(ctrl-up)    action(cursor up)
=key(ctrl-down)  action(cursor down)
```

Actually since we are just returning to the default settings it isn't actually necessary to state the action. If an empty action is given the mapping returns to the original default.

```
=key(up)         action()
=key(down)       action()
=key(ctrl-up)    action()
=key(ctrl-down)  action()
```

For a complete description of the =key() command please see section *18.15 =key* on page 125.

For those of you who are not overly keen on typing these command you can use the *=import* command described in section*18.14* on page 123. The *=import* command allows you run a set of commands contained in a file. So, all you need do to try these examples is copy and paste the commands into a temporary file, then use **=import** to see their effects.

6.7.2 Key commands

Have you ever found yourself typing the same commands into **RUNMQSC** over and over again. For example 'DIS Q(*)' or perhaps 'DIS CHS(*) ALL'. Wouldn't it be nice if this could be achieved in a single key press? Well, that's what key commands are all about. Just as you can assign a key to a particular action you can also assign them to a command. For example:

```
=key(ctrl-q) cmd(DISPLAY Q(*) CURDEPTH)
```

I suspect that an intelligent reader such as yourself has already figured out what this means. However, just to be clear, this command is stating that each time <Ctrl-q> is pressed that the command 'dis q(*) curdepth' should be issued. In a similar fashion many more commands can be assigned to various keys; whichever commands you find yourself issuing again and again. Needless to say, this can be a real time saver as well as avoiding typing mistakes.

Virtually any command, be it MQSC or MQSCX commands, can be assigned to a key. You could even assign an **=import** command which means that a whole sequence of commands can be issued by a single key press. However, I would recommend that you think twice before assigning any destructive commands such as DELETE. There are times when it's still useful for a command to be awkward to enter!

For a complete description of the **=key()** command please see section *18.15 =key* on page 125.

6.7.3 Keyboard mapping example

During the development of this program one of the machines used was a Linux system running on a Power 64 chip. This machine was accessed using an SSH client over a VPN link. What was immediately noticeable was how different the keyboard mapping was to the other Linux machine **MQSCX** had been run on. The major difference was that some of the modifier keys such as 'shift' and 'ctrl' were not recognised in combinations with other keys. So, for example, 'ctrl-up' generated exactly the same keyboard code as 'shift-up'. Other keyboard combinations such as 'ctrl-home' didn't generate any keyboard codes at all!

However, it was noticeable that the 'alt' key did generate unique codes. So, in this environment the following keyboard mappings were used for **MQSCX** to function correctly.

```
=keyname(Shift-Down)  seq(1B5B42)
=keyname(Shift-Left)  seq(1B5B44)
=keyname(Shift-Right) seq(1B5B43)
=keyname(Shift-Up)    seq(1B5B41)

=keyname(Alt-Home)    seq(1B1B5B317E)
=keyname(Alt-End)     seq(1B1B5B347E)
=keyname(Alt-Down)    seq(1B02)
=keyname(Alt-Up)      seq(1B03)
=keyname(Shift-Alt->) seq(1B3E)
=keyname(Shift-Alt-<) seq(1B3C)
```

```
=keyname(End)          seq(1B5B347E)
=keyname(Home)         seq(1B5B317E)

=key(Alt-Up)           action(Cursor Up)
=key(Alt-Down)         action(Cursor Down)
=key(Ctrl-Up)          action(none)
=key(Ctrl-Down)        action(none)
=key(Alt-Home)         action(Cursor Top)
=key(Alt-End)          action(Cursor Bottom)
=key(Shift-Alt-<)      action(Select Cursor Top)
=key(Shift-Alt->)      action(Select Cursor Bottom)
```

You can see that in this environment many of the key combinations generate escape sequences. You may find that your system generates the same or entirely different combinations. Either way, these commands give you an idea of what you can do with the *=key* and *=keyname* commands.

If you are lucky enough that your setup generates the same keys then by all means enter the same commands into your own version of MQSCX. The easiest way to do this is using the *=import* command. Copy and paste these commands into a file and then use the *=import* command to run these commands from the file.

If you find that your system generates different key codes then you will have to enter different commands. The easiest way of capturing a particular key stroke is to use the special format of the *=keyname* command. Suppose the program didn't recognise the key combination 'shift-left'. All you need do is enter the command:

```
=keyname(Shift-Left) seq(#)
```

MQSCX will then wait for you to press the keyboard combination which should be associated with the name 'Shift-Left'. At any time you can look at the user assigned keys by going to the keys screen by pressing F3 (by default) or issuing the command **`=show scrn(keys)`**.

6.7.4 Keyboard mapping example (z/OS)

During the development of this program on z/OS, similar issues were seen as those above on the Power Linux system.

If you're lucky enough that your setup generates the same keys as in this example, again you can copy these into a file and use the **`=import`** command to run them.

```
=keyname(Home)       seq(27ADF1A1)
=keyname(End)        seq(27ADF4A1)
=keyname(PageUp)     seq(27ADF5A1)
=keyname(PageDown)   seq(27ADF6A1)
=keyname(Delete)     seq(27ADF3A1)
=keyname(Shift-Tab)  seq(27ADE9)
=keyname(Ctrl-Up)    seq(27D6C1)
=keyname(Ctrl-Down)  seq(27D6C2)
```

6.8 Synonyms

In a similar way to key commands you can also set up command synonyms. So, if you don't like the idea of having to remember lots of different key combinations how about giving commonly used commands nicknames. The way you do it is via the *=syn* command. A synonym can point to a number of different types of command.

6.8.1 A single command

```
=syn(dc) cmd(DISPLAY CHANNEL(*) ALL)
```

Issue this one command once and then, every time you enter a command of 'dc', the actual command issued will be 'DISPLAY CHANNEL(*) ALL'. This can save a lot of typing! Just like key commands you can assign virtually any command in this way, including both MQSC and MQSCX commands.

6.8.2 A list of commands

```
=syn(dqa) cmd(DIS q(Q1); DIS Q(Q2))
```

As with all commands fields it can be defined as a list of commands. Each command will be run in turn one after the other.

6.8.3 A list of synonyms

Suppose we wished to run both of the synonyms above we could enter the command:

```
=syn(both) cmd(dc; dqa)
```

The list can contain a mixture of both synonyms and commands.

6.8.4 Recursion

Since a synonym can 'point' to another synonym the possibility exists to have recursive definitions. If the program detects recursion then an error message will be displayed.

For a full description of the *=syn* command please see section *18.26=syn* on page 134.

6.9 Copy and Paste

The **MQSCX** program has native copy and paste capabilities rather than relying solely on the OS command shell.

UNIX only - On UNIX copying to the clipboard buffer is restricted to the **MQSCX** program only. This can be useful if one wishes to copy a value from a previous command output, such as an object name or connection id, but not ideal if you wish to paste the answer into a secondary program such as a word processor. The reason for this is that UNIX lacks the notion of a global clipboard buffer inherent in the UNIX system, in a similar fashion to the Windows clipboard[7].

For pasting the story is a little better although it does depend on the exact environment and the way in which **MQSCX** is being run. Essentially **MQSCX** uses CURSES for it's interface so will operate the way that CURSES supports. If you find your normal paste method not working try **<shift>+mouse** right-click or **<shift>+<insert>**.

6.9.1 Copy

To copy some text you first need to select the text. Move the cursor to the start of the text to be copied. Then, holding down the shift key, move the cursor to the end of the text to be copied. If you wish to select all the text in an area then you can press <Ctrl-a>. To copy the data to the clipboard press <Ctrl-c>. To cut the data to the clipboard then press <Ctrl-x>. Note that you can only cut the data from the command line since this the only input field.

[7] If anyone has a suggestion about how this restriction might be overcome then please feel free to contact me with any information which may be helpful.

6.9.2 Paste

To paste data move the cursor to the place where you want the data and then press <Ctrl-v>. Text may only be pasted into the command line since this is the only input field.

6.10 Undo/Redo

Each change made to the command line may, if required, be undone. This includes user typing or pasting data. If changes are undone then they can, if required, be redone. By the default the keys for doing this a <Ctrl-z> and <Ctrl-y> respectively but these keys can be changed if another scheme is more suitable.

The undo/redo sequence only spans a single command. Once a command is entered the undo/redo sequence is cleared.

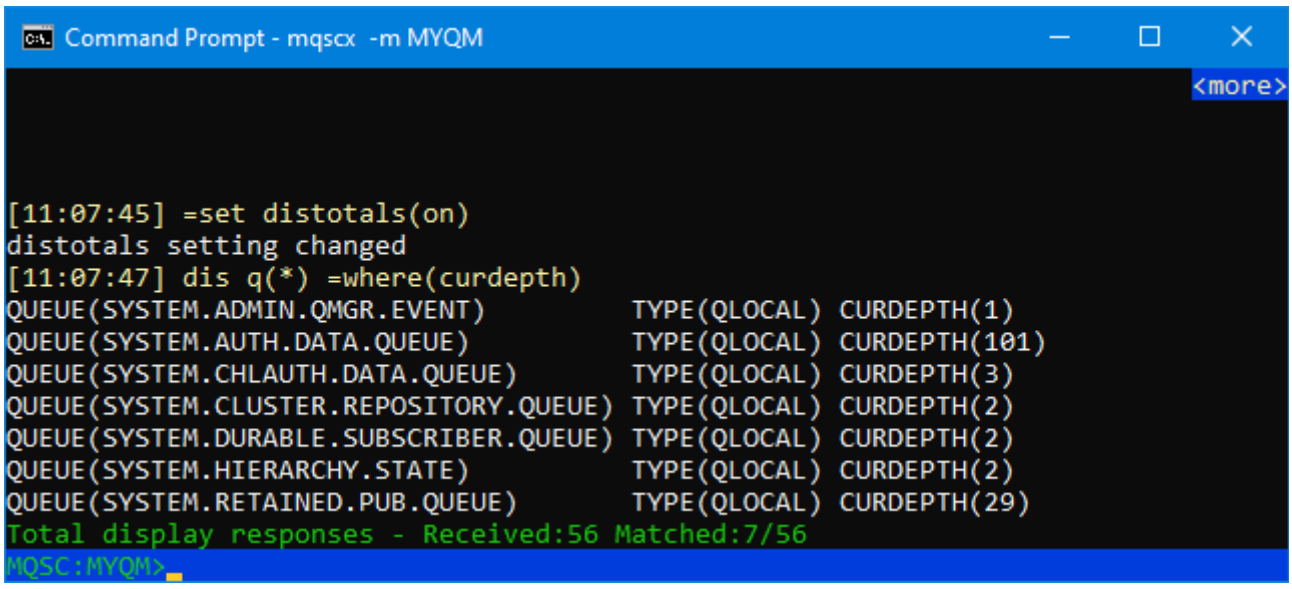
6.11 Display Totals

Have you ever issued a `DISPLAY` command and wished there was some simple way to know how many answers were returned? For example you might be trying to gauge how many channels or clients are running. Or perhaps how many connections you have to your Queue Manager. Well, **MQSCX** offers a very simple mechanism to find out. By default the option is switched off, since it uses an extra screen line, but to switch it on issue the command:

```
=set distotals(on)
```

Now whenever you issue a `DISPLAY` command you will get a summary line stating the number of responses. If you have used some sort of filter such as **`=FIND()`** or **`=WHERE()`** then you will be told both how many objects were returned to **MQSCX** and how many matched the filters. In addition if you use **`=MAXRESP()`**, and the display was limited because of it, then you will be told how many of the results have been shown.

So, let's try a simple example:



In this example, therefore, the total line is telling us that there are 56 queue definitions and 7 of the queues have messages on them.

6.12 Namelist modification

Have you ever wanted to make a small change to a namelist and discovered that the command to do it requires that you specify the entire list. This awkwardness is usually hidden from you if you use a GUI but there are times when you want to modify a namelist in a script file. Suppose, for example, that the installation of a new application requires you to modify a namelist with a new cluster or a new queue. This is very difficult to do in MQSC but in **MQSCX** it is very simple. It is made possible by the addition of two new fields **=add()** and **=rmv**.

This means we can now issue commands like:

```
ALTER NAMELIST(MYNL) =add(name1,name2,name3)
ALTER NAMELIST(MYNL) =rmv(name2)
ALTER NAMELIST(MYNL) =add('lowercasename')
```

6.12.1 =add()

The **=add()** keyword allows you add a list of names to the namelist. Normal MQ rules apply. If the given names are not enclosed in quotes then they will be added in upper case. If they are in quotes then the exact name is taken.

If the name already exists then the command completes without error, the name will not be duplicated. Note however that like all MQ objects the name is case sensitive and the check is treated as such.

6.12.2 =rmv()

The **=rmv()** keyword allows you to specify a list of names to be removed from the list. The name can contain wildcard, so for example, these commands are valid.

```
ALTER NAMELIST(MYNL) =rmv(*)           Removes all names
ALTER NAMELIST(MYNL) =rmv(*TEST*)      Removes all names containing test
```

Wildcards '*' and '?' can be used.

The case rules are different for the **=rmv()** keywords than for other MQ keywords. If the string is in quotes then an exact case match is required. If the string is not in quotes then an insensitive match is made. So the command example above will remove all list entries containing the characters 'TEST' regardless of their case.

If the name does not exist in the current list the command completes without error.

6.13 Search

Sometimes, after you've issued a command and seen the responses, it would be nice to be able to search the text for a particular string. MQSCX let's you do this easily using a special form of command starting with a '/' (forward slash)

For example the following command will search for the string 'q1'.

```
/q1
```

The search will be forwards through the entire output and it will be case insensitive.

There are times though where we would like to do something a little different and you can request these by adding one or more suffix characters to the command. The suffixes characters are listed below:

Character	Meaning
b	Perform the search backwards
c	Perform a case sensitive search
s	Search only a single command

So, for example, we can make the same search go backwards by adding a suffix 'b' on the command.

```
/q1/b
```

Note that you need to terminate the search string with a '/' character before adding the 'b' suffix.

We could make a case sensitive, backward search with a command such as this:

```
/q1/bc
```

It does not matter the order in which the 'b' or 'c' characters are specified.

Once you have registered a search string you can use it over and over again using the 'find next' or 'find previous' commands. These are assigned to the key strokes <Ctrl-f> and <Ctrl-g> respectively although they can, of course, be changed. The repeat find commands will use the same options as the last actual find command. For example 'find next' will always be in the same direction as the original find command and 'find previous' will be in the reverse direction.

The starting position for the search will be the current cursor position if it lies in the output text. If it is in the command line then the starting position will either be the start or end of the data depending on the search direction.

Suppose you've just received the response to a command and want to search for a string but only in the response of that last command. You can issue a command something like:

```
/q1/s
```

This command will start the search from the start of the last command and search both the command and all of its responses.

7 Control Language

MQSCX undoubtedly adds some very useful features to the MQSC commands. It adds more sophisticated filtering and some great ease of use capabilities, especially in interactive mode. However, as things stand it is still limited to just issuing a set of MQSC commands against the command server. All this is about to change and this opens up a whole new world of possibilities. MQSCX can now process the responses from the command server and change it's processing accordingly. Don't worry if you are not a programmer....this is all very easy.

7.1 Getting started with the control language

So, let's start simple. Suppose I just want to issue something to the screen. Well I have the print command. So, type in the following command and press enter.

```
print "Hello World"
```

We see that **MQSCX** responds, not surprisingly with what we asked it to print. Suppose we give it something a little more complicated.

```
print 4 * 5
```

We can see that **MQSCX** treating what it has been given as as expression to be evaluated.

So, can we print out something we have received from the Queue Manager. Suppose we want to print out all our channel names, how could we do that ? Well, try typing in the following but make sure you type it just as shown (better yet use copy/paste[8]).

```
foreach(DISPLAY CHANNEL(*)) print CHANNEL; endfor
```

So, we run this command and we see that **MQSCX** does indeed write out all the channel names but we also get lots of other stuff. Well, these are the commands that **MQSCX** is executing under the covers. Normally these wouldn't be shown but they can be very handy to see it going on. We can suppress them using the command

```
=echo langcon(no)
```

This command basically says don't echo control language lines even when entered from the console. So, if we issue the foreach command again.

```
foreach(DISPLAY CHANNEL(*)) print CHANNEL; endfor
```

We now see that just the channel names are displayed. However, suppose we also want to see the channel type. How would we do that ? Well, use command recall and add 'type' to the print statement.

```
foreach(DISPLAY CHANNEL(*)) print CHANNEL,TYPE; endfor
```

Issue this command and we see that we now have the channel name and channel type but it all looks a little messy. We would probably like it in columns.

So, let's add a format specifier, :20:, to the channel field in the print command. There are a number of different format specifiers you can add, which will be described later, this one just says “make the width of this field 20 characters”.

```
foreach(DISPLAY CHANNEL(*)) print :20:CHANNEL,TYPE; endfor
```

[8]You have to be a little careful with copy and paste, especially when the text contains double quote characters. Unfortunately there are different flavours of double quote characters using in documents, you need to ensure that you use the standard one for programming.

This looks a lot neater. We could of course continue this and add parameters and change the formatting but I think you get the idea. It would be better perhaps to discuss what is happening since you have taken it all on faith so far.

The first thing you would have noticed is the 'foreach' command. This is a very simple command which takes some form of MQ DISPLAY command as a parameter and will execute the sequence of instructions up to it's corresponding 'endfor' statement for each response it receives from the command server.

The responses themselves are not echoed to the screen but can be shown by issuing a **`=set noecho(yes)`** command.

So, that gives you an idea of how simple it is to process the response we get from the command server but we have clearly only scratched the surface of what you can do with **MQSCX** command files. To learn more you now have a choice, you can either continue reading here and we shall go through the language and language constructs. Or you could jump to Examples and go through a few more worked examples.

7.2 Variables

Perhaps the mainstay of any programming language, variables allow us store, retrieve, calculate and compare values. **MQSCX** supports three types of variables, user, system and response. We will consider each of these in turn.

7.2.1 User Variables

User variables are variables which have, for the most part, been defined by control program itself. They are the only variables for which you can change their value. The key thing about user variables is that they always start with an '@' characters. The variables can contain any type of data, strings, integers, lists or real numbers. All of the following are valid.

```
@a = 1
@a = "This is a test string"
@a = 3.1415
```

It is not necessary to define a variable before use you just use them whenever you wish. An easy way to check the value of a variable is to just print it to the screen.

```
print @a
```

Note that if we try and print a variable that does not exists yet we get an error message

```
print @xxx
Error Message: Variable '@xxx' is not defined in the current scope.
```

There are exceptions to this and those are environment variables. If a variable has not been defined by the program then **MQSCX** will look to see whether there is an environment variable of that name, Try this:

```
print @temp
```

The chances are that you have a 'temp' environment variable so the print statement will have printed it's value. Now you can override the value will something like this.

```
@temp = "My Value"
```

If you do this then you have created an in program version of a @temp variable. You have not changed the temp environment variable itself. So, the program need not worry about accidentally using a name that has an environment variable. However, environment variables can be a useful way to effectively pass parameters into your command files. For example, you could specify the path to a file or perhaps switch on a debugging flag.

There are some rules about naming of user variables

- The name is limited to 30 characters, which includes the '@' character.
- The character following the '@' must be alphabetic or the underscore '_' character.
  It is recommended that you avoid calling your user variables starting with @_ since these may be used by the MQSCX program itself.
- Following characters can be alphanumeric, '.' or '_'
- User variables are case sensitive so @a and @A are different variables.

When a variable is defined, or first used, then it is defined in the current stack frame. The variable is available in the current stack frame or any higher stack frames. Any variables defined at the lowest stack frame will remain until the program ends unless it is explicitly deleted using the **`delvar()`** function, see Expression Functions for a description of available functions. However, variables created in functions, i.e. Higher stack frames, will be deleted when the function returns. Please see Variable Scope and Stack Frames on page 40 for more information on this topic.

7.2.2 Arrays

There are times when you want to store large amounts of data. For example, suppose you wish to read all the queue definitions and store the values. It would not be feasible to define a different variable for each value. So, **MQSCX** allows you to use arrays. An array is essentially a user variable with a subscript. The following are all valid uses of an array:

```
@var[5]                 - the 5th element of array @var
@queue[6,10,2]          - multiple subscripts can be used
@mult[3,5,6,1]          - up to four subscripts can be used
```

It is not necessary to define the array before use. Arrays are sparse. This means that just because element @var[5] is defined it doesn't necessarily mean that element @var[4] is defined. Arrays are indexed from 1. Array element @var[0] is not valid. Each element of the array can contain a different data type. For example the following sequence is perfectly valid.

```
@val[1] = "Value"
@val[2] = 24
@val[3] = "Average"
@val[4] = 7.8
```

Once a variable is defined it remains until the program ends unless it is explicitly deleted using the **`delvar()`** function, see Expression Functions for a description of available functions. The **`delvar()`** can be used to delete the entire array or just a single element.

7.2.3 System Variables

System variable are special variables which are provided by **MQSCX** itself. Their value can not be changed by an assignment. System variables all start with the underscore '_' character.

The following system variables are defined:

Name	Value
_ccdt	The name of the current CCDT file. This system variable is not available on z/OS.
_ccdtmode	Boolean indicating whether **MQSCX** is currently in CCDT mode.
_client	Boolean indicating whether **MQSCX** is currently connected as a client or not.
_cmdok	Boolean indicating whether the previous MQ command was successful.
_cmdlevel	The integer command level of the Queue Manager **MQSCX** is administering. A value of -1 is returned if **MQSCX** is not currently connected.

_connqmgr	The name of the Queue Manager **MQSCX** is currently connected to. This will differ from system variable _qmgr if you are connected in via mode.
_errno	The current system errno.
_errnostr	A brief description of the current system errno.
_height	The height of the current window or zero if in file mode.
_idxEach	Only meaningful inside a foreach(...) loop. It contains the index of the object which is being processed. Outside of a foreach(...) loop the variable has the value 0.
_idxItem	Only meaningful inside a foritem(...) loop. It contains the index of the item which is being processed. Outside of a foritem(...) loop the variable has the value 0.
_idxWhile	Only meaningful inside a while(...) loop. It contains the iteration number of the loop. Outside of a while(...) loop the variable has the value 0.
_item	Used in a foritem(...) endfor loop. It contains the current list item value.
_lastrc	The last MQ reason code. For example from an =conn command If **MQSCX** is not currently connected it will return MQRC_NOT_CONNECTED (6124)
_lastrcstr	A brief text description of the last MQ reason code above.
_lastcmdresp	Related to the _lastresp value below. This variable contains the error message associated with the last command issued. For example, “AMQ8147E: IBM MQ object XXXX not found.”
_lastresp	The last response from an `DISPLAY` command For example, “QUEUE(Q1) TYPE(QLOCAL) CRDATE(25122013).....”
_lic_cn	The licence contact name.
_lic_em	The licence email address.
_lic_lc	The licence licensee value.
_lic_rm	The number of days remaining on the licence.
_matches	The number of responses which matched on the last `DISPLAY` command This will differ from _responses if you have some form of **MQSCX** filtering such as `=WHERE()` or `=FIND()`.
_mqevcmdlevel	The integer command level of the **MQEV** processor **MQSCX** is administering. A value of 0 indicates that **MQSCX** has not yet contacted **MQEV**.
_monacmdlevel	Returns the command level of the **MQMONA** program running at the target or 0 if **MQMONA** is not active.
_nl	Newline character. When printed this string will cause a new line.
_numEach	After processing a foreach(...) loop this variable contains the number of iterations of the loop.
_numItem	After processing a foritem(...) loop this variable contains the number of iterations of the loop.
_numWhile	After processing a while(...) loop this variable contains the number of iterations of the loop.
_os	The Operating System **MQSCX** is running on. Possible values are: “AIX”,”LINUX”,”PLINUX”, “WINDOWS” or “MVS”
_platform	A string containing the platform of the Queue Manager **MQSCX** is administering. A value of “NOTCONNECTED” is returned if **MQSCX** is not currently connected.
_qmgr	The name of the Queue Manager **MQSCX** is currently administering
_responses	The number of responses from the command server for the last `DISPLAY` command
_respwait	The current value of the *respwait* setting Please see the description of the =set command on page 130 for a description of respwait.

_sep	Separator. When printed this string will cause a separator line to be written.
_time	The current time in seconds from January 1st 1970.
_width	The width of the current window or width setting if in file mode.

7.2.4 Constant Variables

Constant variables allow you to use a text string to refer to standard MQ constants.

For example:

```
=conn qm(MQG1)
if (_lastrc = const.MQRC_Q_MGR_NOT_AVAILABLE)
  print "Try again later"
endif
```

If there are constants that are not included in this that you would like added, please get in touch.

7.2.5 Response Variables

Response variables are the simplest way of processing responses from a DISPLAY command. A response from a DISPLAY command might look like this

```
QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE)     TYPE(QLOCAL)    CURDEPTH(0)
```

This response will automatically define three response variables, **QUEUE**, **TYPE** and **CURDEPTH**. The values of the response variables will be values contained between the brackets. A response variable will always exist but they may have zero length if not returned by the command itself. This is helpful when writing scripts which are to be MQ version independent. You can specify any name, for example one that hasn't been defined yet on this Queue Manager, and MQSCX will just return the empty string if it is not returned on the command. It does however mean that you have to be careful about how you spell the variable since spelling mistakes will not be reported.

Response variable are case insensitive, so although the command server returns the field in upper case it quite all right to refer to them in lower case.

For example if we printed out the values we would see the following.

```
print QUEUE,TYPE,CURDEPTH
SYSTEM.DEFAULT.LOCAL.QUEUE QLOCAL 0
```

Response variables are over written at the next MQ command. Therefore if you need to save the value of a response variable you should assign it to a user variable. For example:

```
@queue = QUEUE
@depth = CURDEPTH
```

As in the **=WHERE** clause indicator fields can be referenced using the suffixes .short' and '.long'.

For example:

```
print qtime.short, qtime.long
```

There are some responses which can not be accessed via response variables.

Consider the following:

```
DISPLAY QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE) TRIGGER
QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE) TYPE(QLOCAL) NOTRIGGER
```

Unfortunately MQSC does not respond with something like **TRIGGER(YES)** or **TRIGGER(NO)**. Instead it returns the single string **NOTRIGGER**. If you really need to know whether the queue returned **TRIGGER** or **NOTRIGGER** you can use processing like this.

```
if (findstr(_lastresp,"NOTRIGGER")) @trig = "NO"; else @trig = "YES"; endif
print @trig
```

This code fragment uses a system variable that we mentioned in the previous section, '_lastresp'. _lastresp contains the whole of the response from the last DISPLAY command[9]. So, if we search for the word “NOTRIGGER” in that response we can determine whether triggering is enabled or not and set our user variable accordingly.

Perhaps the most common use of response variables are in a **foreach()** clause. We have already seen a few examples in the introduction, we just didn't know they were called response variables.

Consider the following.

```
foreach(DISPLAY QUEUE(*))
  print queue,curdepth
endfor
```

In a **foreach()** clause the sequence of commands contained in the **foreach()** block is invoked for every response to the DISPLAY command. So, naturally any response variables contained in the response are available during the loop execution.

7.3 Variable Scope and Stack Frames

All user variables belong to a particular stack frame. When a user variable is defined or first used it is created in the current stack frame. The current stack frame is merely the current level of the stack. For example, when the program starts you are at stack frame 0. If the program now invokes a function you are now at stack frame 1. If that function invokes another function you would be at stack frame 2 and so on. Each of these stack frames can have user variables defined. When execution reached the end of a stack frame then any variables owned by that stack frame will be deleted. Consider the following example:

```
func bar(a)
  print @a
endfunc

func foo(a)
  bar(@a+1)
  print @a
endfunc

@a = 1;
foo(@a+1)
print @a
```

This results in the output:

```
3
2
1
```

So, even though the program only ever prints out the value of `@a` we get three different values. This demonstrates that we can have multiple variables each with the same name at the different stack levels. As you can see, this is achieved because each of the functions specified that 'a' is a parameter. By definition parameters are variables local to the current stack frame. However, suppose we didn't define a variable in the functions, would it be visible ?

[9]Provided that no other MQ commands have been issued since the DISPLAY response

Let's try the following:

```
func bar(a)
  print @a,@b
endfunc

func foo(a)
  bar(@a+1)
  print @a,@b
endfunc

@a = 1;
@b = 2;
foo(@a+1)
print @a,@b
```

We've just added a new variable, `@b`, and given it the value of 2 in the main program. The functions make no declaration of `@b`, they just print out the value.

As suspected the output is now:

```
3 2
2 2
1 2
```

When a function accesses a variable it finds the definition with the nearest stack frame. So, if we updated the value of `@b` in function `foo()` would the change be reflected in the main program? Let's try it:

```
func bar(a)
  print @a,@b
endfunc

func foo(a)
  @b = 3
  bar(@a+1)
  print @a,@b
endfunc

@a = 1;
@b = 2;
foo(@a+1)
print @a,@b
```

The output is:

```
3 3
2 3
1 3
```

Yes!, the updated value is seen by everyone. Perhaps this is not a surprise since there is only one actual variable `@b`. However, suppose now I wanted function `foo()` to have it's own variable `@b`, to ensure that it couldn't mess up its callers value.

Well we can do that by adding a `var`.

```
func bar(a)
  print @a,@b
endfunc

func foo(a)
  var b
  @b = 3
  bar(@a+1)
  print @a,@b
endfunc

@a = 1;
@b = 2;
foo(@a+1)
print @a,@b
```

The output is now:

```
3 3
2 3
1 2
```

So, now we see that both `foo()` and `bar()` see `@b` as having the value 3 but when we drop back to the main program it sees `@b` as having the original value 2.

The notion of stack variables is very common among programming languages and you should find it fairly intuitive. The only aspect that is somewhat unusual is that variables can be found anywhere up the stack. Most languages have the concept of local and global variables. By defining variables in a function **MQSCX** allows you to define values which are partially global, i.e. only visible to functions further up the stack.

7.4 Expressions

Expression in **MQSCX** follow the normal convention in term of operator precedence. A full list of the operators are given in Appendix A. Expression Operators on page 152.

7.4.1 Data Types

MQSCX understands the following data types

- String
  A string can be treated as a list using `foritem(...)` clause.
- Integer
- Real

7.4.2 Coercion

In general data types are coerced automatically so that an expression can contain operands of different types without problem. Perhaps the most surprising and yet useful coercion is when a string is used as a number. Consider the following expression.

```
Print +"Hello World"
```

Here we have an arithmetic operator '+' and a string operand. **MQSCX** could have disallowed this combination but instead it coerces the string into a number, and that number is the string length. So, the expression about will print out 11. This is more useful than you might think. Consider the following example:

```
if (channel) print "Client connection"; endif
```

Here we are using a string value as the expression in a boolean field. The if() clause wants a TRUE or FALSE result and yet we are passing it a string. However, since a string is coerced to a numeric value by virtue of it's length we get the desired result. In other words, the if expression will be TRUE if the channel variable has a value and FALSE if it has no length.

However, there are some combinations of operand and operators which are not allowed. For example consider the following expression.

```
print "Hello" / "World"
```

This really doesn't make any sense. Trying to divide one string by another has no real meaning so MQSCX will report an error.

7.4.3 String Concatenation

One common operation you may wish to perform is concatenating strings. This can be achieve very simply using the '+' operator. For example:

```
@str = "Hello" + "World"
print @str
HelloWorld
```

Note that **MQSCX** concatenates the string exactly, no spacing character is added. You need to add that to the original strings if required.

However, now consider the following:

```
@depth = 50
@str      = "CURDEPTH(" + @depth + ")"
print @str
61
```

61! This may have seemed a very strange result until you remember our discussion previously about coercion. Remember that if you use a string and a number in the same expression that the string length will be used in the calculation. What we need is a way of converting the number into a string. And luckily there is a very simple function `str()`. So, let's try this:

```
@depth = 50
@str      = "CURDEPTH(" + str(@depth) + ")"
print @str
CURDEPTH(50)
```

Great, that's just what we wanted. The `str()` function can be used on any data type.

7.5 Inserting code fragments

It is possible to add extra code into the command stream. Consider the following:

```
@cmd = "DISPLAY QUEUE(SYSTEM.DEFAULT.LOCAL.QUEUE)"
@cmd
```

The `@cmd` statement will actually run the command that is contained in the variable. This may not be too surprising but it is actually more powerful than it first appears. Now consider the following:

```
@cmd = "if (curdepth) print 'Queue is not empty'; endif"
@cmd
```

This demonstrates two things.

1. **There is no limit on the commands which can be inserted in this manner.**
   The control program can construct the command at run time and then execute them; a self modifying program if that is useful.

2. **This examples shows how to insert quotes characters inside a string**
   To identify a string in an expression you can use either ' or “ characters. The string is ended when that character next appears in the string. So, to write a string inside another string all you need do is use the other quote character.

7.6 Substitution commands

Substitution commands are a simple way of inserting a variable into command. Suppose we wish to display a queue but want the queue name to come from a variable. We could do the following:

```
@q = "SYSTEM.DEFAULT.LOCAL.QUEUE"
@cmd = "DISPLAY QUEUE(" + @q + ")"
@cmd
```

This works just fine and could well be the way to do it in some cases. However, using substitutions we can use a short cut. Look at the following:

```
@q = "SYSTEM.DEFAULT.LOCAL.QUEUE"
DISPLAY QUEUE(<@q>)
```

This will generate the same command. Essentially **MQSCX** looks for a variable between <> characters. If it finds either a user or system variable between the <> characters then the variable is replaced by the value of the variable. The substitution isn't limited to a simple variable, arrays such as the following will also work:

```
DISPLAY QUEUE(<@Names[@index]>)
```

Note that this substitution will only happen when the command is about to be run. It will not happen in an assignment as is demonstrated by the following:

```
@q   = "SYSTEM.DEFAULT.LOCAL.QUEUE"
@cmd = "DISPLAY QUEUE(<@q>)"
print @cmd
DISPLAY QUEUE(<@q>)
```

This means that you can easily insert code fragments containing substitutions.

7.6.1 Functions

There are a number of functions defined which can be used in expressions. For example, functions are available for calculating dates, opening files, finding substrings and many more. Please refer to Expression Functions on page 154 for the complete list.

In addition you can write your own function. Please refer to Functions on page 54 for more information.

7.7 Running the control commands

As was mentioned before the most convenient way of running a set of control commands is to have them in a file and import them when necessary. This means you can use your favourite editor to edit the command file to get it just how you want it. To run the file you can just say something like:

```
=import file(d:\mqscx\age.mqx[10])
```

[10]Of course the path to your control file is likely to be different

However, this is a lot of typing. Fortunately **MQSCX** gives us synonyms. So, let's use those. Type in the command.

```
=syn(age) cmd(=import file(d:\mqscx\age.mqx))
```

Now all we need to do is type **age** every time we want to see the ages of all the queues.

However, we can go one step further. It is often the case that you want to issue a set of control commands against a particular object rather than the whole Queue Manager. As an example suppose we wish to have a simple command, let's say **qinfo,** which displays data about a given queue. Let's assume we want to know it's depth and which processes are currently using it.

The first step would be to write a set of commands, let's call it ***qinfo.mqx***. To save you the typing it the a set of samples are available as a separate download.

```
* Check we've been given a parameter
if (!exists(@_Parm[1]))
  print "Please enter a queue name, something like qinfo(Q1)"
  leave
endif
* Switch off command echoes
=echo cmds(no) resp(no)

* The parameter is the queue name
@q = @_Parm[1]

* Issue a display on that name
DISPLAY QUEUE(<@q>) ALL

if (_matches<1)
  * Did we get an answer ?
  @msg = "Unknown queue '" + @q + "'"
  print @msg
else
  * So, we had an answer
  @bq = @q

  * Construct an output and print it out
  @msg = "QUEUE(" + QUEUE + ")"
  if (ipprocs) @msg = @msg + " IPPROCS(" + str(IPPROCS) + ")"; endif
  if (opprocs) @msg = @msg + " OPPROCS(" + str(OPPROCS) + ")"; endif
  if (type = "QLOCAL") @msg = @msg + " CURDEPTH(" + str(CURDEPTH) + ")"; endif
  print @msg
  if (target)
    @bq = target;
    print "Alias queue - displaying usage of target",@bq
  endif

  * Now, let's see whether anyone is using the queue
  foreach(DISPLAY QSTATUS(<@bq>) TYPE(HANDLE) ALL)
    if (channel)
      print :7:PID,APPLTAG,"over",CHANNEL,"from",CONNAME
    else
      print :7:PID,APPLTAG
    endif
  endfor
  if (!_matches) print "No users found."; endif
endif
```

At first glance this may seem complicated but it's actually very simple.

The main difference and the reason we are introducing it here is the first few lines.

```
if (!exists(@_Parm[1]))
  print "Please enter a queue name, something like qinfo(Q1)"
  leave
endif
```

What this is saying is “if the variable element `@_Parm[1]` does not exist” write out an error message. So, how does the array `@_Parm[]` get defined. Well, that is another trick of the **`=syn`** command. When you enter a synonym **MQSCX** will check to see whether you follow it with any parameters. For example,

```
qinfo(a,b,c,d,e)
```

In this case `@_Parm[1]` is set to “a”, `@_Parm[2]` is set to “b”, `@_Parm[3]` is set to “c”, and so on. Essentially we have a simple way of passing parameters to a set of control commands. However, of course you can not guarantee that the user will type a parameter so it is sensible to check whether the parameter exists before you use it. In this case we have chosen to fail the command if no parameter is specified. However, you could just as easily use some default value. The statement 'leave' is quite useful here. It causes execution of the current file to stop and control is passed back to whatever invoked the control file.

Since we are here let's just explain one or two concepts in the control commands which may worth mentioning. The first thing to mention is the line:

```
  @q = @_Parm[1]
```

It is always a good idea to save the parameters in user variables before you do anything else. After all you may wish to call further synonyms which will overwrite your values. Secondly it makes your code more descriptive. The last reason is that you can do some things to a user variable which you can't do to an array. Consider the line:

```
foreach(DISPLAY QSTATUS(<@q>) TYPE(HANDLE) ALL)
```

This is using the provided parameter in a substitution command. This would not work if we had specified `<@_Parm[1]>` since arrays can not be specified between the <> characters.

The last thing we want to mention are the lines

```
DISPLAY QUEUE(<@q>) ALL
if (_matches<1)
```

This is a very simple command sequence. Essentially you issue a DISPLAY command and then check whether you have received a response. The value of system variable `_matches` will give the number of responses to the last DISPLAY command.

7.8 General syntax

Commands can be entered anywhere on the line and white space is ignored. So, the following are all valid and equivalent:

```
@a=3
@a  =   3
    @a  =   3;
```

The last examples shows that an end of statement character is optional. This is really just a matter of style whether you like to add the semicolons or not. These are end of statement markers. Where they are really useful is if you want to enter multiple commands on the same line like this.

```
@a = 3; @b = 4; @c = 5
```

Here we enter three statements on the same line. In the last one we can omit the end of statement indicator since the end of the line will essentially do that for us.

7.8.1 Continuation

There are times when we might want to have single statement span a single line. To do this we use the same syntax as MQSC commands and have a character at the end of the line which indicates 'continuation'. There are two ways of signifying continuation.

1. **If the line ends in a '+'**
   The following line is concatenated starting at the first non-blank characters
2. **If the line ends in a '-'**
   The following line is concatenated starting at the first character

7.8.2 Comments

Any line where the first non-blank character is an asterisk '*' is considered a comment.

7.9 Statements

7.9.1 break

The break statement can only be issued from within a loop. The loop can be a while(...) clause, a foritem(....) clause or a foreach(....) clause. It causes execution to continue at the statement following the end of the current loop.

7.9.2 continue

The continue statement can only be issued from within a loop. The loop can be a while(...) clause, a foritem(....) clause or a foreach(....) clause. It causes execution to continue at the end of the current loop. Essentially this is a short-cut way of processing the next iteration of the loop.

7.9.3 foreach(....) clause

The `foreach(..)` clause is the way the responses to a DISPLAY or RESET QSTATS command are processed. The syntax is as follows:

```
foreach( <DISPLAY COMMAND> )
   <control statements>
endfor
```

You can specify any MQSC DISPLAY command or RESET QSTATS command in the brackets immediately following the `foreach` word. The command must be a single MQ command. The set of statements between the `foreach(....)` clause and the `endfor` statement will be run for each response from the command server. The statements will not start processing until all responses have been received from the command server. `foreach(....)` clauses can be nested which means that you can query other MQ objects based on the responses you receive from the first `foreach` clause.

The MQ command and the responses are not echoed to the screen. However, if you wish to see the commands, say for debugging purposes, you can display non-echoed lines using the command =show noecho(yes).

There are two other statements, break and continue, which can control the flow of the loop. In addition, if the while loop is in a function you can use return and goto to break out of the loop. Note that you can use goto to break out of a loop but you can not use goto to jump in to a loop.

7.9.4 foritem(....) clause

The foritem(....) clause provides a simple way to process a list. A list it just a simple comma separated string.

The syntax is as follows:

```
foritem( <String> )
   <control statements>
endfor
```

As an example consider the following:

```
foritem("a,b,c")
  print _item
endfor
```

When you run this set of commands MQSCX will print out the values “a”, “b” and then “c”. It will run the sequence for each item in the list and put the current item in the system variable **_item**. The item will be stripped of any whitespace characters, such as spaces. For example the statements:

```
foritem("   a   ,  b ,  c  ")
  print _item
endfor
```

will yield the same results.
The foritem(...) clause is particularly useful when processing lists in MQ objects. For example, take a look at the namelist example namelists.mqx.

```
foreach(DISPLAY NAMELIST(*) NAMES)
  foritem(NAMES)
    print _item
  endfor
```

```
endfor
```

This very simple sequence of commands will print out the names contain in all of the defined namelists. Of course it isn't just namelists that use lists. Channel exits, group names, connection names, channel auth user lists and more all use lists of values.

There are two other statements, break and continue, which can control the flow of the loop. In addition, if the while loop is in a function you can use return and goto to break out of the loop. Note that you can use goto to break out of a loop but you can not use goto to jump in to a loop.

7.9.5 fprint statement

The fprint statement allows data to be written to a file. The statement is the same as the print statement we are already familiar with except it has an initial parameter which identifies the file to write to. So, suppose we wish to take the namelists example in the previous section and write the namelist names to a file rather than to the screen. We could modify the code as follows:

```
@hf = fopen("c:\temp\namelist.txt","w")
foreach(DISPLAY NAMELIST(*) NAMES)
  foritem(names)
    fprint @hf,_item
  endfor
endfor
fclose(@hf)
```

You can see that very easily we can generate a file containing whatever MQ objects or values we choose. Essentially the only changes required are to add a call to the function **fopen()** to open a file to write to. The fopen function returns a file identifier which is then passed to all subsequent fprint statements. Once writing to the file is complete the file is closed with a call to **fclose()**. Any file handles created will remain open until explicitly closed by a call to fclose() or until the MQSCX program is ended. For a description of the formatting that is available please see the print statement description.

7.9.6 goto

The goto statement allows you to jump execution to a predefined label. Goto statements can only be used in functions.The syntax is:

```
goto <label name>
```

The label name can be up to 30 characters. It must start with an alphabetic character but this can followed by any alphanumeric character plus the '.' and '_' characters.

Many proponents of structured programming question the purity of using goto statements since they can make programs difficult to read and maintain. However, when used sparingly that can greatly simplify coding and actually improve readability. For example, one of the common uses it have a goto statement which branches to the end of a function in case of failure.

For example:

```
func foo(x)
  connect()
  if (_lastrc != 0) goto MOD_EXIT; endif
  … …
  … …
label MOD_EXIT;
  ;
```

```
endfunc
```

With a goto statement you can jump either forward or backward in a function. However you can not jump in to loops. You can jump out of a loop, but not jump in to one.

7.9.7 if(....) clause

The if(...) clause is the control language way of providing conditional execution.

There are two forms of the if(....) clause depending on whether an 'else' clause is required.

```
if( <Boolean expression> )
   <control statements>
endif


if( <Boolean expression> )
   <control statements>
else
   <control statements>
endif
```

The boolean expression can be any expression which results in a TRUE or FALSE result. The result is considered TRUE if the result is non-zero. The following are all valid if clauses:

```
if (curdepth > 100)              Depth of queue greater than some value
if (conname)                     Connection name is non-blank
if (!(queue == "SYSTEM.*"))      Not a system queue
if (exists(@option))             User Variable @option exists
if (findstr(descr,"test"))       Has "test" in the description
```

7.9.8 label

The label statement allows you to define a point that can be jumped to from elsewhere in the function. Labels can only be defined in functions.

The syntax is:

```
label <label name>
```

The label name can be up to 30 characters. It must start with an alphabetic character but this can followed by any alphanumeric character plus the '.' and '_' characters.

You can have as many labels in a function as you like but you should not define the same label in multiple places.

7.9.9 leave

The leave statement causes execution to leave the current file. Control is passed back to the calling code. So, imagine that you had a control file A which imported another control file B. If a leave statement was executed in control file B then the next statement executed would be the statement immediately following the the =import file(B) command in file A.

The leave statement is useful when detecting error conditions and can prevent a large build up of if-then-else type processing. The leave statement can not be used in functions. If you wish to return prematurely from a function use the return statement.

7.9.10 print statement

The print statement is the general way in which you can output data.

The syntax is as follows:

```
print [:<format string>:] <print item> {,[:<format string>:] <print item> }
```

Essentially it is a comma separated list of items to be displayed. The items can be anything including strings, expressions or variables. Before each item you have the option to provide a format string which can provide some finer control of what is displayed.

The format string can contain:

Value	Meaning
Initial Number	A number immediately following the first ':' gives the minimum width of this item
'p'<number>	A 'p' followed by a number gives the precision of the real number
'l'	Item should be left aligned By default items are aligned according to their type, numerical values are right aligned, string values are left aligned.
'n'	No space before this item
'r'	Item should be right aligned By default items are aligned according to their type, numerical values are right aligned, string values are left aligned.
's'	If added to the last item on the print statement it will suppress the newline. Nothing will appear on the screen until a print statement which writes a newline is entered.

So, in the tradition that examples are the best way of learning, here are a few:

```
print "a","b","c"               - print out 3 items
print :10:"a",:5:"b",:20:"c"    - print out 3 items in columns
print :10:"a",:5r:"b",:20:"c"   - as above but right align second column
print "a","b",:n:"c"            - no spacing on last item
print 1/7                       - print 1/7 th
print :p6:1/7                   - print 1/7 th to 6 decimal places
```

By default each print statement will result in a new line of output. However, the new line can be suppressed using the 's' formatting flag. Equally a new line can be forced in the print statement by printing the system variable _nl. In addition a separator line can be printed by printing the system variable _sep.

All of the formatting available for the print statement is also available in the fprint statement.

7.9.11 return

The return statement allows either a value to be returned from a function or return from a function to be executed before reaching its end. The return keyword can be optionally followed by the value that should be returned.
The syntax of the statement is:

```
return [<expression>]
```

The expression can resolve to data type. For example, consider the following function:

```
func plus(a,b)
  return @a+@b;
endfunc
```

This function, useful really for only demonstration purposes, will return the sum of the two parameters. For example if both parameters are integers then it would return the integer sum. If both parameters are real numbers then it would return the real sum. Further if both parameters are strings then it will return the concatenation of the two strings. This behaviour demonstrates that the type of the parameters and return value are not defined but handled at run time. Each invocation of the function could use different data types.

A function will **always** return a value. If function execution reaches the end of the function without reaching a return statement then the integer 0 will be returned.

7.9.12 var

The var statement allows the programmer to force variables to be defined in the current stack frame. It can only be used in a function and, traditionally, would be defined at the start of the function although that isn't enforced.
The syntax of the statement is:

```
var <variable>{,<variable>}
```

You can therefore define as many variables as you need at the start of your function. For example,

```
func foo()
  var x, y, z
  … …
  … …
endfunc
```

Note that arrays are not supported by the var statement.

Although the variables are defined they have no value. Any attempt to use the variables without assigning a value would result in a runtime 'variable not defined' error. If required you can use the function ***exists()*** to check whether a variable has a value.

Please see Variable Scope and Stack Frames for further information.

7.9.13 wait() statement

The purpose of the wait() statement is merely to add a delay in processing.

The syntax of the command is:

```
wait(<Delay in seconds>)
```

Imagine that you wanted monitor for a particular event. Let's say you wanted to check periodically the depth of some important queues. Let's say every 5 seconds you want to issue a DISPLAY QUEUE command and check whether any queues are over 80% of their maximum depth. Well the way you would do it is by inserting a wait in the command loop. For example:

```
@i = 0
while(1)
  foreach(DISPLAY QUEUE(*) TYPE(QLOCAL) CURDEPTH MAXDEPTH)
    if (CURDEPTH > (MAXDEPTH * 0.8))
      print "Queue",queue,"is getting full, depth is ",CURDEPTH
    endif
  endfor
  @i = @i + 1
  if (@i > 10) break; endif
  wait(5)
endwhile
print "Done"
```

Of course this example is just for demonstration purposes so the loop terminates after about a minute. And in practise you would probably want to do something else rather than just print to the screen. For example you could use the **system()** function to invoke a local command. The local command could do practically anything from sending you an email to starting further Queue Managers.

Using this technique it becomes fairly trivial to monitor any combination of conditions you can think of.

7.9.14 while(...) clause

The while(...) clause is the statement which is used to run a sequence of commands again and again.

```
while( <Boolean expression> )
   <control statements>
endwhile
```

The while(...) clause will loop continuously while ever the Boolean expression evaluates to TRUE. Clearly you need to be careful with the while() clause to ensure that the program doesn't loop forever unless that is what is required.

There are two other statements, break and continue, which can control the flow of the loop. In addition, if the while loop is in a function you can use return and goto to break out of the loop. Note that you can use goto to break out of a loop but you can not use goto to jump in to a loop.

7.10 Functions

Like most other programming languages MQSCX supports the definition of user functions. Functions allow the definition of a group of statements which can be called from anywhere in the program by referring to the function name. Parameters can be passed to the function if required. The syntax of a function is as follows:

```
func FunctionName([Parameter [, Parameter])
    … …
    … …
endfunc
```

7.10.1 Function Basics

There is no restriction on how long a function can be or what statements it contains. Functions can call other functions and even call themselves (known as recursion). They also always return a value either explicitly using the **return** statement or an implicit zero. So an example might be:

```
func greeting(name)
  print "Hello",@name
endfunc

greeting("Mary");
```

Here we have a very simple function called 'greeting' being defined and we see a call to it from the main line program. The function takes a single parameter which it calls 'name'. The values passed from the function invocation are assigned to these parameter variables when the function is run. This would result in the output:

```
Hello Mary
```

If we try to pass too many parameters MQSCX will report a syntax error explaining how many parameters are defined. However, MQSCX will always allow us to pass less than the defined number of parameters. This can be useful if you want the function to accept different numbers of parameters, perhaps for qualification of some kind. If the function tries to reference a parameter which has not been passed in then it will generate a runtime error. If you want to check whether a parameter has been passed in you can use the **exists()** function.

For example, we could modify our function like this:

```
func greeting(name)
  if (exists(@name))
    print "Hello",@name;
  else
    print "Who are you?"
  endif
endfunc
```

Now if we call the function without passing in a parameter the function responds with:

```
Who are you?
```

Whether you protect the parameters or not probably depends on whether you expect the function to be called from the command line or not. Functions can be a great way of augmenting the functionality of the command interface in MQSC. MQSCX already has synonyms which allow you to assign a 'command name' to a MQ command. Functions allow you to go one step further and assign some logic to the sequence of commands.

Functions also allow you to return a value. So, in this example, we could decide to return the greeting and have the caller decide how to print it out. So, we would have something like this:

```
func greeting(name)
  if (exists(@name))
    return "Hello " + @name;
  else
    return "Who are you?"
  endif
endfunc

print ">",greeting("Mary")
```

Now if we call the function we get given a string back and we can print it out however we wish:

```
> Hello Mary
```

7.10.2 Function Definitions

Although you could type in your functions at the command line it is far easier to write your functions in an editor and a script file and then import the file. Any functions contained in a file that is imported will remain after the import is completed. In fact, once a function has been defined it will remain for the life of the MQSCX program. You can overwrite the function definition just by importing a file with a new definition of the function.

If, for whatever reason, there is a syntax failure in the new function definition then an error message will be issued and the original function definition will remain active. It is only possible to have one definition of any one function and they all have global scope. For this reason it is recommended that you give your functions reasonable names to avoid name clashes.

It can be useful to have certain functions always available. They can act as short-cut commands or they can be used by other function definitions. If it is desirable to have certain functions always available then you can put the definitions in the Bootstrap file described on page 59.

7.10.3 Function Invocation

Functions can be invoked in three different ways:

- From the command line
- In any expression.
- From other functions or an imported command stream

The only rule that should be followed is that the definition of the function must be registered before a call is made to the functions.

Calling a function from the command line can be very useful. Essentially it allows you to write a piece of logic which will be invoked by typing a simple word on the command line. In some ways they are similar to synonyms plus functions allow us to make use of parameters.

Calling a function from another function is perfectly straight forward however you must remember the rule above that the called function should be defined before the function that makes the call. However, there are a couple of special cases which we will discuss now.

Functions can be defined with up to 20 parameters. These parameters can accept any data type except arrays. An array can be defined in a function or it can be defined outside the function and referenced within it but you can't pass an array as an actual parameter.

7.10.3.1 Recursion

Occasionally it can be useful for a function to call itself, this is known as recursion. The classic example that is often used is to calculate factorial. (I'm not sure why since calculating a factorial would be far more efficient if done in a while loop.) However, it does demonstrate the principal fairly nicely.....

```
func fact(n)
  if (@n <= 1) return 1;
          else return @n * fact(@n-1)
  endif
endfunc
```

So, here we have a very simple function which returns the factorial of the number passed. It achieves this by calling itself if the number passed is greater than one. So, we can printout out the factorial of a number like this.

```
print fact(6)
```

Clearly one of the key things here is to ensure that the stack does unwind sooner or later. It is very easy to create infinitely recursive loops. When this happens, of course, the program will run out of stack and abnormally terminate.

7.10.3.2 Mutual Recursion

Mutual recursion is very similar to normal recursion but in this case we have two functions that each want to call each other. So, at first glance you might think that this would do the trick:

```
func flip(a)
  print "flip",@a
  if (@a > 0) flop(@a-1); endif
endfunc

func flop(a)
  print "flop",@a
  if (@a > 0) flip(@a-1); endif
endfunc
```

However, this breaks the golden rule that a function needs to be defined before it is used since flip() tries to call flop() before it has been defined. So, it would seem we are at an impasse. Well, not quite. Remember that you can define a function as many times as you like and each definition will overwrite the previous one. So, all we need to do is to make a minimal definition of flop() before we define flip(). It would look something like this :

```
func flop(a); endfunc

func flip(a)
  print "flip",@a
  if (@a > 0) flop(@a-1); endif
endfunc

func flop(a)
  print "flop",@a
  if (@a > 0) flip(@a-1); endif
endfunc
```

Now everything works just fine. You can consider the dummy definition of flop() as an indication to MQSCX that there will be a function definition coming later. Some languages refer to this type of thing as a *forward definition*.

7.10.4 Dynamic Execution

Sometimes it is useful to define a function that operates largely the same on each invocation but its behaviour can be modified dynamically depending on the input parameters. For example in C one might pass in a function pointer that has the desired dynamic behaviour. MQSCX does not have function pointers, it doesn't have pointers at all, However it can achieve a very similar effect using the '**eval**' function. The **eval** function is a very useful function, it will evaluate any given expression. This expression can be anything, including an invocation of a function.

Consider the following function:

```
************************************************************************
*                                                                      *
* FUNCTION: WaitForState                                               *
*           Wait for a specific state to occur                         *
*                                                                      *
* - If you don't supply the number of iterations to loop round waiting*
*   for the channel to be running, then it will try it up to 10 times.*
*                                                                      *
************************************************************************
func WaitForState(Command, Expression, Iterations)
  var rc; @rc = 0

  if (!exists(@Iterations)) @Iterations = 10; endif

  while (1)
    if (_idxWhile > @Iterations) @rc = -1; goto MOD_EXIT; endif
    wait(2)
    <@Command>
    if (eval(@Expression)) goto MOD_EXIT; endif
  endwhile
label MOD_EXIT
  return @rc
endfunc
```

This function uses a number of concepts but there are two things to notice.

- The use of substitution command to issue a command that is passed in
- The use of the eval() function to check the result of the command.

These two features means that this function is very dynamic. So, what would we use this function for? Well suppose we need a script that started a channel. We might well want to wait until the channel was running before continuing. We could now write something like this:

```
 START CHANNEL(<@ChannelName>)

 @rc = WaitForState("DISPLAY CHSTATUS(<@ChannelName>)", "_matches > 0")
```

It should be clear that the exact behaviour of the function can be changed enormously just by changing the Command and Expression parameters. The function can now be used for all sorts of things where you need to wait for some state in MQ to change.

Of course the concept of dynamic execution doesn't have to involve an MQ command, that is just an example of its use. The key thing is that a parameter can be passed to the **eval()** function. Since the **eval()** function will evaluate the expression, including calling functions, we now have a way of effectively passing a function pointer into a function.

7.10.5 Comments

In most of the examples throughout this manual we have not included comments. Ironically this is for clarity so you can concentrate on the actual instructions rather than any additional descriptions we may have provided. After all, the text in this document describes the code. However, when you are writing **MQSCX** scripts we hope that you make liberal use of comments. In any programming language it is good practice to describe what is happening to make the code more maintainable. However, consider the following:

```
* MQSCX Example code written by Paul Clarke
*   taken from the MQSCX manual

* Function : foo
* Purpose  : Demonstrates the passing of optional parameters
func foo(name)
  if (exists(@name))
    print "Hello",@name;
  else
    print "Who are you?"
  endif
endfunc
```

Clearly any comments defined between the **func** and **endfunc** statements belong to the function `foo()` but it is common coding practice to put the description of the function immediately before the function definition, rather than inside it. So, **MQSCX** must make a decision on how many of the preceding comments to include as part of the function. The reason for this is that it is possible to display the contents of any loaded functions using the command:

```
=show func(*) list
```

Clearly it is useful for the function code to be preceded with any appropriate comments.

So, the method used by **MQSCX** is that all comments up to the second blank line will be include as part of the function definition.

```
  < Second blank line is function delimiter nothing prior to this included>
* Block comments included
* Block comments included

  < First set of optional blank lines included in function >

func foo(name)
  … …
endfunc
```

7.11 Invoking other programs from your script

There will be times in your scripts that you wish to invoke a program or command/shell script to do something outside of **MQSCX**, such as email or text someone, or run a program, for example you could use the Q program to send messages to a queue that might trigger other processes. To do this, you can use the **MQSCX** control language function `system()`. For more details of the syntax of this and other **MQSCX** control language functions, see Appendix B.Expression Functions on page 154.

Here's an example of using the Q program to put a message to a queue from an **MQSCX** function. The command string to invoke is built up using the `_qmgr` system variable rather than hard-coding the queue manager name,. Read more about System Variables in 7.2 Variables on page 36.

```
func PutMsgToQueue(qName)
  @CmdStr = "q -m" + _qmgr + " -o" + @qName + ' -M"some text"'
  system(@CmdStr, const.SYNC)
endfunc
```

Invocations using the `system()` function can be either synchronous or asynchronous, specified in the optional second parameter. If the second parameter is omitted, the default value is to run asynchronously in order to maintain behaviour from prior releases. Depending on the command you are issuing, it may be that synchronous mode is more appropriate.

7.11.1 Synchronously

Invocations made using the `system()` function in synchronous mode only return control to **MQSCX** once the command completes, so be careful not to use long running commands in this way.

Commands are invoked by the command processor (**`CMD`**) on Windows, and through the shell **`/bin/sh`** on other platforms, so redirection of output, say to a text file, is possible. Otherwise the output from the command may show up in the **MQSCX** output window in some environments.

Using synchronous mode may be useful when testing the script to ensure you have the correct command string since output is easier to see.

We could imagine extending our earlier example like this using the `_os` system variable to choose a temporary file to redirect our command output to. Note on z/OS, running a program using the **`sh`** shell, means redirection of output can only use an HFS file, not an MVS dataset.

```
func PutMsgToQueue(qName)
  @CmdStr = "q -m" + _qmgr + " -o" + @qName + ' -M"some text"'
  if (_os = "WINDOWS")
    @CmdStr = @CmdStr + "> c:\temp\Q.out 2>&1"
  else
    @CmdStr = @CmdStr + "> /u/gemuser/Q.out 2>&1"
  endif
  system(@CmdStr, const.SYNC)
endfunc
```

7.11.2 Asynchronously

Invocations made using the `system()` function in asynchronous mode, return control to **MQSCX** immediately, so if you have a long running command to invoke, you should perhaps use this mode.

```
  system(@CmdStr, const.ASYNC)
```

Redirection of output written to `stdout` or `stderr` is not possible in this mode.

7.12 Bootstrap file

It can be handy to have a set of statements run each time MQSCX is started. One of the common reasons for this is to register any function definitions, however it could also be useful to run some initial queries against your Queue Managers. For this purpose MQSCX allows the user to specify a bootstrap file. By default **MQSCX** will look for, and run, a file call **bootstrap.mqx** in the same directory as the configuration file, or a member called ***BOOTSTRP*** in the same PDS(E) as the configuration file. If this file is not there then **MQSCX** will not generate an error since it is entirely optional.

If you wish to override this default name then you can define an environment variable **MQSCX_BOOTSTRAP** specifying the full path and file name of the file you wish **MQSCX** to run. In this case **MQSCX** will report an error if the file is not found or can't be opened.

7.13 Examples

You will find a few example scripts available for download at http://www.mqgem.com/mqscx_download.html. Most of these scripts are fairly self explanatory but if you are new to MQSCX if may be worth going through them to learn how they work.

7.13.1 Calculating the age of a queue (age.mqx)

```
* Prevent commands and responses from echoing
=echo cmds(no) resp(no)

@oldest  = 0
@oldestq = "";

* Get all of the queues
foreach(DISPLAY QLOCAL(*) CRDATE CRTIME RETINTVL)
  * Calculate age of queue in seconds
  @age     = _time - mqtime(CRDATE,CRTIME)

  * Convert seconds to hours
  @age_hours = @age / (60 * 60)

  * Convert seconds to days
  @age_days  = @age_hours / 24

  if (@age_days > @oldest)
    @oldest  = @age_days
    @oldestq = QUEUE
  endif

  * Check the retain interval
  if (RETINTVL != 999999999)
    * How long should we retain
    @retain = RETINTVL - @age_hours
    if (@retain > 0)
      print :48:queue,"has",:6p0:@retain,"hours left"
    else
      print :48:queue,"expired by",:p0:-@retain,"hours"
    endif
  endif
endfor

print "The oldest queue is",@oldestq,"at",:p0:@oldest,"days."
```

This could be typed in at the command prompt but that is probably not realistic. In reality the way to run control language commands is to have them in a file and then import them as necessary. Essentially this script is very similar to our very first example, it issues a DISPLAY command in a foreach statement and then runs a sequence of commands against the returned responses. However, it does introduce a few new concepts which we'll discuss now.

Firstly though let's all understand what it is trying to do. The purpose of the command file is to calculate the ages of all the local queues. It then compares the age to the RETINTVL setting for the queue. It prints out a message saying how long the queue should remain for or whether it should be deleted. Lastly the script prints out the name of the oldest queue.

So, we clearly we need to issue a DISPLAY Q(*) type command. Because we wish to know their ages we have to ask for the creation date and time to be returned, which are CRDATE and CRTIME respectively. We also need to know the RETINTVL value so we add that as well.

Now, inside the foreach() block we see the line

```
 @age = _time - mqtime(CRDATE,CRTIME)
```

This introduces the notions of variables. A variable is just a simple way of storing a value for something. A variable starting with a '@' is known as a user variable. These can be any name that is meaningful to you up to 30 characters. Another type of variable are those starting with '_'. These are known as system variables and are provided by MQSCX itself. You can not change the value of a system variable. Here we see the variable _time being used which, perhaps not surprisingly, is the current time in seconds[11].

We also see a call to something called mqtime(). mqtime() is a function provided by MQSCX which will convert a date and time into the number of seconds. By subtracting the queue time from the current time we can therefore determine the queues age in seconds. The following line divides this time by the number of seconds in an hour and we therefore have how old the queue is in hours. We also calculate the age of the queue in days.

```
if (@age_days > @oldest)
  @oldest  = @age_days
  @oldestq = QUEUE
endif
```

The following few lines are fairly self-explanatory but it shows that you can have conditional processing. Here we are saying that if the age of this queue is older than any queue we've seen so fair we remember its age and name.

```
if (RETINTVL != 999999999)
  * How long should we retain
  @retain = RETINTVL - @age_hours
  if (@retain > 0)
    print :48:queue,"has",:6p0:@retain,"hours left"
  else
    print :48:queue,"expired by",:p0:-@retain,"hours"
  endif
endif
```

We then check the value of RETINTVL for this queue. If it doesn't have the 'infinite' setting then we check to see how this retention interval compares with the current age of the queue. The queue can either have a certain time to live or it should already have been deleted. We print out a message for whichever of these situations is true for this queue. The print statement is not new to you but the formatting option p0 might be. All this does it set the precision of the @retain parameter. Since @retain comes from @agehours which is essentially the result of a divide instruction it will be a real number. Something like, 123.45546. This looks rather ugly and the decimal places don't contribute very much so by specifying a precision of 0 we tell the print instruction not to bother to output the decimal places at all.

The last thing the script does is print out the oldest queue just for good measure.

What this script shows us is how easy it is to process the results from the command server, perform calculations and then display the results. Clearly we could extend this script to include much more checking. From a usability point of view perhaps one change that would be nice is to print out the queues in the order they should expire. What we could do with is a nice simple way of sorting data. And that leads us nicely to our next example.

[11]As is common in these circumstances the time is the number of seconds since January 1st, 1970.

7.13.2 Sorting data (sort.mqx)

Since we have the ability in the MQSCX language to perform calculations, comparisons and make assignments we could clear write some instructions to do some sorting, perhaps a simple bubble sort. However, there is a useful sort() function which we shall demonstrate now.

Consider the following code:

```
* Command file to show the effect sort() function on an array
delvar(@a)

@a[ 1] = "that"
@a[ 2] = "MQGem"
@a[ 4] = "script"
@a[ 7] = "example"
@a[ 8] = "Another"
@a[ 9] = "works!"
@a[10] = "MQQSCX"

print "Before"
@i   = minsub(@a)
@max = maxsub(@a)
while(@i <= @max)
  if (exists(@a[@i]))
    print :12:@i,@a[@i]
  endif
  @i = @i + 1
endwhile

sort(@a,1)

print _nl,"After"
@i   = minsub(@a)
@max = maxsub(@a)
while(@i <= @max)
  print :12:@i,@a[@i]
  @i = @i + 1
endwhile
```

This simple script is just demonstrating some language concepts it does actually issue any commands or process any responses from the Queue Manager. The program initialises various elements of array @a to various strings. Before we initialise the array elements we issue the delvar() function to delete all the previous elements of the array. Apart from the fact that initialising strings are in a random order the other thing to notice is that the elements are not contiguous. Elements 3,5 and 6 are not set to any value. This allows us to demonstrate that MQSCX arrays are what are know as sparse arrays. In other words, just because element 100 is defined does not imply that all elements from 1 to 99 are also defined. Of course normally you probably would use contiguous elements but this allows us to demonstrate what happens when you don't. To take account of the fact that some of the array elements may not exists the program calls the function exists() to check whether the element exist before it tries to print it out.

So, run the program with a command like:

```
=import file(d:\mqscx\sort.mqx)
```

We see that after the sort() function is called the 'After' display is quiet different to the 'Before' display. Perhaps not surprisingly we find the elements are now sorted alphabetically. What might be a surprise is that the values don't

start at element 1. Instead they start at 4. This is because the 3 undefined element have been 'sorted' to the start of the array, Essentially an undefined array element is considered the lowest value, even lower than an empty string. Of course, now that the array is sorted all the undefined elements are together. This means that when we call minsub() and maxsub() to find the extent of the array the undefined elements are no longer in the return range, this can be a useful mechanism to filter out undefined values.

Another thing to notice about the sort() function is the collating sequence. It is not what compare would naturally do if you were comparing either ASCII or EBCDIC strings. For comparison the collating sequences are:

Type	Sequence
ASCII	ABCDEFGHIJKLMNOPQRTSUVWXYZabcdefghijklmnopqrstuvwxyz
EBCDIC	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRTSUVWXYZ
MQSCX sort()	AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz

The MQSC sort() collating sequence generally produces lists which are easier to read, often the human brain can remember the words involved but not the case that is used. Using this collation means that similar words are grouped together regardless of case.

The other way sorting can be useful is to help us filter out duplicates and we'll discuss that in the next example.

7.13.3 Printing out namelist values to a file (fnamelists.mqx)

Suppose our requirement is to write a command file which writes out all the names uses in the namelist objects. However, we want a list that doesn't contain duplicates. Can we do that fairly simply? Yes, of course, consider the following program.

```
* Simple script to write all the namelist names to a file

if( _os = "WINDOWS" )
   @filename = "c:\temp\namelist.txt"
else
   @filename = "/tmp/namelist.txt"
endif

@hf = fopen(@filename,"w")
if (@hf = -1)
  print "Can not open file '<@filename>' errno:<_errno> <_errnostr>"
  leave
endif

* Populate an array with the names
@i = 0
delvar(@names)
foreach(DISPLAY NAMELIST(*) NAMES)
  foritem(NAMES)
    @i = @i + 1
    @names[@i] = _item
  endfor
endfor

* Now sort the names
sortd(@names)

* Now, print out the array only if different from the previous
@count = 0
if (@i)
  * Print the first name
  fprint @hf,@names[1]

  * Now, print the rest if different from previous
  @max = @i
  @i   = 1
  while(@i < @max)
    @i = @i + 1
    if (@names[@i] != @names[@i-1])
      fprint @hf,@names[@i]
      @count = @count + 1
    endif
  endwhile
endif

print @count,"names written to",@filename

fclose(@hf)
```

The first thing the program does is open a file to write to. We use the _os system variable to determine the type of OS we are running on to decide on the file name. If the open of the file fails then we write an error message and leave the program.

If the open fails then the fopen() function will return us -1 so we check for this. The next line makes use of useful concept called Substitution commands. Clearly we could construct a string for the error message using an expression, or indeed we could print out multiple values. However, MQSCX supports the ability to replace a a user or system variable name with the value of that variable. You can do this in any statement and can be very useful when issuing MQ commands. However, in this case use it to print out the reason for the file open failure.
The next part of the command script just puts all the names into an array:

```
* Populate an array with the names
@i = 0
delvar(@names)
foreach(DISPLAY NAMELIST(*) NAMES)
  foritem(NAMES)
    @i = @i + 1
    @names[@i] = _item
  endfor
endfor
```

We delete our array first to ensure that we don't have any values left over from the previous run. The other thing to notice here is that we use a foritem() loop to make it simple to process the list of names.

Once we have completed the array we can sort it using the sort() function.

Now all we need to do is print out the values in a loop. The line:

```
if (@names[@i] != @names[@i-1])
```

ensures that we don't print out duplicates, we only print out the value if it is different from the previous one.

Notice that we use the fprint statement rather than just the simple print statement. The fprint statement takes the file identifier, that was returned from the fopen() function, as the first parameter but otherwise it is the same as the print statement.

And finally, once the loop has completed we print out a brief statement about what we've done and then we close the file by calling fclose().

Just a few simple statements and we can extract and collate the data we need and much of it is thanks to the sort() function.

The sort() function has one more trick that we'll discuss in the next section.

7.13.4 The fullest queues (top.mqx)

Quite often we want to display or report a sorted columnar list but we don't necessarily want to sort on the first column. In this example we'll take as our example that we want to print out the queues with the greatest depth. Further we want the queues printed out in descending order.

Let's look at the code:

```
* Get the local queues and their depth and store in array

* First delete any previous results
delvar(@val)

* Now, get a list of queues with messages
foreach(DIS QL(*) CURDEPTH WHERE(CURDEPTH GT 0))
  @val[1,_idxEach] = QUEUE
  @val[2,_idxEach] = CURDEPTH
endfor

* Sort the array in descending order by second column
sortd(@val,2)

* Print out a screen full unless there are even less returned
if (_height)           @limit = _height-3; else @limit = 20; endif
if (@limit > _numEach) @limit = _numEach; endif

* Ok, print out the top few
print :48:"Queue Name","Depth"
print :48:"==========","====="
while(_idxWhile < @limit)
  print :48:@val[1,_idxWhile],:5:@val[2,_idxWhile]
endwhile
```

As in the previous example we issue our now familiar foreach() statement to gather the data. In this case we add a simple WHERE clause since we are only interested in queue with a non-zero current depth.

However, this time we store two values for each queue definition returned. We store both the queue name and the current depth value. Notice that we specify the column number as the first index and the object index as the second index into the array. The way to imagine this is as columns of data on a grid. Each element has a grid position given by it's X and Y coordinate. In this case we only have two columns, X position 1 and 2. However, we could just as easily have many columns.

Once we have collected all the data we drop down to call the sortd() function. The sortd() is identical to the sort() function except that it sorts in a descending rather than ascending order. And here you see the sort() functions new trick. Until now we have sorted a simple one dimensional array. However, here we are passing a two-dimensional array and giving the column number we want to sort on. Of course not only that but, as we know, the second column is a numeric field. This shows that, perhaps not surprisingly, that sort() knows how sort numbers as well as strings.

Now all we need do is print out the top set of queues. We could have just had a fixed number of queues to print out but we use the _height system variable to tell us how big the screen is. If the value of _height is 0 it means we are printing to a file and so we just choose a hard-coded value 20 queues.

7.13.5 Reading files (fgets.mqx)

We have seen how to write a report to a file and so, if only for symmetries sake, here is a simple example of reading a file. There could be many reasons you might want to read a file, for example to read configuration data or even perhaps the MQ error logs. In this example though we are just reading a simple test text file and displaying its contents to the screen.

```
* A simple example showing how to read a file
@filename = "c:\temp\test.txt"
@hf       = fopen(@filename,"r")

* Check that the open worked
if (@hf < 0)
  print "Can not open file '<@filename>' errno(<_errno>) <_errnostr>"
  leave
endif

* Now, read each line of the file
while(fgets(@hf,@line) >= 0)
  print @line
endwhile

fclose(@hf)
```

The program uses the fopen() function with the “r” mode which opens the file in read mode and will fail if the file does not exists. The we get returned a file identifier of -1 we report the failure and leave. If we get a valid file identifier we drop through to a while loop which repeatedly calls the fgets() function. The fgets() function will return the length of line it reads or -1 to signify end of file or -2 to signify an error. If the fgets() function does read a line of data then the data is put in the passed variable. In this case, @line. We then need only print out @line.

7.13.6 Transmission Queues (xmitqs.mqx)

This simple sample shows how easy it is to issue one MQ command based on the response from another. The idea here is to tell you the status of your transmission queues. The command file will query all your transmission queues and if there are messages on the queue report the status of any channels processing the queue. Of course in an ideal world any transmission queues with messages would have running channels so this script could be useful to identify channels with either triggering or connection problems.

```
* Get all the transmission queues
foreach(DISPLAY QUEUE(*) CURDEPTH WHERE(USAGE EQ XMITQ))
  * Is the depth non-zero ?
  if (curdepth > 0)
    @qname = QUEUE
    @depth = CURDEPTH
    * Read channel status for this queue
    foreach(DISPLAY CHSTATUS(*) WHERE(XMITQ EQ <@qname>))
      print @qname,"depth",@depth, CHANNEL,"is",STATUS
    endfor
    if (_matches = 0)
      print @qname,"depth",@depth,"No channel status"
    endif
  endif
endfor
```

What this script demonstrates is how easy it is to issue MQ commands in response to previous MQ commands. The fact that you can nest foreach() statements, as in this case, makes getting additional data or checking status conditions very easy.

7.13.7 Augmenting the command line using a function (conns.mqx)

MQ provides a number of MQSC commands to display what is connected to the Queue Manager. However, it can be difficult to get a quick sense of the number and types of connections. For example, how can one quickly tell how many client connections are currently being made in to your Queue Manager ? It would be nice if we could have a simple command, say “conns”, which shows the connection summary. This is what **conns.mqx** does for us.

```
************************************************************************
* Function : conns                                                     *
* Purpose  : Print out the current state of connections to the QM      *
************************************************************************
func conns()

  =echo resp(no)

  if (_connqmgr = '')
    print 'Please connect to your queue manager before issuing "conns"'
    return
  endif

  DISPLAY QMSTATUS CONNS
  print 'Total connections:',CONNS
  @totalconns = CONNS

  DISPLAY CONN(*) WHERE(CHANNEL EQ ' ')
  @chlconns = @totalconns - _matches

  print '   Local      :',_matches

  @total = 0
  @svrcn = 0
  foreach(DISPLAY CHSTATUS(*) CURSHCNV CHLTYPE WHERE(STATUS EQ RUNNING))
    if (CHLTYPE = "SVRCONN")
      @svrcn = @svrcn + 1
      @total = @total + CURSHCNV
    endif
  endfor
  print '   QMgr Chls  :',@chlconns-@total
  print '   Client Chls:',@total
  print
  print 'Total Running Channel instances:',_matches
  print '   QMgr   Channels:',_matches - @svrcn
  print '   Client Channels:',@svrcn

endfunc
```

The first thing that is quite noticeable is that the file is just a single function definition. We can do this because any command issued on the command line will be compared to the list of defined functions. If a match is found then the function is run. Clearly the function needs to be loaded before it is run and of course you can do this using an =import statement. However, the simplest way of ensuring the conns command is available is to put the function definition into the bootstrap.mqx file described in section 7.12 Bootstrap file on page 59. Of course many of the examples given in this manual could easy be converted into functions in a similar manner.

By now it should be fairly obvious how this function works so we'll just describe the basics. The first things we need to know is how many connections there are in total. MQ gives this to us as a response to DIS QMSTATUS CONNS. However, it gives no idea of the type of the type of connection and we want to know whether the connections are from channels or from local applications. So the next step is to issue a DISPLAY CONN with a

WHERE clause to determine how many of them are local connections. Clearly it follows that the number of channel connections must be the total number of connections minus the number of local connections.
We could have written this command a different way, something like this:

```
foreach(DISPLAY CONN(*) CHANNEL)
{
  if (CHANNEL != '') @chlconns = @chlconns + 1
                else @lclconns = @lclconns + 1; endif
}
```

This method has the advantage that it can easily be extended to include additional selection criteria. For example, suppose we wish to count the number of connections coming from each different channel name or perhaps each different application name.

The next part of the function is concerned with calculating the number of running channel instances. We use a simple foreach(....) clause to process the responses from a DISPLAY CHSTATUS command. In this case we are in counting the number of client and non-client channels. In addition if the channel is a client then we add up how many actual conversations the channel has.

We then print out the values we have and we are done. As you can see it is very easy to augment the command line with additional commands which present the MQ data in a more consumable fashion.

7.13.8 RESET QSTATS

RESET QSTATS, for some reason, can only be issued as a PCF command on Distributed Products. However, it is a useful command so MQSCX will do the translation for you and allow you to issue the command in MQSC form. For example, we can run the following script.

```
while(1)
  foreach(RESET QSTATS(*))
    if (_idxEach = 1)
      =cls
      print date(_time,"HH:M:S"),"Active Queues",
    endif
    if (msgsin | msgsout)
      print :48:qstats,"IN(",:5n:msgsin,:n:") OUT(",:5n:msgsout,:n:") INT(",:n:resetint,:n:")"
    endif
  endfor
  @k = getkey(10)
  if (@k = 27) break; endif
endwhile
print "Active Display End"
```

This simple script will repeatedly issue the RESET QSTATS command to your queue manager and display those queues which are currently active. i.e. Have had messages either put or got from them.

The script will issue a new RESET QTATS command every 10 seconds until the user presses the <ESC> key. This is achieved by calling the getkey() function which will return the key code of the next key pressed.

7.13.9 Monitoring (monitor.mqx)

Clearly MQSCX is not designed as a monitoring application and no one would suggest you would use MQSCX to monitor your whole Queue Manager, unless you were prepared to write a lot code! However there are times, for example when developing MQ applications or doing an MQ demonstration, when a simple mechanisms for repeatedly issuing a set of commands and checking the results can be very useful. Here we show a very simple example of this:

```
while(1)
  foreach(DISPLAY QUEUE(*) TYPE(QLOCAL) CURDEPTH MAXDEPTH WHERE(CURDEPTH GT 0))
    if (CURDEPTH > (MAXDEPTH * 0.8))
      print "Queue",queue,"is getting full, depth is ",CURDEPTH
    endif
  endfor
  if (_idxWhile > 10) break; endif
  wait(5)
endwhile
print "Done"
```

The purpose of this script it merely to print out a message if any queue reaches 80% of its maximum depth. It uses a standard while() loop and the wait() function to add some fixed delay to the processing. Of course one could monitor anything, channels starting, queues being opened or closed or the transmission rates of channels. And of course you can monitor for multiple conditions, for example for a particular set of channels running, or a set of applications all being connected.

The key thing to ensure with a script like this is that the while() loop does terminate for one reason or another. In this case the loop check the while index system variable and only loops 10 times. However, you could use a returned value from an MQ object. For example, you could check the description fields of a process called MQSCX_CONTROL. This would allow you to end the script manually, at the time of your choosing, by simply changing an MQ process definition.

7.13.10 Emulating z/OS responses (disdqm.mqx)

As we all know there is a very high degree of commonality between the z/OS and Distributed command sets. However, there are differences. One of the ones which is often mentioned is the DISPLAY DQM command on the z/OS platform. This command just isn't supported on the Distributed product. This is perhaps a shame because it provides a nice simple summary of the remote aspects of your Queue Manager. Now, at last, we can fill that gap by emulating the command in MQSCX.

The code contains very little we haven't seen already so it will not require much explaining. However this is our longest example code so it may take a little working through to see how it works.

```
************************************************************************
*                                                                      *
* FUNCTION: DISDQM                                                     *
*           An equivalent function to the z/OS command DISPLAY DQM,    *
*           for a distributed queue manager.                           *
*                                                                      *
************************************************************************
func DISDQM()
  if (_qmgr = '')
    print 'Please connect to a queue manager before running DISDQM'
    return
  endif

  ********************************************************************
  * The MVS platform already has its own flavour of this command      *
```

```
  ********************************************************************
  if (_platform = "MVS")
    DISPLAY DQM
    return
  endif

  =echo resp(NO)

  DISPLAY QMSTATUS CHINIT
  if (CHINIT = "RUNNING")
    print _qmgr,'Channel initiator active'
  endif

  @MaxChannels = 0
  @MaxActiveChannels = 0
  ********************************************************************
  * If the connection is to a local queue manager, read the qm.ini    *
  * for some maximum values. If the connection is a client, or a via  *
  * connection then this is not possible.                             *
  ********************************************************************
  if (!(_client | (_connqmgr != _qmgr)))
    @filename = @MQ_DATA_PATH+"/qmgrs/"+_qmgr+"/qm.ini"
    @hf = fopen(@filename,"r")
    if (@hf)
      while (fgets(@hf,@line) >= 0)
        if (findstri(@line, "MaxChannels") > 0)
          @offset = findstr(@line, "=")
          if (@offset > 0)
            @MaxChannelsStr = substr(@line,@offset+1,strlen(@line)-@offset)
            @MaxChannels = eval(@MaxChannelsStr)
          endif
        endif
        if (findstri(@line, "MaxActiveChannels") > 0)
          @offset = findstr(@line, "=")
          if (@offset > 0)
            @MaxActiveChannelsStr = substr(@line,@offset+1,strlen(@line)-
@offset)
            @MaxActiveChannels = eval(@MaxActiveChannelsStr)
          endif
        endif
      endwhile
      if (@MaxActiveChannels = 0 | @MaxActiveChannels > @MaxChannels)
        @MaxActiveChannels = @MaxChannels
      endif
    endif
  endif

  @Running  = 0
  @Starting = 0
  @Stopped  = 0
  @Paused   = 0
  @Retrying = 0
  ********************************************************************
  * Count up the number of channels in each state                     *
  ********************************************************************
  foreach(DISPLAY CHSTATUS(*))
         if (STATUS = "RUNNING")  @Running  = @Running  + 1
```

```
    else if (STATUS = "STARTING") @Starting = @Starting + 1
    else if (STATUS = "STOPPED")  @Stopped  = @Stopped  + 1
    else if (STATUS = "PAUSED")   @Paused   = @Paused   + 1
    else if (STATUS = "RETRYING") @Retrying = @Retrying + 1
    endif;endif;endif;endif;endif;
  endfor

  ********************************************************************
  * Print out the channel state totals, and maximums if we have them  *
  ********************************************************************
  print _qmgr, _matches, :s:'channels current'
  if (@MaxChannels)
    print :n:', maximum',@MaxChannels
  else
    print
  endif
  print _qmgr, @Running, :s:'channels active'
  if (@MaxActiveChannels)
    print :n:', maximum',:s:@MaxActiveChannels
  endif
  print :n:', including', @Paused, 'paused'
  print _qmgr, @Starting,'channels starting,', @Stopped, 'stopped,',
@Retrying, 'retrying'

  ********************************************************************
  * Print out any running listeners                                  *
  ********************************************************************
  foreach(DISPLAY LSSTATUS(*) ALL WHERE(TRPTYPE EQ TCP))
    print _qmgr, 'TCP/IP listener', LISTENER ,'started, for port', PORT,
'address', IPADDR
  endfor
  if (_numEach = 0)
    print _qmgr, 'TCP/IP listener not started'
  endif
  foreach(DISPLAY LSSTATUS(*) ALL WHERE(TRPTYPE EQ LU62))
    print _qmgr, 'LU6.2 listener', LISTENER ,'started, for tp-name', TPNAME
  endfor
  if (_numEach = 0)
    print _qmgr, 'LU6.2 listener not started'
  endif
endfunc
```

The code implements the command as a function 'DISDQM'. More than likely you would put this function in to your Bootstrap file so that the code was always available.

The first thing we do in the code is see whether we are administering a z/OS Queue Manager. If we are then we can just invoke the DISPLAY DQM command itself. It's only on Distributed platforms that we need to do the emulation.

A fair amount of the code is concerned with getting the values for *MAXCHANNELS* and *MAXACTIVECHANNELS*. Unfortunately these values are not returned by any MQ command so instead the MQSCX script has to parse the qm.ini file directly to see whether the values are set. Sadly we can only do this for local queue managers since over a client connection we have no access to the qm.ini file on that Queue Manager. We use the **_client** system variable to tell us whether we are currently connected over a client.

The DISPLAY DQM command is essentially a conglomeration of the output of various MQ commands so the rest of the script is concerned with issuing these commands and tying them all together.

8 Debugging

As command files become more complicated it can be very useful to have some way of debugging your command files. Perhaps the simplest form of debugging mechanism is trace. MQSCX provides a simple command trace when run in file mode. By starting MQSCX with the -Dt parameter MQSX will write out the statements it is executing to the stderr file stream. So, for example, you could run a debug trace of the top.mqx file using the command[12]

```
mqscx -Dt -i top.mqx 2>c:\temp\mqscx.trc
```

Of course sometimes a trace is not sufficient and you need something a little more interactive. To that end MQSCX provides a simple line debugger.

8.1 Debugger

The MQSCX interactive debugger allows you to debug a command file in file mode. The way you invoke the debugger is to use the -! parameter. This parameter must be combined with the -i parameter to identify the command file you wish to debug. So, for example, we would issue a command such as:

```
mqscx -! -i d:\mqscx\top.mqx
```

Assuming that the command file is found and the Queue Manager is available this will start debugging session. This program is one of the example programs shipped with MQSCX so we'll use it as an example. Issue the command and MQSCX will respond with something like the following:

```
MQSC Extended MQSC Program - Version 9.0.0
Connected to 'NTPGC1'
CMD:      1 > delvar(@val)
DBG>
```

There are two things to note here.

1. The line starting CMD: shows the current command we are about to execute
2. We have a prompt DBG> which is clearly waiting for us to do something

Just to get our bearings let's list a bit more of the program. For that we issue the command 'l' (short for list). So, just enter 'l' at the prompt. Now, we see a bit more of the program

```
CMD:        * Get the local queues and their depth and store in array
CMD:
CMD:        * First delete any previous results
CMD:    1 > delvar(@val)
CMD:
CMD:        * Now, get a list of queues with messages
CMD:    2   @i = 0
CMD:    3   foreach(DIS QL(*) CURDEPTH WHERE(CURDEPTH GT 0))
CMD:    4     @i = @i + 1
CMD:    5     @val[1,@i] = QUEUE
DBG>
```

We can see that the program looks very similar to what we would see if we opened the top.mqx file in an editor. However, there can be differences so you should not assume that the format of the display is exactly the same as the input. We will discuss this soon but for now just see the list as a sequence of statements that MQSCX is going to run. The keen eyed amongst you will have noticed two things:

[12]This is a Windows command, clearly the exact command depends on the platform

1. **One of the lines has a > on it**
   This is the same line as we were shown initially. Showing, perhaps not surprisingly that this is still the current line and the next one that will be run.

2. **Only some of the lines are shown and the file has many more lines**
   We just issued the very simple 'l' command which just lists a few lines around the current line. There are many different lists commands we could have issued. For those of you curious you can see the whole file by entering the command 'l *'.

So, I mentioned that the debug list format and the actual file format may not be the same. If you issued the 'l *' command you may already have seen it. The most noticeable example of this are multiple commands on a single lines. Consider the following line that appears later in the file.

```
if (_height)     @limit = _height-3; else @limit = 20; endif
```

This line actually contains multiple statements. If MQSCX reported the line exactly as it is then it would make some things awkward[13]. For example consider the list output:

```
CMD:    9 > if (_height)     @limit = _height-3; else @limit = 20; endif
```

MQSCX can tell us that it's about to run one of these statements but we wouldn't know which one. Later on we'll see that MQSCX also allows us to set breakpoints and having multiple statements on a single line makes that awkward too. So, instead of listing the commands on a single line MQSCX will split them into multiple lines like this:

```
CMD:    9   if (_height)
CMD:   10   @limit = _height-3
CMD:   11   else
CMD:   12   @limit = 20
CMD:   13   endif
```

This makes things much easier since each separate statement has a separate line. It is always clear which line MQSCX is about to run and each statement can be identified uniquely for, say, setting breakpoints. In general you should not have to worry about this distinction, just bear in mind that the statement numbers you see are not line numbers in the file but rather just global statement number and serve only to allow you to easily identity a statement. Another clue to this behaviour is that comment and blank lines do not even get a line number. These lines are logically part of the following statement. In actual fact it is possible, since MQSCX allows you to dynamically modify and include code, that the line number assigned to a particular statement could change during the running of the file.

However, the key thing you probably want to know is what commands you can issue at the debug command line. By all means try a few in the example we are currently running.

8.1.1 <Enter>

The the simplest 'command' you can enter is nothing, ie. Just press enter at the prompt. The debugger will run the current command, display any output and then pause on the next command. Repeatedly pressing enter will effectively step through the program one command at a time.

[13]Those of you familiar with other source level debuggers will be familiar with this problem. How many times have we had to change the program and recompile just so we can set a breakpoint on a multi-line statement ?

8.1.2 print

At any time you may be curious at the value of a variable or expression. You can issue a print command to have the values displayed. For example:

```
print @i, queue, _time, sqrt(100)*3
```

This example shows us printing a user variable, a response variable, a system variable and an expression. The debug print command will use the current debugger stack frame. For more information on this please refer to the 'sf' command on page 79.

8.1.3 eval

The are times when you just want to see what the current command will evaluate to before you run it. If the current statement is a while() statement, an if () statement or an assignment then entering the command 'eval' will evaluate the current statement and display the result. For example:

```
CMD:   32     @value = 3+5
DBG>eval
Result:'8'
```

There may be times where you You may wish to evaluate another expression altogether. In this case you can just follow the keyword 'eval' with the expression you wish to evaluate.

```
DBG>eval 6*7
Result:'42'
```

The expression can contain user variables, system variables and function calls if you wish. For example:

```
DBG>eval @a + @b
Result:'134'
```

Of course it is entirely possible that you have more than one variable @a in the program. If you are inside a function it is possible that there is both a local variable @a that belongs to the function and another variable @a that is in the main program. In fact, if you have a stack of function calls then there could be a myriad of @a variables. So, how does MQSCX know which @a to use in the evaluation ?

Well, the debugger has a 'current' stack frame. A stack frame is what dictates the current scope of variables. Take a look at 'Variable Scope and Stack Frames' on page 40 for more information. By default the debugger will use the current stack frame for its evaluation. However, you can use the 'sf' command to change it to the stack frame you want.

8.1.4 Assignment

Sometimes it is useful to be able to change values of user variables. You can, therefore, enter an assignment directly at the debug line. You can either change existing variables or define new ones.

```
DBG>print @value
10
DBG>@value = 20
DBG>print @value
20
```

Assignment will use the current debugger stack frame. For more information on this please refer to the 'sf' command on page 79.

8.1.5 list (short-form 'l')

The list command will list the command source. There are a number of different variants of the command allowing the user to display different portions of the command list.

Command	Meaning
list	List a few source lines around the current position
list 10	List 10 lines around the current position
list @3,10	List 10 lines starting at statement 3
list @3,@20	List from statement 3 to statement 20
list *	List all source lines
list +	List the next few statements starting at the last list position
list +,10	List the next 10 lines starting at the last list position
list -	List the previous few statements starting at the last list position
list -,10	List the previous 10 lines starting at the last list position
list func	List out the defined functions
list func test	List out some of the function test
list func test *	List out all of function test
list func test @3,@8	List out function test from statement 3 to statement 8

Note that statements are not the same as lines. This is because blank lines and comments are not given statement numbers. The advantage of this is that the list output is far more readable since only actual statements are given numbers. It also means that when a statement line is output it's preceding comment line(s) are also output.

8.1.6 llist (short-form 'll')

The llist command accepts the same parameters as the list command, the only difference is that the llist command will output the file line numbers in addition to the global line number.

8.1.7 where

The where command will display the current command and it's line numbers and potentially some context about where in the set of command you are. For example, you could have the following response.

```
DBG>where
Line:     8 [ 0]+CMD:     3   foreach(DIS QL(*) CURDEPTH WHERE(CURDEPTH GT 0)
Line:    10 [ 0]+CMD:     5!>   @val[1,@i] = QUEUE
```

Notice that the file line number as well as the statement number is displayed. In this example you can see that we are currently on line 10 at statement 5. The where command also tells us the loops that we are currently in, so in this case we are processing the foreach loop on line 8. If we were inside an imported file the where command will also give us the name of the file and place at which the lines were imported.

The where command also show us the stack frames associated with each statement (eg,**[0]**) and it shows us which stack frame is currently active. This is signified by the plus (+) sign immediately following the stack frame number.

8.1.8 Breakpoints

MQSCX allows the user to set a breakpoint on any execution line. You may set up to 10 breakpoints at any one time. There are three commands which allow you to manage the breakpoints in the program.

8.1.8.1 bl

This command will list the current breakpoints. Lines with breakpoints will also be shown with an exclamation mark '!' next to the command when it is displayed using the list command.

8.1.8.2 bp

Set a break point. The command can be used a number of ways.

- To set a break point in the main line code
```
bp <Statement Number>
```
- To set a break point at start of a function
```
bp <function name>
```
- To set a break point at a statement in a function
```
bp <function name> <Statement Number>
```

If you attempt to set a breakpoint on an invalid line MQSCX will try to set a breakpoint on the next execution line. At any one time there can be up to 10 breakpoints set in the program.

You can, if required, set a breakpoint on a line beyond the current set of loaded lines. This can be useful, for example, if you program is going to load other commands using the =import command.

8.1.8.3 bc

Clear break point. The command can be used a number of ways.

- To clear a break point in the main line code
```
bc <Statement Number>
```
- To clear the break point at start of a function
```
bc <function name>
```
- To clear a break point at a statement in a function
```
bc <function name> <Statement Number>
```

Note that a command line may not have the same statement number for the entire life of the program. As files are imported or command inserts occur the exact position of a line may change. Consequently it is entirely possible that to clear a breakpoint you must enter a different statement number than was used to set the breakpoint.

8.1.9 end

The end command can be used to terminate the command file. The MQSCX program will end without running any further commands.

8.1.10 run

The run command instructs MQSCX to run the commands either until the next breakpoint or until the end.

8.1.11 runout

The runout command is essentially a shorthand way of setting a breakpoint after the end of the current loop, issuing the command run, and then clearing the breakpoint. It is useful if you have confirmed the processing of one iteration of the loop and want to continue debugging on the statements following the loop.

8.1.12 sf

By default the current stack frame is used for evaluations but this command can be used to override it temporarily. Please see 'Variable Scope and Stack Frames' on page 40 for a description of stack frames.
The syntax of the command is:

```
sf <stack frame number>
```

Stack frames are number consecutively from 0 depending on how many levels of stack you are in. For example, if we were debugging the factorial function described in the section on 'Recursion' on page 56 we could issue the where command and be shown something like this.

```
DBG>where
Line:    7 [ 0] CMD:    1   print fact(6)
Line:    3 [ 1] FNC:    5             return @n * fact(@n-1)
Line:    3 [ 2] FNC:    5             return @n * fact(@n-1)
Line:    3 [ 3] FNC:    5             return @n * fact(@n-1)
Line:    3 [ 4] FNC:    5             return @n * fact(@n-1)
Line:    1 [ 5]+FNC:    1 > func fact(n)
```

The where command shows where program execution is right now. It essentially shows the stack. In the example above you can see that we are nested a few levels deep in the calls to function fact(). The where command output shows the stack frame after the line number as something like **[2]**. You can see that as each function call is made the stack frame number increases. The stack frame the debugger is currently using is signified by it being followed by a plus '**+**' sign. So, if we issued a debug command like **print @a** we would see the value of @a in stack frame 5.

```
DBG>print @n
2
```

However, suppose we are interested in the value of @a at the first invocation. That is where the **sf** command comes in. Now we enter the sequence.

```
DBG>sf 1
DBG>print @n
6
```

If we now issued the where command we would see

```
DBG>where
Line:    7 [ 0] CMD:    1   print fact(6)
Line:    3 [ 1]+FNC:    5             return @n * fact(@n-1)
Line:    3 [ 2] FNC:    5             return @n * fact(@n-1)
Line:    3 [ 3] FNC:    5             return @n * fact(@n-1)
Line:    3 [ 4] FNC:    5             return @n * fact(@n-1)
Line:    1 [ 5] FNC:    1 > func fact(n)
```

You can see that the plus (+) sign now indicates that the debugger stack frame is now set the 1. Note that the current command is still the same as it always was and execution of the program is not affected by the **sf** command. The **sf** command only affects commands entered in the debugger such as eval, print and assigning values to variables. As soon as you execute any of the actual program lines the stack frames reverts back to the actual stack frame in use by the program.

8.1.13 Help (short-form ?)

The help command provide a quick memory jogger of the debug commands which are available. There are two forms of the command, either just on it's own or with the command you want help on.

```
help
```

The command issued on its own will just show a simple list of the commands available. If you optionally follow the word 'help' (or indeed ?) with the name a command then a description of that command will be shown. For example:

```
help list
```

8.1.14 Command alteration

The debugger does not currently allow you either insert or delete command in the command stream. However, there is an occasion where you can change the commands run. Consider the following short program:

```
CMD:    1   @mycmds = "print 3*4"
CMD:    2   @mycmds
```

In this case MQSCX will run whatever commands are contained in the user variable @mycmds. It follows therefore that you set a breakpoint on line 2 and then change the values of @mycmds you could essentially insert commands into the command stream.

9 Modifying the Client Channel Definition Table

The MQSCX program may be used to either create a new Client Channel Definition Table (CCDT) table file or edit an existing one. MQSCX actually supports two formats of CCDT, either the standard binary CCDT file or the JSON file.

In order to edit or generate CCDT files the program needs to be in one of the CCDT modes. This can be achieved two ways:

- the program is started with either -n or -j flags
- either command **=ccdt or =ccdtj** is entered when the program is running

In CCDT mode any commands entered are processed directly by the **MQSCX** program itself, they are not sent to an MQ Command Server. As such an MQ connection is not required. You don't even need MQ to be installed on the machine to run in a CCDT mode.

MQSCX supports the normal MQSC commands for the manipulation of channels and authinfo[14] objects with the usual MQSCX extensions. The MQSC commands supported are:

```
DISPLAY, DEFINE, ALTER, DELETE CHANNEL
DISPLAY, DEFINE, ALTER, DELETE AUTHINFO
```

The syntax and capabilities of the commands are essentially the same as those supported by Queue Managers but there are some key differences explained below. As always, whenever a file is edited, it is strongly recommended that a backup is taken of the client channel table before any commands to alter the file are issued.

9.1 Channel Table Location

There is a default name and location for channel table file or it can be given explicitly to the program using the –t parameter. The channel table name is constructed from two paths: [FILE PATH][FILE NAME]. These two parts are constructed as follows:

9.1.1 File Path

The path to the location is determined using the following values, in order of precedence:

1. Any path specified using the –t parameter A path will be considered to be included if the value contains a directory separator character (or ':' for Windows). For example, **mqsc –n –t c:\tables**

2. The value given by the MQCHLLIB environment variable

3. The install path of IBM MQ (Windows Only)

4. **MQSCX** Default location
   a) c:\mqm For Windows
   b)/var/mqm For Unix

9.1.2 File Name

The name of the file used is determined using the following values, in order of precedence:

1. Any file name given using the –t parameter The file name will be assumed to have been specified if the –t parameter does not end with a directory separator. For example, **mqsc –n –t myfile.dat**

[14]AUTHINFO objects are only supported in binary CCDT file mode.

2. The value given by the MQCHLTAB environment variable

3. AMQCLCHL.TAB

In JSON CCDT mode the file will have a file type of **.json**.

9.2 Binary CCDT file specifics

9.2.1 Channel Definition Version

When a channel definition is written to a client channel table file is contains an inherent structure version. The version of structure dictates which versions of IBM MQ can read the definition. It is not possible to define an IBM MQ Version 7 client channel table file and have it read by an IBM MQ 6.0 client. A consequence of this is that client channel table definitions should be defined on a Queue Manager which is sufficiently old to include the versions of all the clients which will use the file.

This can be quite awkward for the administrator if they have to keep an old server around just to update their client channel tables. **MQSCX** solves this problem by allowing the specification of the client channel versions in the file either on a global level using the –V parameter or on each client channel definition using the `VERSION` attribute. By default, without using any of the mechanisms described below, the client channel definitions are written using the latest structure version known. Essentially that will be the same version as the first two version numbers of the **MQSCX** program.

Examples would be

- **mqscx –n –V8.0**
  Write all channel records at the IBM MQ Version 8.0 level.
  Issuing the command **mqscx –n –V?** will show you what versions are supported.

- **mqscx –n –Vs10**
  Write all channel records with version 10 structures. This clearly requires knowledge of which structure version is appropriate for each IBM MQ release. The mapping of which structure version is appropriate for a particular IBM MQ release or a particular platform, is documented here
  http://pic.dhe.ibm.com/infocenter/wmqv7/v7r5/topic/com.ibm.mq.ref.dev.doc/q109070_.htm

- **mqsc –n –V0**
  Write all channel records at the same version as they we read from the file. Any new channel definitions will be written in the latest version.

9.2.1.1 Useful commands

The following commands can be useful for displaying and manipulating the record versions of the CCDT file.

DISPLAY CHANNEL(TEST) VERSION
Display the version of a particular channel. Since this is not a Queue Manager MQSC supported attribute, it is not included when **`DISPLAY CHANNEL(TEST) ALL`** is used, but must be requested specifically, for example

```
DIS CHANNEL(TEST) ALL VERSION.
```

The value shown in this attribute will be the IBM MQ product version number, for example, `VERSION(7.5)`. If the version of the client channel is earlier than the support versions, it will be shown as ‘s’ followed by the MQCD version number, for example, `VERSION(s3)`.

DEFINE CHANNEL(TEST) ……VERSION(7.0)
Write this particular channel at IBM MQ 7.0 level. This means that this channel can be read by a IBM MQ client at version 7.0 or later.

ALTER CHANNEL(*) VERSION(7.5)
Alter all channel definitions in the channel file to be written out in IBM MQ 7.5 level. This command is clearly useful for migrating channel definitions from one release to the other. Channel records can be altered both to later or earlier releases. Note, however, that when migrating to older levels that some of the information in the channel definition may be lost since the earlier release did not support that option. As always it is recommended that a backup of the channel table is taken before any modifications are made.

9.2.1.2 Useful functions

It may be that you want to write a script to inspect or manipulate the CCDT records. In such cases it can be useful to have a function which can be used to compare two version numbers. There is such a function available and it is called **`cmpver(v1,v2)`**. It takes two string parameters which should be version numbers, such as "9.1.2" and will compare the two versions and return 1, 0 or -1 depending on whether the first version is greater, equal to or less than the second version.

9.2.2 Authentication Information Objects

The channel definition file doesn't really contain AUTHINFO object definitions; what it contains is a list of Certificate Revocation List (CRL) locations in a single object definition. However, to manipulate this single object it is easier to think of it as multiple objects. The defined object in IBM MQ which maps to the information contained in the list is the AUTHINFO object. However, this is a mapping which, unfortunately, is not complete. The notable differences are:-

- **Object name**
  With discretely defined objects each one has a unique name. However, in the client channel file none of the CRL location entries has an associated name. The consequence of this is that the name has to be generated. The name used is merely the index in the list. So, for example the first entry in the list is given the name '1', the second '2' and so on. Bear in mind that the name is not fixed for a particular entry. The same entry could have different names depending on where it currently exists in the list.

- **Description**
  Normally each AUTHINFO object is entitled to have its own description. However, since all AUTHINFO objects are part of the single object they can only have a single description which they share. Therefore changing the description of one AUTHINFO object will change it for all. A channel file which has come from the server will not contain a description.

- Alter Time/Date In a similar fashion to the description field above the alter time/date fields are shared by all the AUTHINFO objects.

The other requirement when manipulating a list of objects is having some control over the order of the list. In particular it is necessary to have the notion of "insert this definition at this point in the list". This is done using the 'INSERT' keyword. This gives us the following options:

- **DEFINE AUTHINFO(1) CONNAME(HOST1)**
  This will define a new AUTHINFO object at the start (position 1) of the list but will fail if the list already contains a definition.

- **DEFINE AUTHINFO(1) CONNAME(HOST1) REPLACE**
  This will define a new AUTHINFO object at the start (position 1) of the list and will replace any definition currently in the first position.

- **DEFINE AUTHINFO(1) CONNAME(HOST1) INSERT**
  This will define a new AUTHINFO object at the start (position 1) of the list and will 'push' any current definitions further down the list. Note that this means the name by which these subsequent AUTHINFO objects are referred to, changes.

There are two other variations will allow you to define an entry at the end of the list

- **DEFINE AUTHINFO(999) CONNAME(HOST1)**
  Currently you can only define 10 definitions. Any number larger than this will be taken to mean 'at the end'. This command will therefore put this definition at the end of the list.

- **DEFINE AUTHINFO(myname) CONNAME(HOST1) INSERT**
  Although the name of the AUTHINFO should be a number the program will allow the use of a name. This is to maintain as much compatibility with Queue Manager MQSC as possible. Essentially a list of DEFINE AUTHINFO commands will be accepted. Note that the name, regardless of what it is, will be treated as 'end of the list'. So, a list of DEFINE commands will define the objects in the order of the DEFINE commands themselves. Also note that once the definition is made, the name itself is lost. The command DISPLAY AUTHINFO(myname) would therefore fail.

9.3 JSON CCDT file specifics

A JSON CCDT file performs the same job as the standard IBM MQ binary CCDT file but it differs in operation in a few ways:

- **Human Readable**
  Perhaps most obviously the file is human readable rather than the binary format of the original CCDT file. As such it can be tempting to create and edit the file manually in a text editor. Generally speaking this is discouraged since it is very easy to make a formatting error which will only be discovered at runtime and perhaps never. You should use a JSON editor with the appropriate schema or a program such as **MQSCX** which knows the channel definition format.

- **Multiple definitions with the same name**
  The standard CCDT file is a 'keyed' file where the channel name is the key. This means that it is not possible to define more than one channel of the same name.

  As a consequence of this a command such as `ALTER CHANNEL(FRED)` is unambiguous since there can only be one channel called FRED. This is not the case with a JSON CCDT file where you can have any number of channels with the same name. To cope with this ambiguity additional fields have been provided on the commands to allow the user to be more specific about which definition the command is applied to.

- **Channels can be in any order**
  The standard CCDT file is always ordered alphabetically by channel name. However, this is not the case with a JSON file. By default the channel definitions are ordered in the order they are created. However, it is possible to be more explicit about where a new definition should be and indeed there is a new command, `MOVE CHANNEL`, which allows you to move a definition to a new position in the file.

- **Escaped Characters**
  The JSON standard makes use of quote(“) and backslash characters(\). It follows therefore that if an attribute value contains either of these characters then they must be treated in a special way. The JSON format requires that these characters are 'escaped' - this means that they should be preceded with a backslash character. So, '\' becomes '\\' and '"' becomes '\"'. **MQSCX** will add the escape characters as required when the value is written to a file. However, these will be hidden, by default, when the field is shown on the screen. So, although a channel may be displayed as such:

```
[16:13:05] display channel(demo)
CHANNEL(demo)        CHLTYPE(CLNTCONN)   DESCR(This is a "test" channel)
POSITION(15)         INDEX(1)            TRPTYPE(TCP)
CONNAME(localhost(1414))                 QMNAME(QMTEST)      LOCLADDR( )
MAXMSGL(104857600)   HBINT(30)           SCYEXIT(C:\TOOLS\MYEXITS(SCYEXIT))
SCYDATA( )           SENDEXIT( )         SENDDATA( )         RCVEXIT( )
RCVDATA( )           COMPHDR()           COMPMSG()           SSLCIPH( )
SSLPEER( )           SHARECNV(1)         CLNTWGHT(0)         AFFINITY(PREFERRED)
DEFRECON(NO)         CERTLABL( )         ALTDATE(2021-05-09) ALTTIME(13.35.43)
```

  the values of DESCR and SCYEXIT will actually be written with additional '\' backslash characters. If you would rather also see these characters on the screen then you can define an environment variable as follows, `MQSCX_SHOW_ESCAPE_CHAR=y`, before you run **MQSCX.**

  When loading a new JSON CCDT file **MQSCX** will check that attribute values contain the required escape characters. If any are missing then **MQSCX** will display a warning message for the first one it finds.

Where ever possible the syntax of the MQSC commands supported on the Queue Managers has been reproduced so it should be possible to run the channel command scripts directly against the program. However, for the reasons outlined above some of the commands have been extended or restrictions removed as described below.

10 CCDT MQSC commands

Where ever possible the syntax of the MQSC commands supported on the Queue Managers has been reproduced so it should be possible to run the channel command scripts directly against the program. However, for usability some of features have been extended or restrictions removed.

10.1 Channel Type

Normal MQSC channel commands require that the channel type attribute is explicitly specified. Since this program is only concerned with channels with a CHLTYPE of CLNTCONN, this restriction is removed. A channel can be defined as simply as DEF CHL(TEST) and will be a CLNTCONN channel.

10.2 Generic names

Traditionally the DISPLAY CHANNEL command can only contain a single '*' at the end of the name. Just as in the local case MQSCX extends this to allow wildcards '*' and '?' to be entered anywhere in the name.

So, for example:

- **DIS CHL(*T)** Will display any channels ending with the character 'T'
- **DIS CHL(*QM1*)** Will display any channels with 'QM1' somewhere in the name
- **DIS CHL(????)** Will display any channels with a four character name

10.3 =SORT Clause

You can use the `=SORT` or `=SORTD` clauses to sort the responses by whatever expression you wish. For example:

```
DIS CHL(*) =SORT(CONNAME)
```

10.4 =FIND Clause

You can use the `=FIND` clause to do a quick filter of the return responses. For example:

```
DIS CHL(*) =FIND(myhost)
```

10.5 =MAXRESP Clause

You can use the `=MAXRESP` to limit how many responses you are shown. It is unlikely that a CCDT file will contain huge numbers of records so limiting responses may not seem useful. However, suppose we just wish to see the maximum value of something, like say, `HBINT.` We could issue the following command:

```
DIS CHL(*) =SORTD(HBINT) =MAXRESP(1)
```

10.6 Where Clause

As in the local case you can use `=WHERE` to provide additional filtering. So, for example:

- **`DIS CHL(*) =WHERE(HBINT GT 300)`**
  Will display any channels with a heartbeat of greater than 300 seconds

- **`DIS CHL(*) =WHERE(CONNAME LK *1414*)`**
  Will display any channels which have the string '1414' in the connection name

10.7 Where Clause on Alter

Queue Manager MQSC only supports the WHERE clause on the DISPLAY verb. However, there are times when it is useful to be able to make a 'selective' alter as well. For this reason MQSCX allows an =WHERE clause to be specified on ALTER when running in CCDT mode. For example:

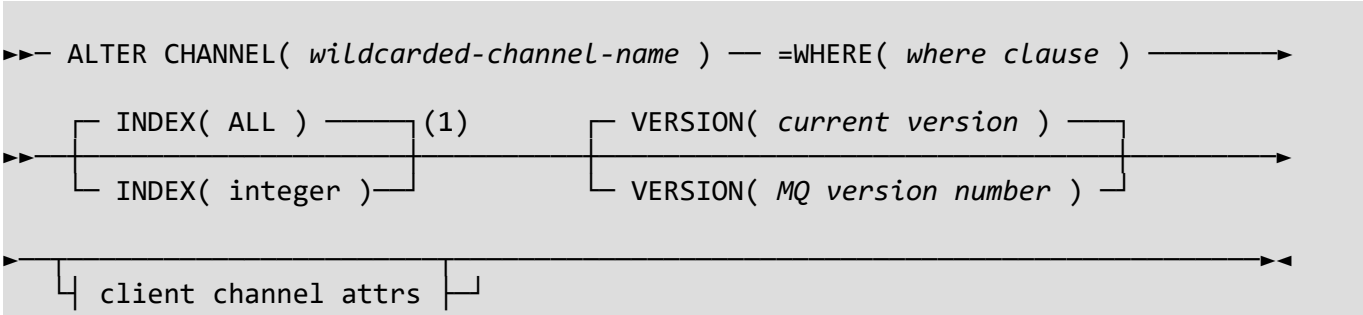
- **ALTER CHANNEL(*) =WHERE(HBINT EQ 300) HBINT(400)**
  Will alter the heartbeat interval, of all the channels that currently have a heartbeat interval value of 300, to 400.

- **ALTER CHANNEL(*) =WHERE(CONNAME LK *host1*) CONNAME(host2)**
  Will alter all channels which are currently configured to connect to ‘host1’ to now connect to ‘host2’.

10.7.1 ALTER CHANNEL

Use the MQSC command **ALTER CHANNEL** to change a channel in your JSON CCDT file. Channel attributes not shown in the railroad diagram follow the IBM MQ **ALTER CHANNEL** command.

10.7.2 Syntax diagram for ALTER CHANNEL



10.7.3 Parameter descriptions for ALTER CHANNEL

(wildcarded channel name)

Traditionally the **ALTER CHANNEL** command cannot contain wildcards. **MQSCX** extends this to allow wildcards '*' and '?' to be entered anywhere in the name.

INDEX

This parameter only applies when using a JSON CCDT file.

The relative index of this channel in relation to other channels with the same name. In order to reference any channel that has more than one definition, you must supply the **QMNAME** or **INDEX** parameter on the command.

Multiple channels with the same name is only possible when using a JSON CCDT file.

Possible Values are:-

(integer)

The numeric value of the index of this channel in relation to other channels of the same name.

ALL

The command applies to all channels of the given name. This is the default value.QMNAME

This parameter only applies when using a JSON CCDT file.

A wildcarded string representing the queue manager name of the channel definition to be changed. This is optional, but it may be enough along side the channel name to uniquely identify the channel definition to be changed. If it is not enough to uniquely identify the channel definition, you must use **INDEX** on your command.

VERSION

The version to use when writing this particular channel definition to the binary CCDT file.

The JSON CCDT does not have the concept of version numbers, so this parameter will be ignored.

The value used in this attribute will be the IBM MQ product version number, for example, **VERSION(7.0)**.

For example, this command indicates MQSCX should write this particular channel at IBM MQ 7.0 level. This means that this channel can be read by a IBM MQ client at version 7.0 or later.

```
DEFINE CHANNEL(TEST) ... VERSION(7.0)
```

=WHERE

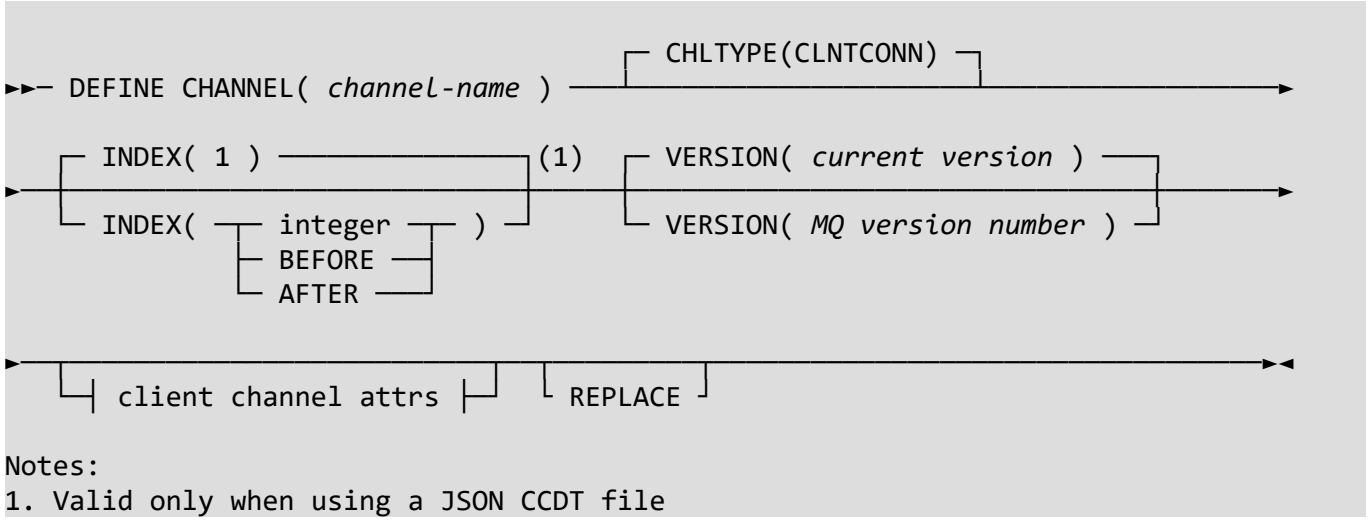
IBM MQ Queue Manager MQSC only supports the **WHERE** clause on the **DISPLAY** verb. However, there are times when it is useful to be able to make a 'selective' alter as well. For this reason **MQSCX** allows an **=WHERE** clause to be specified on **ALTER** when running in both CCDT modes. For example:

- **ALTER CHANNEL(*) =WHERE(HBINT EQ 300) HBINT(400)**
  Will alter the heartbeat interval, of all the channels that currently have a heartbeat interval value of 300, to 400.
- **ALTER CHANNEL(*) =WHERE(CONNAME LK *host1*) CONNAME(host2)**
  Will alter all channels which are currently configured to connect to 'host1' to now connect to 'host2'.

10.8 DEFINE CHANNEL

Use the MQSC command **DEFINE CHANNEL** to create a channel in your CCDT file. Channel attributes not shown in the railroad diagram follow the IBM MQ **DEFINE CHANNEL** command.

10.8.1 Syntax diagram for DEFINE CHANNEL



Notes:
1. Valid only when using a JSON CCDT file

10.8.2 Parameter descriptions for DEFINE CHANNEL

CHLTYPE

Normal MQSC channel commands require that the channel type attribute is explicitly specified. Since this program is only concerned with channels with a **CHLTYPE** of **CLNTCONN**, this restriction is removed. A channel can be defined as simply as **DEFINE CHANNEL(TEST)** and will be a **CLNTCONN** channel.

INDEX

This parameter only applies when using a JSON CCDT file.

The relative index of this channel in relation to other channels with the same name. In order to define more than one channel with the same name, you must supply the **INDEX** parameter on the command.

Multiple channels with the same name is only possible when using a JSON CCDT file.

Possible Values are:-

(integer)

The numeric value of the index of this channel in relation to other channels of the same name. If this parameter is not specified, it defaults to the value of 1.

The specified number must be positive, and either within the range currently in use, or the next index. If writing a script, which may be run multiple times, it is recommended that you use specific integer index values.

BEFORE

Insert this channel definition before all channels with the same name. This has the effect of incrementing the index of all the channels of that name.

AFTER

Add this channel definition after all the channels with the same name. This does not change the index of any other channel of that name.

VERSION

The version to use when writing this particular channel definition to the binary CCDT file.

The JSON CCDT does not have the concept of version numbers, so this parameter will be ignored.

The value used in this attribute will be the IBM MQ product version number, for example, **`VERSION(7.0)`**.

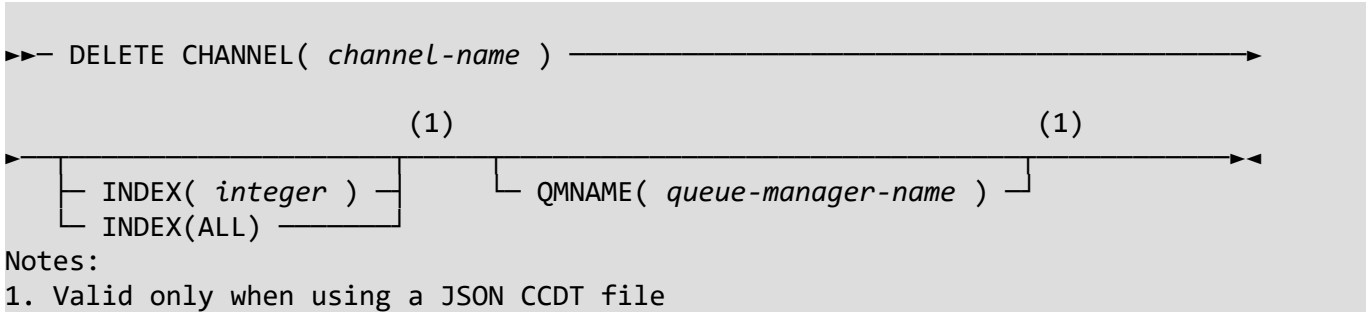
For example, this command indicates **MQSCX** should write this particular channel at IBM MQ 7.0 level. This means that this channel can be read by an IBM MQ client at version 7.0 or later.

```
DEFINE CHANNEL(TEST) ... VERSION(7.0)
```

10.9 DELETE CHANNEL

Use the MQSC command **DELETE CHANNEL** to delete a channel in your CCDT file. Deleting a channel results in the indexes of other channels of the same name being decremented, so do not rely on any previously displayed indexes after using this command.

10.9.1 Syntax diagram for DELETE CHANNEL



10.9.2 Parameter descriptions for DELETE CHANNEL

INDEX

This parameter only applies when using a JSON CCDT file.

The relative index of this channel in relation to other channels with the same name. In order to reference any channel that has more than one definition, you must supply the **QMNAME** or **INDEX** parameter on the command.

Multiple channels with the same name is only possible when using a JSON CCDT file.

Possible Values are:-

(integer)

The numeric value of the index of this channel in relation to other channels of the same name.

ALL

If there are multiple channel definitions with the same name (and optionally the same queue manager name), using **INDEX(ALL)** will remove all the referenced channel definitions.

QMNAME

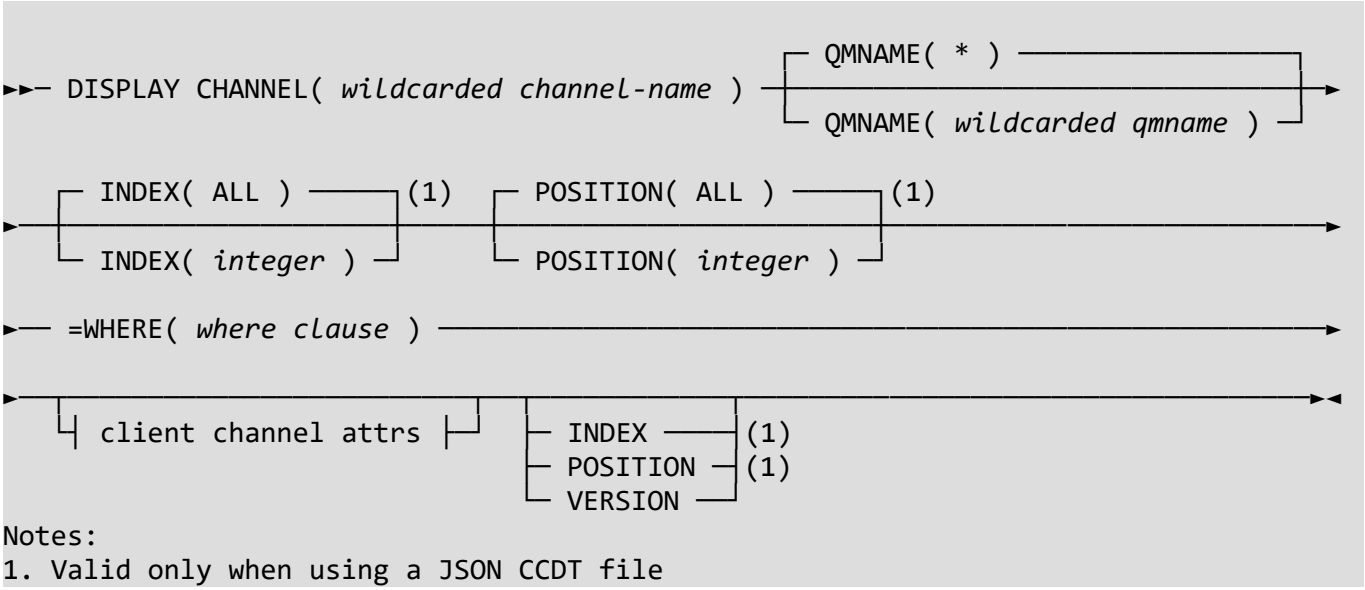
This parameter only applies when using a JSON CCDT file.

The queue manager name of the channel definition to be deleted. This is optional, but it may be enough along side the channel name to uniquely identify the channel definition to be deleted. If it is not enough to uniquely identify the channel definition, you must use **INDEX** on your command.

10.10 DISPLAY CHANNEL

Use the MQSC command **DISPLAY CHANNEL** to view channels in your JSON CCDT file. Channel attributes not shown in the railroad diagram follow the IBM MQ **DISPLAY CHANNEL** command.

10.10.1 Syntax diagram for DISPLAY CHANNEL



10.10.2 Parameter descriptions for DISPLAY CHANNEL

(wildcarded channel name)

Traditionally the **DISPLAY CHANNEL** command can only contain a single '*' at the end of the name. Just as in the local case **MQSCX** extends this to allow wildcards '*' and '?' to be entered anywhere in the name.

So, for example:

- **DIS CHL(*T)** Will display any channels ending with the character ‘T’
- **DIS CHL(*QM1*)** Will display any channels with ‘QM1’ somewhere in the name
- **DIS CHL(????)** Will display any channels with a four character name

INDEX

This parameter only applies when using a JSON CCDT file.

The index of a channel definition to be displayed, for the specified channel or channels.

This parameter can be used to limit the number of channel definitions displayed. If it is not specified, the display is not limited in this way.

POSITION

This parameter only applies when using a JSON CCDT file.

The position of the channel definition to be displayed, for the specified channel.

This parameter can be used to limit the output to a single channel definition. If it is not specified, the display is not limited in this way.

QMNAME

This parameter only applies when using a JSON CCDT file.

The queue manager name for which channel definitions are to be displayed, for the specified channel or channels. This can be a wildcarded string.

This parameter can be used to limit the number of channel definitions displayed. If it is not specified, the display is not limited in this way.

=WHERE

As in the local case you can use **=WHERE** to provide additional filtering. So, for example:

- **DIS CHL(*) =WHERE(HBINT GT 300)**
  Will display any channels with a heartbeat of greater than 300 seconds
- **DIS CHL(*) =WHERE(CONNAME LK *1414*)**
  Will display any channels which have the string '1414' in the connection name

10.10.3 Requested Parameters

INDEX

This parameter only applies when using a JSON CCDT file.

The relative index of this channel in relation to other channels with the same name.

This attribute is included when **ALL** is specified.

POSITION

This parameter only applies when using a JSON CCDT file.

The position of this channel definition in the JSON CCDT relative to all channels.

This attribute is included when **ALL** is specified.

VERSION

Display the version of a particular channel. Since this is not a Queue Manager MQSC supported attribute, it is not included when **DISPLAY CHANNEL(TEST) ALL** is used, but must be requested specifically, for example:

DIS CHANNEL(TEST) ALL VERSION.

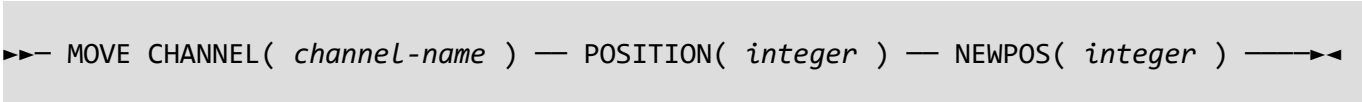
The value shown in this attribute will be the IBM MQ product version number, for example, **VERSION(7.5)**. If the version of the client channel is earlier than the supported versions, it will be shown as 's' followed by the MQCD version number, for example, **VERSION(s3)**.

In **=ccdtj** mode, this will always display the current MQ version.

10.11 MOVE CHANNEL

Use the MQSC command **MOVE CHANNEL** to change the position of a channel in your JSON CCDT file. This command only applies when using a JSON CCDT file. Changing the position of a channel results in in the indexes and positions of other channels being shuffled to cater for the moved definition, so do not rely on any previously displayed indexes or positions after using this command.

10.11.1 Syntax diagram for MOVE CHANNEL



10.11.2 Parameter descriptions for MOVE CHANNEL

(channel name)

The name of the channel record to be moved to a new position in the JSON CCDT file.

POSITION

The current position of the channel record to be moved to a new position in the JSON CCDT file.

NEWPOS

The new position of the channel record after it has been moved.

11 MQEV Event Monitoring

MQEV is a program which will receive, store and process MQ event messages and MQ accounting and statistics messages. It greatly alleviates a number of the disadvantages of trying to process these output messages from your queue manager, and allows you to simply and efficiently search, consolidate, summarise and total the data gathered. For a complete description you should read the MQEV manual which can be downloaded from http://www.mqgem.com/mqev.html

In order to view and administer the MQEV program, you can use either an command line tool such as MQSCX, a GUI tool like MO71 (also from MQGem Software), or if you prefer, you can write your own tool to interface with MQEV. This chapter will detail how to use MQSCX to interact with MQEV.

11.1 Licensing

In order to use MQSCX to interact with the MQEV product, you only require an MQEV licence. If a licence file is bought you will be sent an ***mqgem.lic*** file. All you need to do is place this licence file in the same directory as the MQSCX program.

If you already have an MQSCX licence for administrating and configuring your IBM MQ queue managers, you can combine your existing MQSCX license file and a new MQEV licence file together in the same file as detailed in *17.3 Multiple licences* on page 113.

11.2 MQSCX running modes

MQSCX can run in different modes depending on what you need to use it for.

- **MQSC mode**
  In this mode, the default mode if you have an MQSCX license, commands are sent to the IBM MQ command server queue, SYSTEM.ADMIN.COMMAND.QUEUE. If you are not already in this mode you can switch to it using the command `=mqsc`

- **CCDT mode**
  This mode allows you to edit CCDT files – see more in *Chapter 9 Modifying the Client Channel Definition Table* on page 81.

- **MQEV mode**
  In this mode, the default if you only have an MQEV license, commands are sent to the MQEV command server queue, MQGEM.MQEV.COMMAND.QUEUE. If you are not already in this mode you can switch to it using the command `=mqev`

So that you can easily tell which mode you are in, you can set your prompt to show the mode you are in. See *12.1 Prompt* on page 97. The default prompt will show when you are in MQEV mode.

11.3 Displaying MQEV data using MQSCX

Hopefully, as discussed before, you have installed the latest version of **MQSCX** in your path and given it access to your **MQEV** licence. Assuming that is the case then we can just start an instance of **MQSCX** with the following command:

```
mqscx -m MQG1
```

Now, when **MQSCX** starts, if you also have an MQSCX license, it will be expecting to issue IBM MQ commands. However, we want **MQSCX** to send it's commands to **MQEV**. The way we tell **MQSCX** to do that is to issue the following command:

```
=mqev
```

What this does internally is to tell **MQSCX** to open and direct all subsequent commands to `MQGEM.MQEV.COMMAND.QUEUE`. Of course in order to do this you must be authorised to perform these actions.

This command is not necessary if you only have an **MQEV** licence and not an **MQSCX** licence. In such cases **MQSCX** will start initially in **MQEV** mode. By default the **MQSCX** prompt will tell you whether you are in **MQEV** mode by displaying something such as:

```
MQEV:MQG1>
```

Now you can issue **MQEV** commands. To learn about the complete set of **MQEV** commands, please read the **MQEV** manual. You can use various commands to configure what **MQEV** will process, for example, to ensure **MQEV** is processing events from the SYSTEM.ADMIN.COMMAND.EVENT queue, issue the following command:

```
RESUME EVQ(SYSTEM.ADMIN.COMMAND.EVENT)
```

If you have some events, or accounting and statistics data generated on the queues that **MQEV** is processing, there are several commands that allow you to see the data:

- DISPLAY EVENTS
- DISPLAY ACCTMQI
- DISPLAY ACCTQ
- DISPLAY STATCHL
- DISPLAY STATMQI
- DISPLAY STATQ

These commands are fully described in the MQEV manual, but let's take a look at an example.

```
DISPLAY EVENTS(*)
```

This will return something that looks a little like this (depending entirely on the events you have processed of course.

```
EVQMGR(MQG1)         EVTIME(2019-10-09 22:15:26 (Local))
SUMMARY(Config - Create Object - Queue:MQEV.TEST.QUEUE)

EVQMGR(MQG1)         EVTIME(2019-10-09 22:15:26 (Local))
SUMMARY(Command - Create Queue - Queue:MQEV.TEST.QUEUE)

Total display responses - Received:4
```

The first question that may spring to mind is “where are all the other fields?”. We all know that a queue definition has a lot of fields so why is the output so short ? Well, the answer is that **MQEV** allows you to choose how much data you get back. By default you will only get back a summary of information such as the Queue Manager it happened on, the time it happened and summary text of what happened in a field called SUMMARY. This simple field can be very useful in getting a quick feel for the event rather than trying to assimilate lots of separate values.

However, let's assume that we are curious type and we want to see the whole event, how would we do that ? Well change your command to the following (using **MAXRESP** to ensure only the most recent two events are returned):

```
DISPLAY EVENTS(*) DISTYPE(DETAIL) MAXRESP(2)
```

You will see that now we get some additional fields. We get told the object name, object type and userid that was most associated with the event and the event reason code.

```
EVQMGR(MQG1)         EVENTS($EVENTS)     EVTIME(2019-10-09 22:15:26 (Local))
EVREASON(CFGCRTOBJ) EVUSERID(mqgemusr)  EVOBJNAME(MQEV.TEST.QUEUE)
EVOBJTYPE(QUEUE)    EVENTID(00000004)
SUMMARY(Config - Create Object - Queue:MQEV.TEST.QUEUE)

```

```
EVQMGR(MQG1  )       EVENTS($EVENTS)      EVTIME(2019-10-09 22:15:26 (Local))
EVREASON(CMDCRTQ)    EVUSERID(mqgemusr)   EVOBJNAME(MQEV.TEST.QUEUE)
EVOBJTYPE(QUEUE)     EVENTID(00000003)
SUMMARY(Command - Create Queue - Queue:MQEV.TEST.QUEUE)

Total display responses - Received:2
```

As I am sure you realise we are still not seeing the whole event. If we really needed to see everything then we could issue the following:

```
DISPLAY EVENTS(*) ALL MAXRESP(2)
```

Which will show you a much large output response will all the fields in the events.

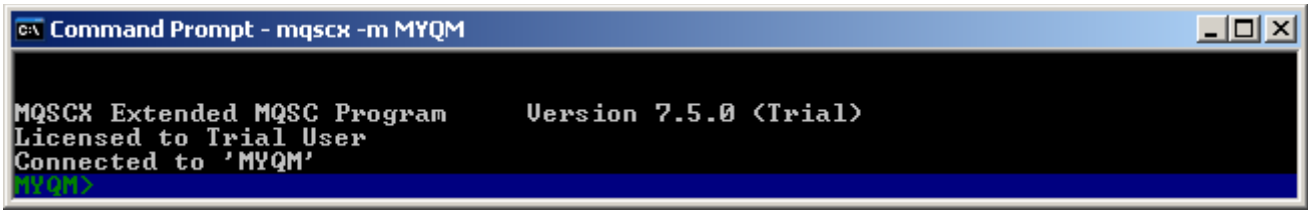
For more details about **MQEV**, and it's commands, check out the “Getting Started” chapter in the **MQEV** manual.

12 Changing appearance

One of the most immediate striking differences between **RUNMQSC** and **MQSCX** output is **MQSCX**'s use of colour. However, there are quite a number of differences in the way the output is portrayed. Most of these can be changed in one way or another.

12.1 Prompt

By default the prompt of the **MQSCX** program shows the Queue Manager the program is currently administering. Normally this would also be the Queue Manager the program is connected to, but not always since you can control one Queue Manager via another. When **MQSCX** is first started, and assuming it managed to connect to the Queue Manager, the prompt will look something like this:



This initial prompt can be changed, if required, via the *=set* command.

If you are in MQEV mode (see *11.2 MQSCX running modes* on page 94) then the default prompt will also show that.

Take a look at the settings panel (either press F4 or enter the command "`=show scrn(sets)`"). You can see that the 'prompt' setting is set to the string `'%eMQEV%e:%m>'`. Essentially whatever characters the prompt value is set to will appear as the prompt, with the exception of characters following a percentage (%) sign. These characters are substitution characters and will be replaced by the appropriate data.

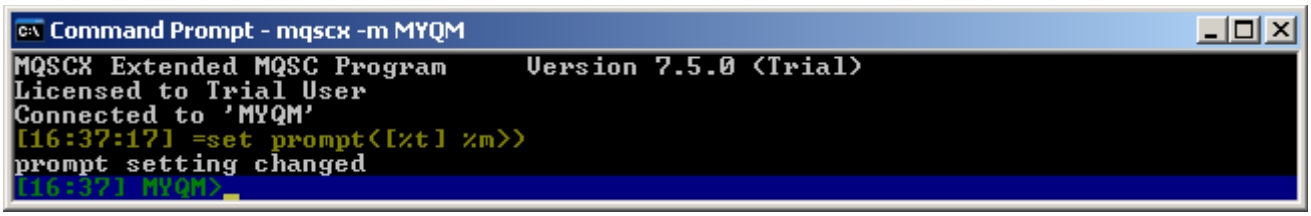
The list of substitution characters are as follows:

Character	Displayed Prompt Value
t	Time in HH:MM format
T	Time in HH:MM:SS format
m	Managed Queue Manager
M	Connected Queue Manager if different from the managed Queue Manager
n	Conditional insertion if program is in CCDT mode (see below)
i	Conditional insertion if program is in MQSC mode (see below)
e	Conditional insertion if program is in MQEV mode (see below)
%	%
a	Conditional insertion if connection is using MQMONA (see below)
c	Conditional insertion if connection is over a client (see below)
l	Conditional insertion if administration is local (see below)
r	Conditional insertion if administration is remote (see below)
C	Conditional insertion if connected to a Queue Manager
D	Conditional insertion if disconnected from the Queue Manager

So, for example, suppose wish to always see the current time in the prompt we could issue the following command:

```
=set prompt([%t] %m>)
```

the effect on the prompt is shown below:

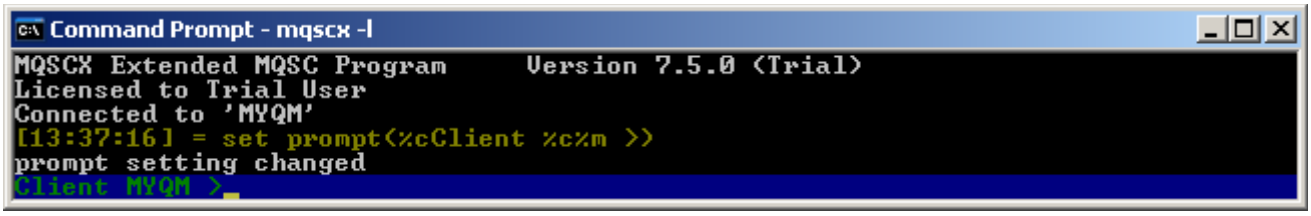


12.1.1 Conditional Insertion

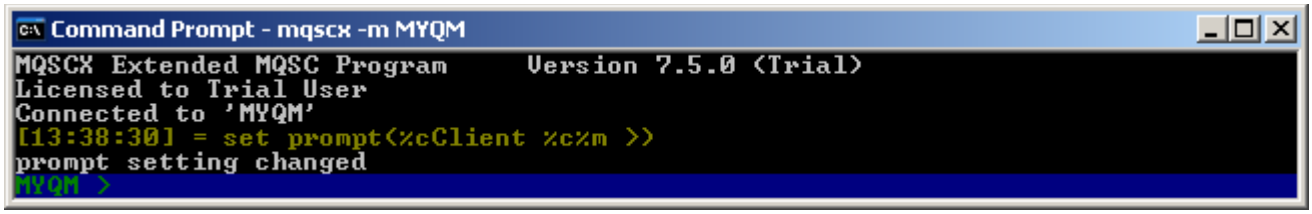
Conditional insertion is a way of adding text to a prompt but only if a 'condition' is true. The general idea is that all text between the tags is included but only if the condition is true. So, for example:

```
=set prompt(%cClient %c%m >)
```

In this case the text between the two %c tags is only displayed in the prompt if the connection is made over an MQ client. This setting would show the following prompt if MQSCX was connect over client connection.



However it would show the more normal prompt if it was a local connection.

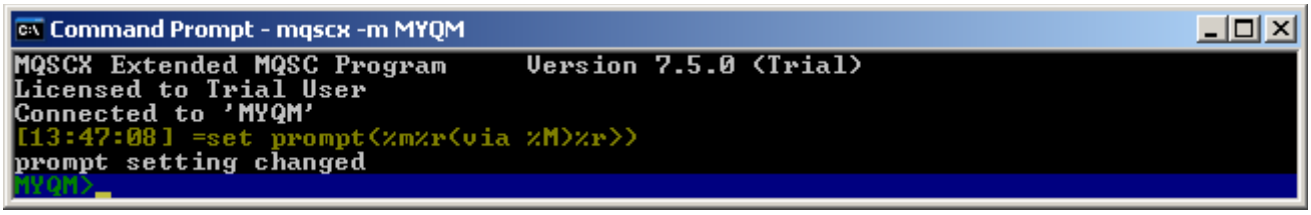


In a similar way we could indicate that the administration of the Queue Manager is being made via an intermediate Queue Manager with a command such as this:

```
=set prompt(%m%r(via %M)%r>)
```

This would show this type of prompt if MQSCX was administering Queue Manager MYQM via an intermediary Queue Manager NTPGC1.

However, again, it would show the normal prompt if not going via an intermediate Queue Manager.



12.2 Colours

One of the most noticeable differences between the **RUNMQSC** program and **MQSCX** is the use of colour to distinguish different output. Colours can be very useful to help the eye make sense of the display screen, without colour a screen full of data can very quickly become tiring to look at. Of course not all of us see colours the same way and each of us has their own preferences so with this in mind the **MQSCX** program allows the user to modify the colours shown.

12.2.1 Setting the various screen elements

The various screen elements can be set to a range of colours independently. Each element has a foreground and background colour setting. There is also the concept of a 'default' value. The default value allows you to, for example, set a number of elements to have the same background colour without having to set them individually.

Suppose we wanted to set the prompt background colour to red. The command would be:

```
=colour bgprompt(red)
```

if we now wanted it's foreground colour to be white we would enter:

```
=colour fgprompt(white)
```

As you can see the colours are set using the 'colour' command and a prefix of 'bg' and 'fg' before the element name.

We could, if we wished, make both changes in a single command:

```
=colour fgprompt(white) bgprompt(red)
```

A description of the elements and the colours you can use is listed in the =colour command description please see section *18.8=colour* on page 116.

12.2.2 Switching colours off

Some users may find colours distracting and actually prefer all text to be output in the same colours. This can be achieved in a single command:

```
=colour colours(off)
```

The screen will immediately revert to displaying all the data using the 'default' foreground and background for each element. Of course when you get tired of the monochrome display you can always go back to the pretty display using:

```
=colour colours(on)
```

12.3 Columns

MQSCX adopts a more flexible approach to columns than the rather rigid two column format used by **RUNMQSC.**

MQSCX supports two types of columnar display, content and fixed columns. The way you choose which types of columns you want is by setting colstyle using a command such as `=set colstyle(content)`. By default **`colstyle`** is set to auto which means the column will be **`content`** if the response fits on a single line or **`fixed`** if it spans multiple lines.

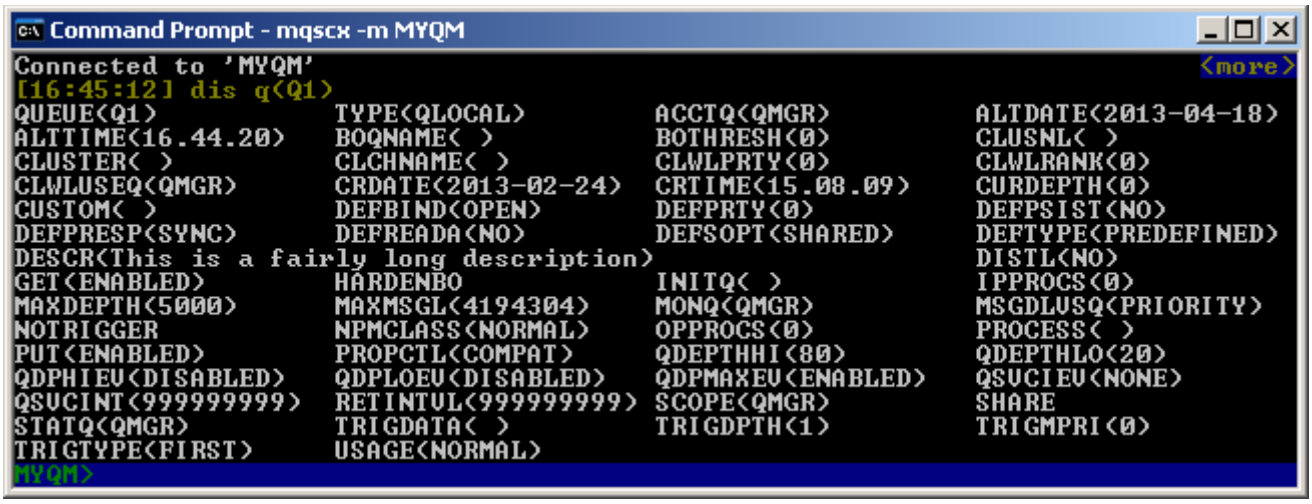
12.3.1 Content Columns

This is traditionally what we think of when we think of columns of data. That is each attribute is given its own, fixed position, column. This is the more ideal form of columns. However, the columnar nature of the output is diminished if the data spans more than a single line. If the data spans multiple lines then the columns are still shown but it is not as obvious that each attribute is displayed at the same screen offset.

12.3.2 Columns based on fixed width portions of the screen

In this mode **MQSCX** will fill the screen columns with the returned data as required. **MQSCX** will display as many columns of data as will reasonably fit across the width of a given screen. As a consequence **MQSCX** makes much better use of the available screen real estate. However, a particular attribute may not be in the same column for each response.

So, for example, if we display a queue definition we see output like this:



In general you will notice that the output is ordered into four columns. However, if an attribute it too large to fit in a column the output is adjusted accordingly. In this example the description field (DESCR) is longer than the column so the remaining attributes are adjusted accordingly.

By default **MQSCX** will adjust the number of columns according to the screen width. However, if you prefer you can explicitly set the number of columns. Suppose we wanted three columns, we would enter the following command:

```
=set columns(3)
```

The setting will apply to all previous output so you will immediately see the effect on the screen. To go back to automatic setting of columns issue the command:

```
=set columns(auto)
```

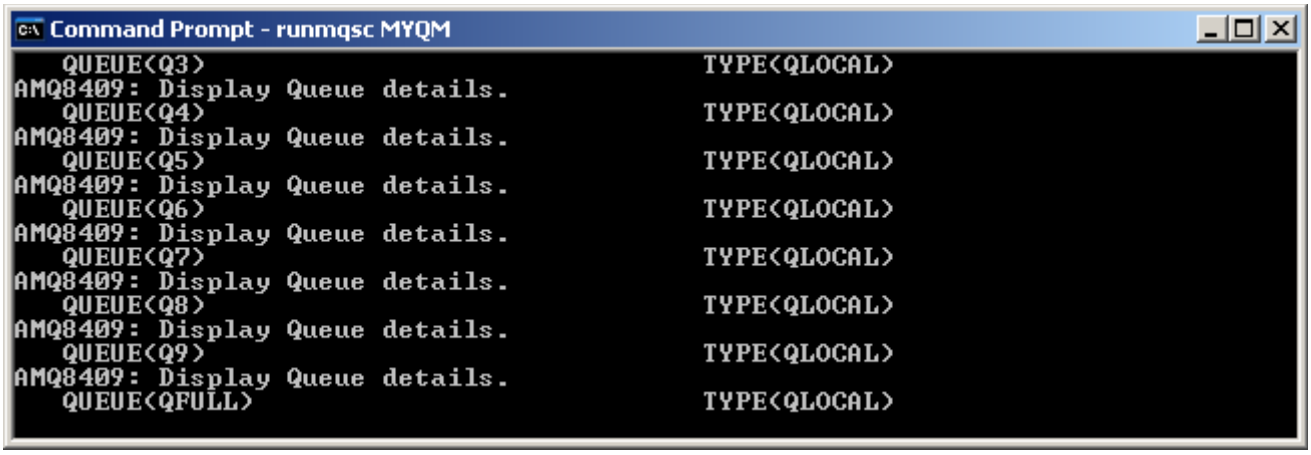
When in 'auto' mode the number of columns is set by the 'ideal' width of each column. This can also be adjusted. By default it has the value of 20 characters, however, experiment with other values:

```
=set colwidth(30)
```

You will notice that this command will probably reduce the number of columns displayed. However, this can produce a neater display if you have many long attributes. Setting a really small column width is likely to produce a rather messy looking display unless all the attributes are of similar length.

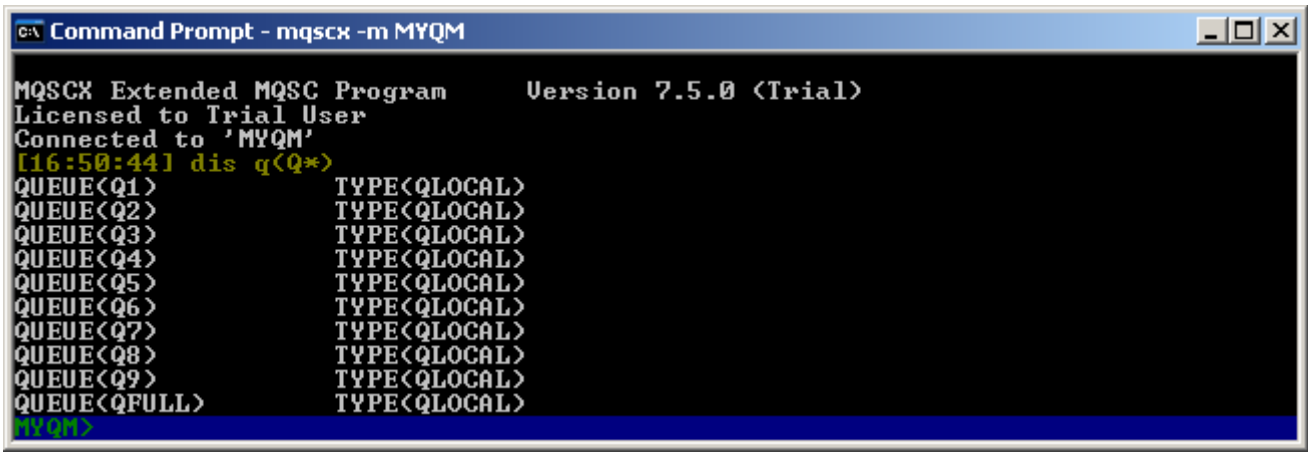
12.4 Separators

The standard RUNMQSC 'DISPLAY' output displays a 'detail' message between each object. Like this:



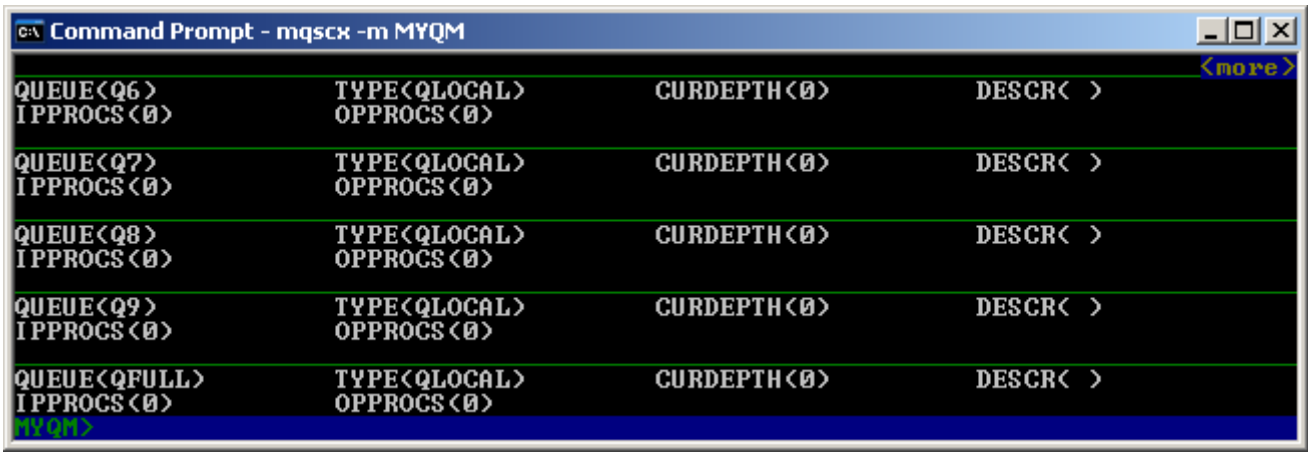
These detail messages can be distracting and clearly they take up a significant amount of the output real estate. MQSCX allows the user to replace these messages or remove them all together.

For example, the same command issued by **MQSCX** will show:



You can clearly see that more objects are shown. Of course there are times when having some form of separator is useful. For example, suppose we decided we wanted a little more information about the queues and changed the command to 'dis q(Q*) curdepth descr ipprocs opprocs'.

The output we see from MQSCX is now:



You can see that we now have some form of separator between the object definitions. Why ? Well, since one or more of the object definitions spans more than one line the screen could look confusing without some form of separator. MQSCX will, by default, automatically insert a separator if it thinks it helps. The setting which controls this behaviour is 'autosep'. By default 'autosep' is on but you can switch it off with the command:

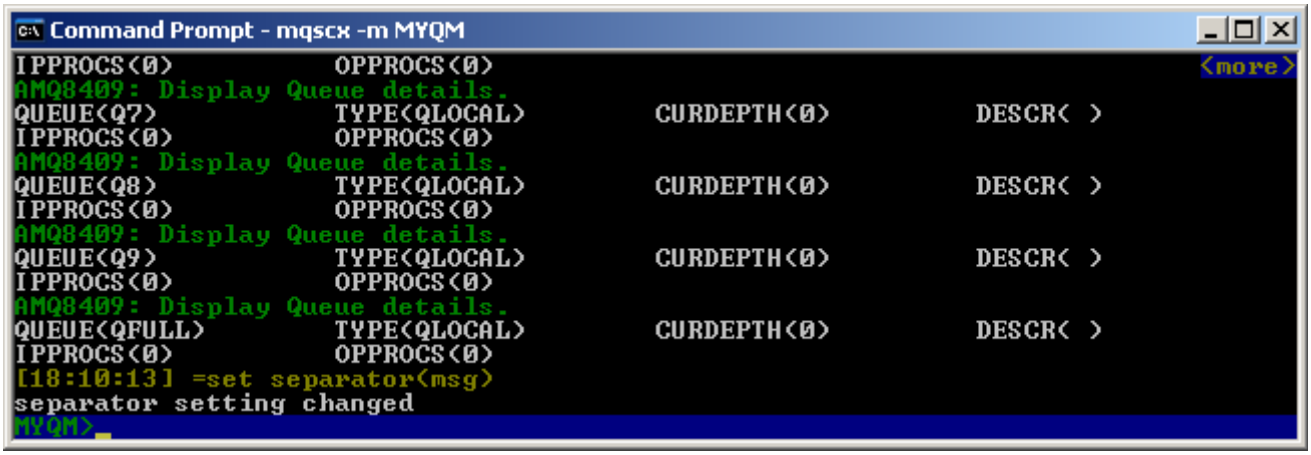
```
=set autosep(off)
```

Issue the command and scroll up (if necessary) to look at the previous 'dis q(*)' command. You will see now that we have separators between each of the objects. Since you have switched auto-separators off they are always displayed.

So, the next question you may be asking is “why do we see a solid line rather than the familiar MQ message ?”. Well again the answer is simply clarity. The MQ message can be distracting to the eye and a single line is less intrusive. However, different people will prefer different things so, not surprisingly, you can change which separator gets displayed. For example, suppose you issue the following command:

```
=set separator(msg)
```

You will see that the display we had previously will immediately change to:



You can see that we have our MQ message back.

The separator setting can be set to a number of values, these are:

Value	Meaning
msg	The original MQSC detail message is shown
none	No separator is shown
blank	A blank line
dotted	A dotted line
solid	A solid line (subject to command font)

You will notice that changing the separator setting will immediately affect the existing display, it is not necessary to re-issue the DISPLAY command after changing the setting.

The last thing you may notice about the separator line is that it is displayed in a different colour from the MQ objects. Again this is to make the display less intrusive. You can change the colour used using a command like the following:

```
=colour fgmsgs(white) bgmsgs(cyan)
```

Feel free to adjust the colours according or go back to the default with the command:

```
=colour fgmsgs(green) bgmsgs(default)
```

For more information about the colour command and the other elements of the screen you can change please see section *18.8=colour* on page 116.

12.5 Window re-size support

MQSCX will automatically detect the size of the command window and set the display accordingly.

Unix and z/OS UNIX
On Unix and z/OS UNIX the display will be adjusted accordingly as the command window is re-sized.

Windows
Unfortunately on Windows MQSCX does not get notifications of screen size changes so if the window is resized for any reason it is necessary to manually cause MQSCX to resize it's screen[15]. The way this is done is to issue the 'Refresh' key action which by default is set to the key combination <Ctrl-r>.

[15] I would be interested to hear from any Developers on the correct way to detect command window resize events.

13 Configuration file

When **MQSCX** runs it will look for a file containing the user preference settings This file is called the configuration file. The file name is always named ***mqscx.cfg*** when stored in a hierarchical file system, and by default it should be in the same directory as the **MQSCX** program, except on AIX and z/OS UNIX where this location cannot be determined.

If you wish to change the directory where the configuration file is read and written to then set the `MQSCXCFG` environment variable to the appropriate value. For example:

On Windows

```
set MQSCXCFG=c:\tools
```

On Unix or z/OS UNIX

```
export MQSCXCFG=/usr/tools
```

Specifics when running on z/OS

If running on z/OS, **MQSCX** will attempt to open DD card `MQSCXCFG`, then check for environment variable **`MQSCXCFG`**, and finally check the current working directory, since the location of the program cannot be determined.

Whether specified using the DD card or environment variable, this can reference a PDSE (in which case the member name ***MQSCXCFG*** will be used), or a UNIX file system directory (in which case the file name ***mqscx.cfg*** will be used).

The configuration file format is a simple keyword/value pair. If necessary the file can be edited by hand but this is not recommended.

The configuration file will be updated when the program ends, providing you are running in interactive mode. That is:

- Not using file mode (**`-f`**)
- File mode with **`stdin`** as your input, e.g. interactive in non-terminal environments

This allows you to be sure that your configuration is not affected by anything the script might choose to do. File Mode is described later.

If you require your configuration to be saved after running a script which makes updates to the configuration, use the **`-S`** flag to force the configuration to be saved when you don't meet the above criteria.

If **MQSCX** can not write to the configuration file for some reason then an error message will be displayed. **MQSCX** does try to ensure that a failure during the file write still maintains a complete and valid configuration file. However, it is recommended that you keep a backup of your configuration file just in case you lose the configuration.

14 File Mode

Up until now we have been talking about using **MQSCX** in interactive mode where the output is being written to a console window and input has been taken from a user typing at a keyboard. Of course there are times when you wish to process a set of commands from a command file and have the output also written to a file. This is referred to as file mode.

There are essentially four ways **MQSCX** can be run in file mode:

14.1 Piping a file in from stdin

```
c:\> mqscx < mycommand.mqs
```

Since you have piped a file in to the program **MQSCX** will switch to file mode since it knows that key user input is not possible.

14.2 Explicitly passing in an input file

```
c:\> mqscx -i mycommand.mqs
```

Here we explicitly pass a file to process and it has the same effect as piping in a file. Since input is from a file the program does not expect key input.

14.3 Explicitly asking for file mode

```
c:\> mqscx -f
```

The **`-f`** parameter requests that **MQSCX** runs in file mode but clearly doesn't supply an input command file. In this case **MQSCX** will read commands in from **`stdin`** but in file mode rather than key mode. This means that many of the **MQSCX** features are switched off. For example, the different display panels are not available nor are colours. Essentially the **MQSCX** program looks and behaves very similarly to the **RUNMQSC** program.

14.4 Implicitly obtaining file mode

Attempting to run **MQSCX** on z/OS in an environment that does not have a TTY terminal, will automatically put **MQSCX** into file mode. This will happen when running in Batch, from TSO, or through OMVS. As with the above mode, **MQSCX** will read commands in from **`stdin`**, and the same restrictions mentioned above apply.

14.5 File mode separators

By default the separator used in file mode is the same as configured in interactive mode if the output is to the console window. However, if the output is to a file then the default will always resort to 'msg'. You can use the **`=set separator`** command to change it if you wish.

In file mode the concept of '**`autosep`**' does not exist so separators are always shown.

14.6 File mode features

Most MQSCX features are not available in file mode since they wouldn't make sense. For example, copy+paste, undo/redo, auto complete are all somewhat meaningless when the command is read from a file. However, some features do still apply when running in file mode. These are:

- **=conn**
- **=count**
- **=disc**
- **=find()**
- **=import**
- **=maxresp()**
- **=set**
  Although not all settings make sense in file mode. Note also that any changes will not be saved.
  The settings which may be changed are:
  - **authmsgs**
  - **columns**
    Of course when writing to a file there is no real concept of screen width however, for the sake of columns, the value must be known. By default it has a value of 80 but can be changed using the -w parameter.
  - **colstyle**
  - **colwidth**
  - **cssort**
  - **cswhere**
  - **distotals**
  - **formatting**
  - **listval**
  - **separators**

- **=show**
  Although only a subset of the show command.
  - **licence**
  - **machine**

- **=sort()** and **=sortd()**
- **=syn()**
- **=where()**

14.7 Configuration file

User preferences and settings are only saved when running in file mode if you are using it interactively. That is file mode with **`stdin`** as your input, e.g. interactive in non-terminal environments. You can force a save with the **`-S`** flag.

15 z/OS File name format

Specifying the file name on z/OS can be done in a number of different ways. Examples of the most common are shown here.

To provide the name of a dataset, whether sequential file or partitioned dataset, directly, use the following file name format (note that this can be fully qualified as in the first example, or will have your TSO user ID (e.g. GEMUSER) pre-pended to the name in the second example, depending on whether you use the single quotes or not):

```
=import file(//'GEMUSER.MY.SCRIPT')
=import file(//MY.SCRIPT)
=import file(//'GEMUSER.MQSCX.SCRIPTS(MEMBER)')
```

You might also use this in program parameters:

```
mqscx -m MQG1 -i "//'GEMUSER.MY.SCRIPT'"
mqscx -m MQG1 -i "//MY.SCRIPT"
mqscx -m MQG1 -i "//'GEMUSER.MQSCX.SCRIPTS(MEMBER)'"
```

The examples shown using single quotes do not work from within a PARM string in your JCL because of the single quotes, so when using this from JCL, you must double up the single quotes as shown below:

```
EXEC PGM=MQSCX,PARM=('-m MQG1 -i "//''GEMUSER.MQSCX.SCRIPTS(MEMBER)''"')
```

You can also use DD cards to identify the data set you wish to use. For example, if you have a DD card similar to one of the two following examples:

```
//INFILE DD DSN=GEMUSER.MY.SCRIPT,DISP=SHR
//INFILE DD PATH='/u/gemuser/script.mqx',PATHOPTS=(ORDONLY)
```

you can then use the following:

```
mqscx -m MQG1 -i "DD:INFILE"
```

For more details see "Performing OS I/O Operations" in "z/OS XL C/C++ Programming Guide".

When setting a filepath for the location of scripts on the **=import** command, you can provide a UNIX HFS directory or a PDSE)\ name (or a DD Name referencing the same).

Here are a couple of examples of using a PDSE as a filepath. First we demonstrate the **=set filepath** command.

```
=set filepath(//'GEMUSER.MQSCX.SCRIPTS')
=import file(MEMBER)
```

Now here's an example of using the **-F** parameter on program invocation and a DD name.

```
//MQSCX    EXEC PGM=MQSCX,PARM=('-m MQG1 -F DD:SCRIPTS')
//STEPLIB  DD DSN=MQGEM.LOAD,DISP=SHR
//         DD DSN=IBM.MQ.SCSQANLE,DISP=SHR
//         DD DSN=IBM.MQ.SCSQAUTH,DISP=SHR
//MQGEML   DD DSN=MQGEM.LICENCES(MQGEMLIC),DISP=SHR
//SCRIPTS  DD DSN=GEMUSER.MQSCX.SCRIPTS,DISP=SHR
//SYSIN    DD *
=import file(MEMBER)
*/
//
```

16 MQSCX Parameters

The MQSCX program does not **require** any parameters if you are connecting to the default Queue Manager however in all other cases you will need to pass at least one parameter. The parameters are passed as command flags in the standard fashion.

The list of parameters are:

Flag	Parameter (if any)	File Mode Only?	Description
-c	<value>	Yes	The number of columns to display Default value: auto
-C	<Command(s)>	No	One or more commands which should be issued immediately see below for more information.
-d	<Channel Definition>	No	Channel Definition fields e.g. CHANNEL(MYCHANNEL) This parameter does not apply when the program is running on z/OS. You can of course connect to a z/OS queue manager using this parameter when the program is running on a platform other than z/OS.
-D	<Debugging flags>	Yes	Flags are: **b** - Display variable values before they are assigned **t** - Switch on debug trace
-e		Yes	No command echo
-f		Yes	Run in file mode When running MQSCX on z/OS from JCL, or in an environment without terminal support such as from TSO or through OMVS, this mode is always on, and so this parameter is optional.
-F	<Default file path>	No	Default path used for =import commands Equivalent of =set filepath(...)
-g	<MO71 Config File>	No	If the user specifies this parameter then it tells MQSCX to look for definitions to satisfy connection requests in the file.. This parameter does not apply when the program is running on z/OS. For more information please see 'Using your MO71 Configuration File' on page 121.
-j			Run in JSON CCDT mode This parameter does not apply when the program is running on z/OS.
-l			Connect as a client This parameter does not apply when the program is running on z/OS. You can of course connect to a z/OS queue manager using this parameter when the program is running on a platform other than z/OS.
-i	<file>	Yes	MQSC input file
-m	<Queue Manager>		Queue Manager to administer
-n			Run in CCDT mode This parameter does not apply when the program is running on z/OS.
-p	<Temp. Queue Prefix>		Temporary reply queue prefix For more information about Reply Queues please see section 19.3 Reply Queue on page 136.

-q	<Queue name>		Command Queue name Default value:SYSTEM.ADMIN.COMMAND.QUEUE
-r	<Queue name>		Reply Queue name Default value:SYSTEM.DEFAULT.MODEL.QUEUE For more information about Reply Queues please see section 19.3 Reply Queue on page 136.
-s		Yes	Stop on first command failure.
-S			Force save of config in file mode When running in file mode, importing files where the configuration file would not normally be saved, you can force it to be saved with this flag.
-t	<CCDT file name>		Name of CCDT file if starting in CCDT mode
-v	<Queue Manager>		Via Queue Manager The Queue Manager to connect to in cases where you don't administer the Queue Manager directly.
-V	<CCDT Version>		The version of channels that should be written to the CCDT when in CCDT mode. The parameter is the appropriate MQ version for example, 6.0 or 7.1
-w	<value>	Yes	Display output width Default value: 80 A value of 0 indicates an effective infinite screen width. This causes each response to be output on a single line.
-x			Suppresses any implicit connect. The program will not connect until the first **=conn** call.
?		N/A	Display simple command usage help text

16.1 Initial Command

The **-C** parameter allows the user to pass one or more commands to the program which will be run as soon as the program starts. On most command processors it is necessary to enclose the command in quotes. For example:

```
MQSCX -m MQG1 -C "STOP CHL(X)"
```

To run multiple commands you should separate them with a semi-colon (;) character. For example:

```
MQSCX -m MQG1 -C "STOP CHL(X); STOP CHL(Y)"
```

If the program is being run in file mode and no input file is provided then the after running the initial commands the program will end.

16.2 Channel Definition Fields

If you wish to make a client connection to the Queue Manager the -d parameter can be used to pass in many of the channel definition parameters explicitly. This means that you don't need either the **MQSERVER** environment variable or a Client Channel Definition Table (CCDT).

An example of passing in the definition this way is:

```
MQSCX -m MQG1 -d "CHANNEL(MYCHANNEL) CONNAME(1.2.3.4(1415))"
```

Values are not upper-cased and should therefore be specified in the case that is required.

The channel fields that can be specified are:

Field	Description	Default (if any)
CHANNEL	Channel Name	SYSTEM.ADMIN.SVRCONN
CONNAME	Connection Name	localhost
RCVDATA	Receive Exit User Data	
RCVEXIT	Receive Exit[16]	
SCYDATA	Security Exit User Data	
SCYEXIT	Security Exit	
SENDDATA	Send Exit User Data	
SENDEXIT	Send Exit[16]	
SHARECNV	Share Conversation	10
SSLCIPH	SSL/TLS Cipher Specification	

Specifying any of the channel definition fields on the command line implies that the connection should be made over client bindings. It is therefore not necessary to also specify the **-l** flag.

[16]Only one exit may be specified; chaining is not supported

17 Licensing

Unless you are running a trial or beta version of MQSCX you will need a valid licence file to run the program. A licence file is a simple text file which contains the definitions of the purchased MQSCX licences. This file should only be used for the intended purpose. Licence files should not be transferred or given to third parties.

If a licence file is bought you will be send a file called **mqgem.lic**. It is a simple, human readable, text file which looks a little like this:

```
* MQGEM Software Licence - created Mon Apr 01 17:39:50 2013
product  = MQSCX
version  = Any
issue    = 010413
support  = 310415
email    = john.smith@example.com
licensee = John Smith
contact  = John Smith
location = London, England
machine  = JOHNSMACHINE
userid   = JSMITH
expire   = 310415
authcode = 4C2A-37DB-54CB-3786
```

Some of these fields are optional so the licence file which you have may not contain all the fields. However, let's briefly discuss them all.

- **product**
  This field identifies the name of the product for which this is a licence. For a distributed platform licence this will be 'MQSCX', and for a z/OS platform licence this will be 'MQSCXz'.

- **version**
  This field identifies the version of the program that is licensed. A licence will also authorise all previous versions of the program.

- **issue**
  The date the licence was issued.

- **support**
  The date when support for this licence will cease.

- **email**
  The email address of the licensee. This is usually the email address that the licence was initially emailed to.

- **licensee**
  The name of the person or organisation who owns the licence. This name will be shown in the **MQSCX** program when it runs.

- **contact**
  The name of the person who is associated with the purchase of this licence.

- **location**
  The location where this licence is applicable. This field is optional depending on the type of licence you have bought, see 17.1 Licence Types below.

- **machine**
  This field is optional depending on the type of licence you have bought, see 17.1 Licence Types below.
  If present the licence is only valid on a machine of that name.

You can see the current machine name by showing the help screen in the MQSCX program by pressing F1 or issuing the command **=show scrn(help)**.

- **userid**
  This field is optional depending on the type of licence you have bought, see 17.1 Licence Types below.
  If present the licence is only valid when run by this user id.
  You can see the current user id by showing the help screen in the MQSCX program by pressing F1 or issuing the command **=show scrn(help)**.

- **expire**
  This field is optional depending on the type of licence you have bought.
  If present it shows the date, in DDMMYY format, of when the licence will expire.

- **authcode**
  This is the authentication code which ensures that the previous fields are valid and have not been tampered with. The authentication code must be present and must match the previous fields for the licence to be valid. You should never change the authentication code.

17.1 Licence Types

There are five types of licences which allow different levels of flexibility about who can run the program. Essentially this is controlled by the presence, or not, of the userid and machine fields.

Type	Fields Set	Description
Emerald	userid, machine	This licence must be used by a single userid on a single machine.
Ruby	machine	This licence can be used by any userid on a single machine.
Sapphire	userid	This licence can be used by a single userid on any machine.
Diamond	location	**MQSCX** is supported by any number of users at the same site on any set of machines. The location field gives the location, for example “London, England” of where the licence is based
Enterprise	None	**MQSCX** is supported across your whole enterprise. This means any number of users at any number of locations. An Enterprise licence can be bought by purchasing just 3 Diamond licences.

17.2 Licence File Location

If a licence file is bought you will be sent an ***mqgem.lic*** file. All you need to do is place this licence file in the appropriate place for the **MQSCX** program to find it as detailed in the table.

Platform	Location
Windows and Linux	Same directory as the MQSCX program.
AIX and z/OS UNIX	Current directory
Z/OS	DD:MQGEML

Alternatively you can set environment variable **`MQGEML`** to point to the directory path where the licence file can be found (in which case the name will be assumed to be ***mqgem.lic***), or MVS file or DD name of the licence file.

If a valid licence is found in either file then the licence check will pass.

So, for example, if you use the program in all of TSO, z/OS UNIX and from JCL, you can have one copy of the licence file saved either as a z/OS UNIX file or in an MVS dataset, and refer to it from any environment.

17.2.1 When running MQEV in z/OS UNIX

To refer to the licence file which is stored in an MVS dataset, set the **MQGEML** environment variable using commands like the following:

```
export MQGEML=//'GEMUSER.USER.LIC(MQGEM)'
mqscx -m MQG1
```

17.2.2 When running MQEV interactively in TSO

You can allocate a DD name for use in TSO, using the TSO ALLOCATE command. For more information about this command, refer to z/OS TSO/E Command Reference > ALLOCATE command and Opening files > DDnames.

To refer to the licence file which is stored in an MVS dataset, using a DD name, use commands like the following.

```
ALLOCATE DDNAME(MQGEML) DSNAME('GEMUSER.USER.LIC(MQGEM)') SHR
mqscx -m MQG1
```

To refer to the licence file which is stored in a z/OS UNIX file, using a DD name, use commands like the following.

```
ALLOCATE DDNAME(MQGEML) PATH('/u/gemuser/licences') PATHOPTS(ORDONLY)
mqscx -m MQG1
```

17.2.3 When running MQEV from JCL

To refer to the licence file which is stored in an MVS dataset, configure your JCL as follows.

```
//MQSCX    EXEC PGM=MQSCX,PARM=('-m MQG1')
//MQGEML   DD   DSN=GEMUSER.USER.LIC(MQGEM),DISP=SHR
```

To refer to the licence file which is stored in a z/OS UNIX file, configure your JCL as follows.

```
//MQSCX    EXEC PGM=MQSCX,PARM=('-m MQG1')
//MQGEML   DD   PATH='/u/gemuser/licences',PATHOPTS=(ORDONLY)
```

17.3 Multiple licences

If you have multiple licences then they can be concatenated into a single **mqgem.lic** file. This can be done using simple OS commands such as **copy** or by using your favourite editor.

Alternatively, if you want to keep user configurations totally separate you can use the MQGEML environment variable to point each user to their own separate licence file.

17.4 Changing your licence file

I said before that the authentication code must match the previous fields in the licence, and this is true. However, that doesn't mean that the previous fields are totally unchangeable. You can modify them slightly if you wish in the following manner:

- You can change the case of any of the values.
- You can add or remove white space such as blanks

18 Commands

Commands are entered on the command line just like MQSC commands. All **MQSCX** commands start with an '=' to distinguish them from MQSC commands. You may enter an **MQSCX** command from any panel although the output will only ever be displayed in the MQSC screen. So, if you want to see the output or are uncertain about the correctness of the command you are entering ensure that you are on the MQSC screen.

18.1 Command Help

Before we talk about the commands themselves it is worth mentioning that MQSCX comes with some simple command help text. For example to see a little of commands you can use enter the following:

```
=?
```

This will then display the **MQSCX** commands. Now, suppose we are interested in the **=set** command we could type:

```
=set ?
```

To get a list of the parameters which can be passed to the **=set** command. We can go one further. Suppose we are curious about the **cmdhist** parameter. Try typing:

```
=set cmdhist(?)
```

This will give us a brief description of the parameter and its values.

Of course, this command help text is not meant as a replacement for this User Guide but it can be a useful memory jogger if you can't remember the command or the spelling of its parameters.

Auto-completion also applies to the **MQSCX** commands so if you are uncertain of what needs to be typed next then hitting the <tab> key is often a good reminder.

So anyway.....let's see what commands we can use.....

18.2 =cache

The cache is the retrieved set of object definitions which is used when auto completing the object names in the various MQSC commands. This command allows the user to modify the cache behaviour.

The syntax of the **=cache** command is as follows:

```
=cache [maxagel(<age>)]
       [maxager(<age>)]
       [purge]
```

where the parameters are:

- **maxagel**
  Sets the number of seconds the cache data is considered valid when administering a local Queue Manger before a new set of definitions is retrieved. The default value is 60 seconds.

- **maxager**
  Sets the number of seconds the cache data is considered valid when administering a remote Queue Manager before a new set of definitions is retrieved. A Queue Manager is considered remote is connected through a client channel or administered via an intermediary Queue Manager. The default value is 1800 seconds.

- **purge**
  When this is specified the current contents of the cache is deleted.

18.3 =ccdt

This command is used to put the program into local Client Channel Definition Table (CCDT) mode. After a successful return from this command all future commands will be directed to the local CCDT until a connection is made to a Queue Manager using the =conn command. After this command is entered **MQSCX** will output the name of the CCDT file that is now selected.

This command is not available on z/OS.

The syntax of the **=ccdt** command is as follows:

```
=ccdt [file(<file name>)
      [version(9.0 | 8.0 | 7.5 | 7.1 | 7.0 | 6.0 | 5.3 | 5.2 | 5.1 | 5.0)]
```

where the parameters are:

- **file(<file name>)**
  This parameter gives the name of the CCDT file you would like to edit. If the parameter is not supplied then the 'default' file will be opened. The default file is chosen according to current environment of the **MQSCX** program including the definition of any MQCHLLIB and MQCHLTAB environment variables.

- **version(<MQ Version>)**
  If required the user can set the version of the modified channel records. By default the channel version will be the latest supported by the MQSCX program. However, if you wish to create a CCDT for a back level client then it will be necessary to specify the version of MQ to use.

For more about running in CCDT mode, see *Chapter 9 Modifying the Client Channel Definition Table* on page 81.

18.4 =ccdtj

This command is used when you wish to display, edit and create a Client Channel Definition Table (CCDT) in JSON format. After a successful return from this command all future commands will be directed to the local CCDT until a connection is made to a Queue Manager using the =conn command. After this command is entered **MQSCX** will output the name of the CCDT file that is now selected.

This command is not available on z/OS.

The syntax of the **=ccdtj** command is as follows:

```
=ccdtj [file(<file name>)
```

where the parameters are:

- **file(<file name>)**
  This parameter gives the name of the CCDT file you would like to edit. The file name will have the extension of **.json**. If the parameter is not supplied then the 'default' file will be opened. The default file is chosen according to current environment of the **MQSCX** program including the definition of any MQCHLLIB and MQCHLTAB environment variables.

For more about running in CCDT mode, see *Chapter 9 Modifying the Client Channel Definition Table* on page 81.

18.5 =checkver

This command checks for a later version of the MQSCX program. You must have access to the mqgem.com website for this command to work. The information obtained about a newer release of MQSCX (whether automatically or by this command) will be retained for one week and can be displayed later by showing the help screen (use the command `=show scrn(help)` or press F1).

The syntax of the **=checkver** command is as follows:

```
=checkver
```

18.6 =clear

This command will clear a queue of messages. The command actually operates in two phases. The first phase will issue a normal `CLEAR QLOCAL` command. This is done for efficiency. However, if this call fails because the queue is in-use then this command will move to a second phase which will try to remove the messages explicitly by issuing `MQGET` calls.

```
=clear qlocal(<queue name>)
      [confirm (yes | no)]
      [depth(<queue depth threshold>)]
```

where the parameters are:

- **qlocal(<queue name>)**
  The name of the queue to clear. This should be a local queue which you have the authority to clear.

- **confirm(yes | no)**
  If the command has to resort to getting the messages off manually then it is prudent to ensure that there aren't too many messages on the queue. For example, suppose there are 10,000,000 one would probably want to be warned first rather than just issue 10,000,000 `MQGET` calls. How many is 'too many' though will depend on the environment. By default `=clear` will ask for confirmation if there are more than 1,000 messages on the queue but you can change this threshold will the depth parameter or, indeed, switch off confirmations by using `confirm(no)`.

- **depth(<queue depth threshold>)**
  This parameter allows you to set the depth threshold beyond which `=clear` will ask for confirmation that the clear should proceed.

18.7 =cls

This command allows the user to clear the current screen. The previous data is still available by scrolling or moving the cursor. If you wish to completely remove the previous data then use the =purge command.

The syntax of the **`=cls`** command is as follows:

```
=cls
```

18.8 =colour

This command is used to control the use of colour in the display. You can set the colour of the various screen elements or, indeed, switch off the use of colours.

The syntax of the **`=colour`** command is as follows:

```
=colour [bright(on | off)]
        [colours(on | off)]
        [fg<element>(<colour>)]
        [bg<element>(<colour>)]
```

where the parameters are:

- **bright(on | off)**
  If you find the screen colours are too bright or too dull you can use this setting to adjust them. Set the value to whichever is most comfortable for your monitor and ambient light.
  The default setting depends no the platform. It is off for Windows and on for Unix.
  Note that not all terminals support the concept of 'bright'.

- **colours(on | off)**
  Switches the use of colours on or off. When set off all output will be in the colours of the 'default' element

colour setting.
If your terminal does not support colours, attempting to set colours(on) will inform you that your terminal cannot use colours. Lack of support will also be noted when you view the settings by pressing F4 or issuing the command `=show scrn(sets)`.

- **fg<element>(<colour>)**
- **bg<element>(<colour>)**
  The screen elements are:

Element name	Description
command	The command line at the bottom of the screen
default	The default colour.
error	The colour of error messages
input	The colour of command input lines
msgs	The colour of supplementary messages and the separators
output	The colour of command output lines
prompt	The colour of the prompt
scroll	The colour of the scroll indicators

Each of these elements can have their foreground and background colours changed independently. For example you can change the prompt colours using the command:

```
=colour fgprompt(white) bgprompt(green)
```

The list of colours you can choose from are:

- default
  Note that the 'default' colour is not a default value set by the program but the default colour setting you can set using the fgdefault and bgdefault values. The concept of a default setting is useful since some users prefer command prompts to have a black background and others prefer a white background. By setting a screen element to 'default' you are saying that it should be displayed in the 'default' colour, whatever that is set to.
- black
- blue
- red
- green
- cyan
- magenta
- yellow
- white

If you set the foreground and background colours of a particular element to the same value then the foreground will be set to white unless the background is also white in which case it will be set to black. This ensures that the text is not entirely invisible although, of course, it doesn't guarantee it won't be hard to read!

18.8.1 Colour limitations

The actual colours displayed will depend on a number of factors such as the implementation of the colour library, the emulator used and the monitor. On many systems 'yellow' particularly does not display as very yellow, especially with brightness switched off. Similarly white, when used as a background, is more of a murky grey colour[17].

[17] Improving the colour selection and clarity is a candidate area for improvement in the next version of MQSCX. I would be interested to hear any user views.

18.9 =conn

This command will connect to a Queue Manager. MQSCX can only be connected to a single Queue Manager at any one time. If this command is issued and MQSCX is already connected to a Queue Manager then the existing connection will be put aside for later use. If you subsequently connect to that 'old' queue manager then the existing connection will be used rather than making a new one. This makes it very quick and efficient to bounce between multiple connections. If you wish to disconnect the current connection then you should use the =disc command.

The syntax of the **=conn** command is as follows:

```
=conn [ccdturl(<CCDT URL location>)]
      [certlabl(<Certificate Label>)]
      [channel(<Channel name>)]
      [client]
      [cmdq(<Queue name>)]
      [comphdr(system)]
      [compmsg(rle | zlibfast | zlibhigh)]
      [conname(<Connection Name>)]
      [pwd(<Password> | '*')]
      [qm(<Queue Manager name>)]
      [rcvdata(<Receive Exit User Data>)]
      [rcvexit(<Receive Exit>)]
      [replypfx(<Temporary Reply Queue Name Prefix>)]
      [replyq(<Queue name>)]
      [scydata(<Security Exit User Data>)]
      [scyexit(<Security Exit>)]
      [senddata(<Send Exit User Data>)]
      [sendexit(<Send Exit>)]
      [sharecnv(<Share Conversation>)]
      [sslciph(<SSL/TLS Cipher Specification>)]
      [user(<User Identifier>)]
      [via(<Queue Manager name>)]
      [wait(<Wait Time>)]
```

where the parameters are:

- **ccdturl(<CCDT URL location>)**[18]
  The URL that identifies the name and location of the CCDT file you wish to use for this connection.
- **certlabl(<SSL/TLS Certificate Label>)**[18]
  The SSL/TLS Certificate Label which should be used for this connection. Note that the IBM MQ Client version must be recent enough to support SSL/TLS Certificate labels – MQ V8.0
- **channel(<Channel Name>)**[18]
  The client channel definition channel name.
  Default: SYSTEM.ADMIN.SVRCONN
- **client**[18]
  Indicates that the connection should be made using a client connection.
- **cmdq(<Q*ueue name*>)**
  The name of the queue to which commands are directed.
  If not specified, SYSTEM.ADMIN.COMMAND.QUEUE will be used.
- **comphdr(system)**
  If specified it tells the client connection to compress the message descriptor header.

[18]Does not apply on z/OS

- **compmsg(rle | zlibfast | zlibhigh)**
  If specified it tells the client connection to compress the message data using the algorithm specified.

- **conname(<Connection Name>)**[18]
  The client channel connection name.
  Default: localhost

- **pwd(<Password> | '*')**[19]
  Used in conjunction with the *user* parameter this allows the user to specify a userid and password combination. See User and Password on page 120 for more information.

- **qm(<*Queue Manager name*>)**
  The name of the Queue Manager to administer. If not specified then the default Queue Manager will be assumed.

- **rcvdata(<Receive Exit user Data>)**[18]
  The client channel definition receive exit user data.

- **rcvexit(<Receive Exit>)**[18 20]
  The client channel definition receive exit.

- **replypfx(<Temporary Reply Queue Prefix>)**
  The prefix which should be used to create a temporary reply queue name.
  If not specified, **MQSCX.%U.*** will be used.
  For more information about Reply Queues please see section 19.3 Reply Queue on page 136.

- **replyq(<Queue name>)**
  The name of the local or model queue which should be used for reply messages.
  If not specified, **SYSTEM.DEFAULT.MODEL.QUEUE** will be used.
  For more information about Reply Queues please see section 19.3 Reply Queue on page 136.

- **scydata(<Security Exit user Data>)**[18]
  The client channel definition security exit user data.

- **scyexit(<Security Exit>)**[18]
  The client channel definition security exit.

- **senddata(<Send Exit user Data>)**[18]
  The client channel definition send exit user data.

- **sendexit(<Send Exit>)**[18 20]
  The client channel definition send exit.

- **sharecnv(<Share Conversation>)**[18]
  The client channel definition share conversation setting.
  Default: 10

- **sslciph(<SSL/TLS Cipher Specification>)**[18]
  The client channel definition SSL/TLS Cipher Specification.

- **userid(<User Identifier>)**[21]
  This parameter specifies the connection user identifier and is specified in conjunction with a password. See User and Password on page 120 for more information.

- **via(<*Queue Manager name*>)**
  The name of the Queue Manager to connect to. This parameter is only necessary if the Queue Manager to connect to is different from the one to be administered.

[19]Can be used with any version of MQ but primarily designed for the MQ V8 connection authentication feature
[20]Only a single exit may be specified; exit chaining is not supported.
[21]Can be used with any version of MQ but primarily designed for the MQ V8 connection authentication feature

- **wait(<Wait Time>)**
  When MQSCX connects to a Queue Manager a message is sent to the command server to determine the platform and version of the remote Queue Manager. This parameter allows you to control how long, in seconds, MQSCX is willing to wait for a response before the connection is considered 'failed'.

18.9.1 User and Password

MQ V8 introduced the ability for the Queue Managers to verify a given userid and password combination, passed in at connect time, and reject the connection if the password was not valid. Prior to MQ V8 the values were still accepted but it was up to the user to define a security exit or their own Object Authority Manager (OAM) which would take notice of the parameters. Now that support for userid and password validation is built-in to the V8 Queue Manager the userid and password becomes a very useful way of validating connection authentication.

To take advantage of this **MQSCX** now allows the user to pass in a userid and password combination on the **=conn** command. So, for example you could enter a command of:

```
=conn qm(MYQM) user(MYUSER) pwd(secret)
```

This would work just fine and in many cases it might be just what you wish to do. However, it does have the disadvantage that the password is 'in the clear' in the command stream. You may wish to keep your password more private. To allow this **MQSCX** allows the user to ask for the password to be prompted for. The way you do this is to specify an '*' as the **pwd** parameter. For example:

```
=conn qm(MYQM) user(MYUSER) pwd(*)
```

If this command is issued **MQSCX** will pause the command and try to read the password from the input stream. When characters are typed only asterisk ('*') characters will be echoed to the screen. Password prompting can only be used if **MQSCX** is being used in interactive mode in a TTY terminal. In other words the user is typing in commands at the terminal. If you wish to specify a password in unattended mode then you should put the password in the **pwd()** parameter.

Some limited caching of the userid and password is supported. For each Queue Manager **MQSCX** will remember the last userid and password combination if the connection is successful. This means that specifying the userid and password is only necessary on the first **=conn** command. Specifying an empty **user()** or **pwd()** parameter will clear the cache for particular Queue Manager. Similarly, if the connection fails for security reasons, the cache will also be cleared for this Queue Manager, requiring that on the next attempt to connect, the password must be provided again. Note that the cache is only maintained for the duration of the program, it is not written to a file.

18.9.2 Client Channel Definition Parameters

A number of the parameters allow you to specify the fields of a client channel definition. This can obviate the need for a Client Channel Definition Table (CDDT). If any of the client channel parameters are specified then client binding is assumed and the 'client' parameter is necessary. It is only necessary to specify the parameters which are required, any unspecified parameters will take the default value, if any.

No upper-casing of the parameters is performed so they should be specified in the case that is required.

18.9.3 Using your MO71 Configuration File

MQSCX is capable of reading an MO71 configuration file and using the definitions of the locations in order to construct the channel definitions needed to make a connection to a client location. The way you tell MQSCX you wish it to do so is to specify the **-g** parameter. This flag is followed by the location of the MO71 file. There are a number of formats the parameter can take.

Parameter Format	Effect
-g <Path><File Name>	Full qualified location of the file. The file must exist.
-g <PATH>	Only the path is specified. The file name will be assumed to be *mqmon.cfg*
-g <File Name>	Only the file name is specified. MQSCX will look for the configuration file in • the directory of the MQSCX program[22] • the current directory.
-g $	Means 'look in the default directory'. MQSCX will search : • location specified by MQMONCFG environment variable (if any) • the directory of the MQSCX program[22] • the current directory

Once configured MQSCX will read any client locations from the file and those definitions will automatically be used should you try to connect to them. For example the command:

```
=conn qm(QM1)
```

will look for a location definition for queue manager 'QM1' and, if it is a client location, the channel definition values will be used to try and make a client connection.

18.10 =count

If **MQSCX** is run in file mode then when it ends it reports the number of commands processed and how many of them were successful or failed, in a similar way to standard **RUNMQSC**. However, there is debate about what should be considered a failure. In particular it is not clear whether a **DISPLAY** command, which does not find any results, should be considered a failure. Just because there were no objects to display does not in itself indicate a failure. One might reasonably expect a failure to refer to something like a syntax error. **RUNMQSC** goes even further than this and outputs the message "**One valid MQSC command could not be processed.**" in this instance, indicating that the command wasn't even processed at all where as clearly it was!

To account for these different requirements, **MQSCX** allows the user to control what is counted.

```
=count cmds( yes | no)
       dspcmds(yes | no)
       emptydsp(fail | ok)
       summary(default, fail, failnz, none, ok, oknz, total)
```

where the parameters are:

- **cmds(yes | no)**
  This controls whether MQSCX counts any commands at all. In other words it allows the user to switch off counting and then, possibly, renew counting at some later stage. This can be useful if you have a script which issues a sequence of commands which, although necessary, are not interesting from a summary point of view. For example, you might want to write a script which does a variety of display and alter commands but you only want the final summary to reflect how many new objects were created. ie. How many **DEFINE** commands were issued.

- **dspcmds(yes | no)**
  Whether **DISPLAY** commands should be counted. In many cases **DISPLAY** commands are somewhat

[22]This is not done on AIX since no method exists to determine such a thing.

uninteresting and just used to support the actual operational commands. Consider, for example, a standards script which enforces local policy. This script might issue **DISPLAY** commands against every object in the MQ estate so the number of **DISPLAYs** is not really of interest. What is of interest though is how many **ALTER** commands were issued to enforce the standards.

- **emptydsp(fail | ok)**
  Whether an empty **DISPLAY** command should be considered a 'fail' or 'ok'. **RUNMQSC** will treat empty **DISPLAYs** as a failure however **MQSCX** will, by default, treat them as successful.

- **summary(default, fail, failnz, none, ok, oknz, total)**
  By default **MQSCX** will output a command summary along the lines of:

  MQSC Command summary - Issued: 23 Success: 23 Fail: 0

  However, you can change this behaviour if you wish to only output some of the fields or the field only if it is non-zero (nz). For examples, **summary(total,failnz)** asks that the total is output and the number of failing commands, only if there were any.

18.11 =disc

The **=disc** command will disconnect from the currently connected Queue Manager. This can be useful if you wish to connect and administer a different Queue Manager without ending the MQSCX program.

The syntax of the **=disc** command is as follows:

```
=disc
```

18.12 =echo

The **=echo** command controls what values are 'echoed' to the screen. There are times when you may wish to hide both the input and output of various commands.

The syntax of the **=echo** command is as follows:

```
=echo blank(yes | no)
      cmds(yes | no )
      func(yes | no )
      input(no | setting)
      lang(yes | no )
      langcon(yes | no )
      resp(yes | no )
```

where the parameters are:

- **blank(yes | no)**
  Controls whether blank input lines are echoed.

- **cmds(yes | no)**
  Controls whether commands such as `'DISPLAY QUEUE(*)'` are echoed.

- **func(yes | no)**
  Controls whether commands and control statements are echoed when issued from within a function.

- **input(no | setting)**
  If set to 'no' then all input will be suppressed. If set to 'setting' then the value of the other settings will be used to determine whether the input is echoed.

- **lang(yes | no)**
  Controls whether control language constructs such as 'if ().....while()....' are echoed.

- **langcon(yes | no)**
  Controls whether control language constructs such as 'if ().....while()....' are echoed when entered from the console.

- **resp(yes | no)**
  Controls whether command responses such as the response to `'DISPLAY QUEUE(*)'` are echoed.

The default settings depend on where the lines are generated.

Line Type	Interactive	File Mode (stdin)	File Mode from file	Imported lines
blank	yes	yes	no	no
cmds	yes	yes	yes	Inherited
func	no	no	no	Inherited
input	yes	yes	yes	Inherited
lang	yes	yes	no	Inherited
langcon	yes	yes	no	Inherited
resp	yes	yes	yes	Inherited

If the **=echo** command is used from within a function all the previous echo settings are restored on return from the function. This means that a function can use the **=echo** command without affecting its caller.

In interactive mode the the non-echoed lines can be displayed by using the command **=show noecho(yes)** command. This can be really useful when trying to debug control commands.

18.13 end

Entering a command of **end** will end the program. If running in interactive mode the current settings will be saved to the configuration file. The **end** command is synonymous with **quit**.

The syntax of the **end** command is as follows:

```
end
```

18.14 =import

The =import command allows the user to run a sequence of commands from a file. The commands can be MQSC commands or MQSCX commands. There are many possible used of this command:

- Just a simple list of MQSC commands to run in sequence
- You could have a sequence of commands in a command file which set up your favourite colours or columns widths.
- You could set up a test environment by connecting to multiple Queue Managers and issuing commands to each one.

The syntax of the **=import** command is as follows:

```
=import file(<filename>)
        [cmdhist]
        [parms(Parm1,Parm2,Parm3,...)]
        [quiet]
        [sof]
```

where the parameters are:

- **file(<filename>)**
  The name of the file containing the commands.

The path of the file can either be an explicit name or relative. If it is a relative path then the actual path used will be based on the file name of the file containing the **=import** command, if any. Otherwise it will be relative to the currently set filepath, if one is set, or if not, the current directory.
On z/OS the name can be a DD name, an MVS file or a z/OS UNIX file in HFS. See 15 z/OS File name format on page 107 for more details. If filepath is set to an MVS PDSE then any extension will be stripped from **<filename>** and ignored; the remainder is upper cased, and used as the member name. If filepath is set to an HFS directory (including not set which results in the current working directory for the user), then **<filename>** will be used unchanged.

- **cmdhist**
  If specified the commands read and processed from the command file are added to the command history. By default the processed commands are not added to the command history.
  Command history only applies to interactive mode so this parameter is ignored in file mode.

- **parms(***Parm1,Parm2,Parm3,...***)**
  This parameter allows for a quick way of setting the _Parm[] array. If the imported file uses the _Parm[] array it is recommended that the imported file copies the contents of the _Parm[] array into file specific variables as soon as possible. Remember that variables, including the _Parm[] array, are global and the array could be overwritten by subsequent uses of the array.

- **quiet**
  If specified and in filemode the input commands will not be copied to the output. If MQSCX is running in interactive mode then only the local commands (e.g.=colour) are suppressed. In interactive mode MQSC commands are always displayed.

- **sof**
  Stop on first failure. The import process will stop if any of the commands fail.

18.14.1 Nesting imports

An import command file can itself also contain =import() commands. This can be very convenient for setting up a more complicated environment especially when you consider that the command file can contain =conn and =disc commands.

Let's say we have a simple example Suppose we have two Queue Managers QM1 and QM2 we could define two configuration files such as:

```
=conn QM(QM1)
DEFINE QLOCAL(MYQUEUE1)
DEFINE QLOCAL(MYQUEUE2)
DEFINE QLOCAL(QM2) USAGE(XMITQ)
DEFINE CHANNEL(QM1.QM2) CHLTYPE(SDR)  CONNAME('localhost(1415)') XMITQ(QM2)
DEFINE CHANNEL(QM2.QM1) CHLTYPE(RCVR)
```

File: QM1.mqs

```
=conn QM(QM2)
DEFINE QLOCAL(MYQUEUE1)
DEFINE QLOCAL(MYQUEUE2)
DEFINE QLOCAL(QM1) USAGE(XMITQ)
DEFINE CHANNEL(QM2.QM1) CHLTYPE(SDR)  CONNAME('localhost(1516)') XMITQ(QM1)
DEFINE CHANNEL(QM1.QM2) CHLTYPE(RCVR)
```

File: QM2.mqs

```
=import(QM1mqs)
=import(QM2mqs)
=disc
```

File: setup.mqs

Although you can nest import files you can not recurse. In other words you can not effectively have a loop of files which all reference each other. This would cause an infinite loop and is therefore detected and prevented by MQSCX.

When using nested **=import** files it is recommended you can use relative, rather than explicit, paths in the file names. In the example above no path is is given to the ***QM1.mqs*** and ***QM2.mqs*** files. This implies the files are in the same directory as the ***setup.mqs*** file. Using relative paths makes copying and using **=import** file much easier since a set of import files, which reference each other, can be copied from one directory tree to another without requiring any of the **=import** files to be changed.

18.15 =key

The **=key** command allows the user to assign either an action or a command to a keystroke. Note that either a command or an action may be specified but not both.

To remove a user key assignment enter the command with either an empty action or empty cmd keyword. When deleted the key action will revert to the default behaviour if one exists.

If you wish to remove a default action associated with a key and have the key do nothing then specify an action of 'none'. Keys of this type will not appear in the keys screen.

The syntax of the **=key** command is as follows:

```
=key(keyname) cmd(<command>) | action(<key action>)
              [clear]
              [noenter]
              [nolog]
```

where the parameters are:

- **keyname**
  The name of the key to which the command applies. For example, <Ctrl-x>. The key names are case insensitive. Some keys can not be assigned to alternate actions, for example 'left', to avoid confusion.

  The key name specified may be:

a) A program defined key name
   For a complete list of the key names please see *Appendix C. Key Names* on page 160.

b) A user defined key name
   A value specified on a previous *=keyname* command

- **cmd(<command>)**
  Specifies the MQSC or MQSCX command which should be issued. For example "DISPLAY QUEUE(*)".
  An empty cmd value signifies that the key assignment should be deleted.

- **action(<key action>)**
  Specified the action which should be performed. For example, "Cursor Up".
  An empty action value signifies that the key assignment should be deleted.
  An action of 'none' indicates that this key should be removed from the key assignment table.

  For a complete list of the key actions please see *Appendix D. Key Actions* on page 161

- **clear**
  This parameter is only applicable when creating a key command.
  When specified the current contents of the command line are cleared before the command is issued.

- **name(<key name>)**
  This parameter allows the user to name the key so that a user name is displayed on the keys panel. This is most useful when the program does not know a key or key combination.

For a complete list of the key names please see *Appendix C. Key Names* on page 160.

User defined names must conform to the name rules.
For the valid name rules please refer to *Appendix E.Valid Names* on page 162.

- **noenter**
  This parameter is only applicable when creating a key command.
  When specified the command is copied to the command line but the command is not issued. For example, suppose you frequently issue the command like **dis chs(*) =where(conname lk 'xxx')** however each time you do the IP address you specify in the =WHERE clauses changes. In this case it can be useful to define a key such as **=key(ctrl-i) cmd(dis chs(*) =where(conname lk '')) noenter clear**

  Now, every time you press <Ctrl-i> the command will be shown to you ready for you to change the IP address mask and issue the command.

- **nolog**
  This parameter is only applicable when creating a key command.
  When specified the command is not logged to the output stream. This only applies to MQSCX commands. MQSC commands will always be logged.

18.16 =keyname

The =keyname command allows the user to create or delete a new keyname. A keyname is a name given to a keyboard keystroke. Due to the nature of platform differences, OS versions and emulators it is not feasible for the MQSCX program to come with predefined keys defined for every key stroke the user may enter. You will tend to find that on Windows most keys will already be defined however on Unix it is far more likely that some key combinations are recognised. It all depends on the OS and emulator you are using.

The keyname command allows us to 'fill the gaps'. Any key strokes which are received by the program which aren't recognised can be given a name. This key name can then be used in a **=key** command and the key stroke can therefore be assign to an action or command.

To remove a keyname enter the command with an empty sequence parameter. When a user keyname is deleted the keyname will revert to the default behaviour if one exists.

The syntax of the **=keyname** command is as follows:

```
=keyname(keyname) seq(<key sequence>)
```

where the parameters are:

- **keyname**
  The name of the key to which the command applies. For example, <MyKey>. The key names are case insensitive.

  The key name specified may be:

a) A program defined key name
   This is useful to override the default key assignment of the key.
   For a complete list of the key names please see *Appendix C. Key Names* on page 160.

b) A user defined key name[23].
   A value specified in a name() parameter on a previous =key command
   For the valid name rules please refer to Appendix E.Valid Names on page 162.

[23] If you find that commonplace key combinations you use do not have a key name then please feedback your requirement. It is possible that some key names are common enough that they could be added.

- **seq(<key sequence>)**
  Specifies the sequence of key strokes, one or more, which are received by the program when this key is pressed. The value can be:

1. **A hexadecimal string of the keystroke**
   In the form : 0xHH [(0xH)] - for example 0x87 or 0x1B (0x2)

   If a key is pressed which isn't recognised this hex string will be displayed on the screen.

2. **An escape sequence**
   On some Unixes, depending on the emulator, a key press may actually cause multiple key strokes to be sent to the program one after the other. These sequences always start with the <Escape> key hence the name. If a key is pressed which generates an escape sequence which isn't recognised the program will display the sequence on the screen.

3. **'#'**
   The special value of # tells the program to assign the next key stroke issued by the user. The program will then wait for a key to be pressed. This is useful to avoid having to remember the key sequence being generated. Pressing <Escape> at this point will cancel the **=keyname** command.

18.17 =merge

The **=merge** command is used to merge all or part of another CCDTs with the current one. It can only be issued while in CCDT mode (please see **=ccdt** command). As with any significant editing it is recommended you take a backup copy of all files before starting.

This command is not available on z/OS.

The syntax of the **=merge** command is as follows:

```
=merge ccdt(<CCDT file name>)
       channel(<channel mask>)
       [replace( yes | no)]
```

where the parameters are:

- **ccdt**
  The name of the other CCDT file which will be merged with the current CCDT file.

- **channel**
  This parameter specifies the names of the channel which will be merged. The parameter is mandatory. This is to prevent accidental merging of all channels when only a subset was required. The parameter can take wildcards for example:

  - channel(*) would merge all channels.
  - channel(*TST*) would merge all channels containing the string 'TST'
  - channel(MYCHL) would just merge the single channel 'MYCHL'

- **replace**
  This parameter controls whether the same named channels are replaced. The parameter is optional and if not specified then the same named channels will not be replaced.

18.18 =mqev

This command is used to put the program into MQEV mode. After a successful return from this command all future commands will be directed to the MQEV command queue, `MQGEM.MQEV.COMMAND.QUEUE`.

If you only have an MQEV license then the program will start up in this mode.

The syntax of the **=mqev** command is as follows:

```
=mqev
```

For more about running in MQEV mode, see *Chapter 11 MQEV Event Monitoring* on page 94.

18.19 =mqsc

This command is used to put the program into MQSC mode. After a successful return from this command all future commands will be directed to the IBM MQ command queue, `SYSTEM.ADMIN.COMMAND.QUEUE`.

If you have an MQSCX license then the program will start up in this mode.

The syntax of the **=mqsc** command is as follows:

```
=mqsc
```

18.20 =output

The **=output** command allows the user to redirect a portion of the command stream (and its responses) to a file. By default no output file is active. There can only be one output file. If there is already an output file open then it will be closed when the **=output** command is issued. Issuing **=output** with no parameters will have the effect of closing any current output file is there is one.

The syntax of the **=output** command is as follows:

```
=output [append( yes | no )]
        [file(<file name>)]
```

where the parameters are:

- **append**
  This parameter controls whether any data is appended to the current file. If the parameter is not specified then the file will not be opened in append mode.
- **file**
  The name of the file which is written to. Care should be made to ensure that the correct file name is specified since any current contents of the file will be overwritten if **append(yes)** is not specified. On z/OS the name can be a DD name, an MVS file or a z/OS UNIX file in HFS. See 15 z/OS File name format on page 107 for more details.

18.21 =purge

The **=purge** command allows the user to purge the current output. Note that all the output will be deleted and can not be restored. If you only want to clear the current screen without losing the output history then use the **=cls** command.

The syntax of the **=purge** command is as follows:

```
=purge
```

18.22 quit

Entering a command of **quit** will end the program. If running in interactive mode the current settings will be saved to the configuration file. The **quit** command is synonymous with **end**.

The syntax of the 'quit' command is as follows:

```
quit
```

18.23 =repeat

The **=repeat** command allows you to issue a command repeatedly, every few seconds, and display the results. This command is useful for development and demonstration purposes. It is not proposed that one uses this command to monitor the IBM MQ product.

The syntax of the **=repeat** command is as follows:

```
=repeat [clear]
        [cmd(<command>)]
        [delay(<delay time>]
        [keep]
        [history]
```

where the parameters are:

- **clear**
  Specifies that the screen should be cleared before the commands are issued.
- **cmd(<command>)**
  Specifies the command to be issued.
- **delay(<delay time>)**
  Specifies the number of seconds delay between issuing each command.
  The default value is 10 seconds.
- **keep**
  Specifies that the output should be retained once the command sequence is complete.
- **history**
  By default the =repeat command will display the output of the commands overlaid over each other. However, if you specify the 'history' option then the output will scroll showing a history of all the command outputs.

For example the following command will repeatedly show a queue definition and allow the user to monitor it's depth and usage.

```
=repeat cmd(dis q(q1)) clear delay(1)
```

18.24 =set

The **=set** command is used to set various user preferences. The syntax of the **=set** command is as follows:

```
=set [authmsgs(on | off)]
     [autosep(on | off)]
     [cmdhist(<history>)]
     [colstyle(auto | content | fixed)]
     [columns(<columns> | auto)]
     [colwidth(<width>)]
     [cssort(on | off)]
     [cswhere(on | off)]
     [defmaxresp(<maxresp>)]
     [distotals(on | off)]
     [filepath(<filepath>]
     [formatting(on | off)]
     [history(<history>)]
     [jsonfields(all | min | date)]
     [listval(peritem | one)]
     [mcursor(on | off)]
     [prompt(<prompt>)]
     [respwait(<respwait>)]
     [respwaiti(<respwait>)]
     [separator(msg | none | blank | dotted | solid)]
     [sound(on | off)]
     [timestamp(on | off)]
     [vercheck(no | version | build)]
     [weakciph(on | off)]
     [wrap(on | off)]
```

where the parameters are:

- **authmsgs(on | off)**
  Specifies whether MQSC failed authorisation messages, such as AMQ8135, should be shown or not on a DISPLAY command. This parameter does not apply on z/OS.

- **autosep(on | off)**
  Controls whether separator lines should be dependant on the data context. If 'off' the separator will be shown according to the ***separator*** setting. If *autosep* is 'on' a separator will only be shown if one or more MQ objects spans multiple lines and multiple objects are returned.

- **cmdhist(<history>)**
  Specifies the size of the command history.
  The default value is 50.

- **colstyle(auto | content | fixed)**
  Specified what type of column style you want. The values are:

content	Attribute columns are based on the size of attribute data returned.
fixed	Columns are based on a fixed width which is turn is based on the **columns** setting below.
auto	The data will be formatted using **content** columns provided all responses fit on a single line. Otherwise columns will be formatted as **fixed** width.

- **columns(<columns> | auto)**
  Specified how many columns should be displayed when **fixed** columns are active.
  The default value is **auto**.

- **colwidth(<width>)**
  Specifies the width of the MQSC output columns.
  This value is only used when **fixed** columns are active and the columns value is set to 'auto'.
  The default value is 20.

- **cssort(on | off)**
  Specifies whether the `=sort()` clause should be case sensitive.
  By default the sort clause is not case sensitive.

- **cswhere(on | off)**
  Specifies whether the `=where()` clause should be case sensitive.
  By default the where clause is not case sensitive.

- **defmaxresp(<maximum responses> | unlimited)**
  Specifies the default number of responses that will be shown for any DISPLAY command.
  By default the value is **unlimited**.

- **distotals(on | off)**
  Specifies whether a total line should be displayed at the end of a DISPLAY command output. This total line will state how many items were returned and how many items matched any local filters.

- **filepath(<file path>)**
  Specifies the default path which should be used for =import commands which **are not** fully qualified.
  Normally the filepath parameter would specify a full qualified rather than a relative path. On Windows this means the path should start with the drive letter. On Unix the path should start with a forward slash ('/').
  On z/OS the name can be
  - an MVS PDSE, in which case it should start with a double forward slash (‘//’)
  - a z/OS UNIX file directory in HFS, in which case it should start with a single forward slash (‘/’)
  - a DD name representing the same, in which case it should start with ‘DD:’

See 15 z/OS File name format on page 107 for more details.
The filepath settting is not saved across restarts of MQSCX.

- **formatting(on | off)**
  Specifies whether MQSCX should format the MQSC responses. If set to 'off' then the MQSC responses are shown just as they were received from the command server additional formatting.

- **jsonfields(all | min | date)**
  Set the output format of the CCDT JSON file.
  This parameter does not apply on z/OS.

- **listval(peritem | one)**
  This setting controls whether lists are shown as a single value or whether each list item is returned as a separate value.

- **history(<history>)**
  Specifies the size of the output history.
  The default value is 20000 lines.

- **mcursor(on | off)**
  Specifies whether a mouse click on the command window should move the cursor. This parameter does not apply on z/OS.

- **prompt(<prompt>)**
  Specifies the format of the command prompt.
  For a full description of the format please see section *12.1 Prompt* on page 97.

- **respwait(<respwait>)**
  Specifies how long MQSCX should wait for a response from the command server for an issued MQ command. MQSCX uses an adaptive wait though. If the command server responds with at least one

response within the time period then MQSCX will wait for a further interval. This is useful as some commands can respond with many response messages and therefore even with a low *respwait* value MQSCX will wait until all responses have been received.

- **separator(msg | none | blank | dotted | solid)**
  Specifies the type of separator which should be used when displaying object output.
  For a full description of separators please see section *12.4 Separators* on page 100.

- **sound(<u>on</u> | off)**
  Specifies whether any errors issues by the program should be accompanied by a 'beep' sound.

- **timestamp(<u>on</u> | off)**
  Specifies whether each command in the output display should be displayed with a timestamp.

- **vercheck(no | version | <u>build</u>)**
  Specifies what kind of version check will be done to discover if a newer version of MQSCX is available. If it is set to 'build' then a check for newer versions and newer builds of the same version is made. If it is set to 'version' then you will only be notified of newer version numbers of the MQSCX program.

- **weakciph(on | <u>off</u>)**
  Specifiies whether weak cipher values should be presented when tabbing through all the values in the **SSLCIPH** channel attribute.

- **wrap(on | <u>off</u>)**
  Controls the level of formatting. If set to 'on' then the MQSC responses will not be displayed in columns and will wrap around the end of the screen.

18.25 =show

The **=show** command is used to control what is displayed to the user.

Note that not all options are available in file mode. The syntax of the **=show** command is as follows:

```
=show [func(<function name>) [list]]
      [licence]
      [machine]
      [noecho(yes | no)]
      [scrn(help | keys | mqsc | next | sets)]
      [scroll(on | off)]
```

where the parameters are:

- **func(<function name>) [list]**
  This parameter allows the user to display what functions are defined. If the **list** option is added then the contents of the function is also displayed rather than just it's name and parameters.

  The function name can contain wildcards. So, the command **=show func(*)** will show all defined functions. The command **=show func(*x*)** will show all defined functions containing the letter 'x' in their name.

  Any blank lines in the function definition will just be displayed as a single blank line to conserve screen space.

- **licence**
  Brief information about the licence is displayed. This information is also shown on the help screen.

- **machine**
  Brief information about the machine and any connected Queue Manager is displayed. This can be useful to determine the running user and machine name needed for licensing. This information is also shown on the help screen.

- **noecho**
  This parameter is used to display lines which normally would not be echoed to the screen. It can be useful to debug control commands to determine the execution path or responses from the server.

- **scrn(help | keys | mqsc | next | sets)**
  Sets which screen should be displayed

Value	Meaning
help	Show the help screen
keys	Show the key assignment screen
mqsc	Show the MQSC display
next	Show the 'next' screen. This command cycles round the screens.
sets	Show the settings screen.

- **scroll(on | off)**
  Sets whether the scroll indicators should be shown or not.

18.26 =syn

The **=syn** command is used to set and remove command synonyms. It allows the user to give commonly used commands a shorter nickname.

For example the command:

```
=syn(dc) cmd(DISPLAY CHANNEL(*) ALL)
```

This command will assign a synonym such that any time the command 'dc' is entered the actual command issued is 'DISPLAY CHANNEL(*) ALL'.

To remove the synonym enter the command with an empty cmd value. In this example it would be :

```
=syn(dc) cmd()
```

The current list of synonyms is displayed on the keys screen. For a description of the screens please see *3.4Screens* on page 15.

The syntax of the **=syn** command is as follows:

```
=syn(<synonym>) cmd(<command>)
```

where the parameters are:

- **syn(<*synonym*>)**
  The name of the synonym to be used.
  This value should be 10 characters or less and must be a valid name. The synonym name is case insensitive.

  For the valid name rules please refer to Appendix E.Valid Names on page 162.

- **cmd(<*command*>))**
  Sets the command to be used. The value can be:

Description	Example
An MQSC Command	`=syn(dc)  cmd(DIS CHL(X))`
An MQSCX Command	`=syn(help) cmd(=show scrn(help))`
A list of commands	`=syn(dqa)  cmd(DIS Q(Q1); DIS Q(Q2); DIS Q(Q3))`
A list of synonyms	`=syn(dall) cmd(dc ; dqas)`

Since a synonym can 'point' to one or more other synonyms there is a possibility of a recursive definition. If an attempt is made to run a recursive synonym then an error message will be displayed.

19 Connecting to the IBM MQ Queue Manager

MQSCX supports either connecting directly to the Queue Manager being administered or going via an intermediary Queue Manager. If you are using the 'via' method then clearly channels need to be set up to exchange messages between the two Queue Managers in the normal way.

To connect to the Queue Manager MQSCX supports both local and client bindings. The type of connection made is controlled by

- parameter '-l' passed to the program
- the parameter 'client' used on the *=conn* command which is described in section18.9 on page 118.

19.1 Connecting to a local queue manager

From the command line it would be something like this:

```
c:\windows\> mqscx -m QM1
```

Or if MQSCX is already running and you just want to connect to a different Queue Manager then:

```
=conn qm(QM1)
```

19.2 Connecting to a remote queue manager over a client connection

If connecting as a client then normal MQ rules apply. The client channel configuration can be either

- In a MO71 configuration file identified by the -g program parameter
- Specified on the =conn command itself using channel, conname etc parameters
- MQSERVER variable
- Client Channel Definition Table (CCDT)
- Active Directory (Windows)
- PreConn exit

So, from the command line we could issue a command such as the following:

```
c:\windows\> mqscx -m QM1 -l
```

Or if MQSC is already running and you want to make a new client connection to a different Queue Manager then a command such as the following would work if the configuration is held in, say, the CCDT or MO71 configuration file :

```
=conn qm(QM1) client
```

You could be more explicit if you require with a command such as:

```
=conn qm(QM1) conname(192.0.2.123)
```

19.3 Reply Queue

The default queue used by **MQSCX** is **`SYSTEM.DEFAULT.MODEL.QUEUE`**. You can, if you wish, change this to any other model queue name by using the **`-r`** parameter or **`replyq() =conn`** parameter. Please see the next section for a discussion of the prefix used for the temporary reply queue created from the model queue.

You can, if you really want, use a local queue. However, please bear in mind that the local queue can not be shared amongst multiple **MQSCX** users, each user will need a different local queue. This can make configuration more complicated and it is for this reason that a model reply queue is recommended.

19.3.1 Temporary Reply Queue Prefix

If you are using a model queue then each user of that model queue will get their own temporary dynamic queue created 'on the fly' when the program runs. The model queue should be defined with DEFTYPE(TEMPDYN) so that the reply queue is deleted when the program ends.

You have some control over the name of the dynamic queue by specifying the temporary queue prefix. By default this has a value of **MQSCX.%U.***. This means that reply queues will get created with a format along the lines of

`MQSCX.PAUL.62CFECA421DAB301`

The **`%U`** is replaced with an upper-case version of the userid running the program and the * is replaced with a sequence of characters which ensure that the queue name is unique. The * should be placed at the end of your prefix.

The only other insert you can specify is **`%u`** which will be replaced with the running userid using the character case that is returned by the system.

The temporary reply queue prefix can be specified in the following ways (given in precedence order)

- **`replypfx()`** on an `=conn` command
- **`-p`** program parameter
- **`MQSCX_REPLY_PREFIX`** environment variable
- Default value of **MQSCX.%U.***

19.4 Connection Status

If an error occurs with the a connection, such as the Queue Manager ending or a network failure, then **MQSCX** will display the failure in a status bar at the top of the screen. **MQSCX** will then disconnect from the Queue Manager. Once the Queue Manager is available again then the user can issue an *=conn* command to connect back to the Queue Manager to continue administration.

19.5 Switching connections

MQSCX allows the user to disconnect and subsequently reconnect to either the same or different Queue Manager without ending the program. The way that this is done is by using the *=conn* and *=disc* commands.

This ability to connect to different Queue Managers without requiring the program to end enables the configuration of multiple Queue Managers from a single script with can be very useful. In addition the use of the =import command can make this scenario very easy to configure. Please see section *18.14.1 Nesting imports* on page 124 for an example.

If you use *=conn* to connect to a different Queue Manager then the existing connection is not implicitly disconnected. Instead it is saved so that if that Queue Manager is required again it can be immediately available. This means that switching between different Queue Manager is very fast and efficient. If you really don't want a connection again then you should use the *=disc* command disconnect explicit.

20 Program Settings

MQSCX has many settings which can be tuned by the user. The current values of these are displayed on the settings panel which is display by the command **`=show scrn(sets)`** or by default pressing F4.

The user settings can be changed by issuing one of the following commands:

- =*set* on page 130.
- =*cache* on page 114.
- =*colour* on page 116.

User settings are written to the configuration file except if **MQSCX** is running in file mode. In this way user settings are restored when the program is restarted.

20.1 Multiple Users

If there is more than one user of the MQSCX program then it is possible for them to each have their own program settings. This is achieved by giving each user their own configuration file.

There are three ways to achieve this:

1. Give each user a separate copy of the **MQSCX** program
   By default **MQSCX** will look for a configuration file in the same directory as the program itself.

2. Use the **`MQSCXCFG`** environment variable
   The **`MQSCXCFG`** environment variable can be set to point to the directory where the configuration file should be stored. Please see section *13* file on page 104 for more information.

3. Use the **`MQSCXCFG`** DD Name (z/OS only)
   The **`MQSCXCFG`** DD name can be set to point to the HFS directory or PDSE where the configuration file should be stored. Please see 13 Configuration file on page 104.

21 Performance

The purpose of MQSCX is not to provide a faster MQSC program and generally speaking the performance of MQSCX is comparable with RUNMQSC. However, there is one area where MQSCX has a considerable advantage and that is in writing to the console. This is particularly noticeable on Windows where writing to the console is particularly slow.

The difference lies in the way that the two programs write to the screen. RUNMQSC essentially treats the screen just like a file. It writes each response line to the screen so the user sees a scrolling list of replies. MQSCX on the other hand uses direct screen access. This means that it only needs to write to the screen when it believes it has something to say. So, when issuing a command with a lot of replies it doesn't need to update the screen for every reply but just at the end[24].

Of course results will vary wildly depending on the OS version, command being issued and the machine the command is running on but to give you an idea here's what I measured on my own machine.

21.1 Performance Test

Command Issued : 'dis q(*) all' with 5,000 defined alias queues
OS : Windows 7 SP1
Machine : Intel i5 3.2 GHz 8GB Ram

RUNMQSC : 67 seconds
MQSCX : 1.5 seconds

[24] In actual fact if it is taking a long time to gather all the responses to a command MQSCX will update the screen after the first second to show avoid the user just looking at a blank screen.

22 National Language Support

MQSCX is currently only available in English. If there is sufficient demand for alternative languages then this may be considered in the future.

23 Security

MQSCX is an MQI application, it is written in 'C' and uses the standard MQI to interface with IBM MQ. It does not use any private interfaces into IBM MQ. As such is it is subject to exactly the same security mechanisms as any other MQ program.

Consequently **MQSCX** does not allow a user to do any more than they could do using other mechanisms, such as writing an MQI program themselves. In particular, **MQSCX** is not a setuid program and therefore does not require the user to be a member of the 'mqm' group to run it.

MQSCX always communicates, through queues, to the command server, even when locally connected to the Queue Manager. Therefore, for successful operation:

- **The command server must be running**
- **The user must have the authority to connect and inquire the connected Queue Manager.**
- **The user must have the authority to display the administered Queue Manager.**
- **The user must have MQPUT authority to the SYSTEM.ADMIN.COMMAND.QUEUE**

- **The user must have MQGET and MQPUT authority to the reply queue which is being used.**
  This means that if the reply queue is a model queue, authority to create a dynamic queue will be needed, and if the reply queue is a local queue then the above authorities will need to be granted on the local queue being used.
  If the reply queue is a model queue, which it is by default, then the reply queue will be created with a dynamic queue name of MQSCX.*. This can be useful in identifying queue usage and also creating security profiles specific to MQSCX if required.

  On z/OS you will need MQPUT access to the generic profile MQSCX.*

- **The user must have the authority run the particular command in question.**
  If MQSCX reports that 'The Command Server is slow in responding' and this message never clears then the most likely cause is that the request has been rejected by the command server for some reason. Look on the Dead Letter Queue and see whether there are messages put by the Command Server complaining of a security failure.

23.1 Example security commands for Distributed Platforms

Suppose you have a user, in the group MQSCXADM, who wishes to use **MQSCX**. Below are listed the minimum authorities the user requires in order to use **MQSCX**.

```
SET AUTHREC OBJTYPE(QMGR) +
    GROUP('MQSCXADM') AUTHADD(dsp,connect,inq)

SET AUTHREC OBJTYPE(QUEUE) GROUP('MQSCXADM') +
    PROFILE(SYSTEM.ADMIN.COMMAND.QUEUE) AUTHADD(put)

  * either *

  SET AUTHREC OBJTYPE(QUEUE) GROUP('MQSCXADM') +
      PROFILE(SYSTEM.DEFAULT.MODEL.QUEUE) AUTHADD(get,dsp)

  * or *

  SET AUTHREC OBJTYPE(QUEUE) GROUP('MQSCXADM') +
      PROFILE(MY.LOCAL.REPLYQ) AUTHADD(get,put)
```

23.2 Example security commands for z/OS (using RACF)

Suppose you have a user, in the group MQSCXADM, who wishes to use **MQSCX**. Below are listed the minimum authorities the user requires in order to use **MQSCX**. It assumes that all the classes in use below are active on the system through the necessary switch profiles, and that the profiles used have already been **RDEFINE**-d.

```
PERMIT qmgr.BATCH CLASS(MQCONN) ID(MQSCXADM) ACCESS(READ)

PERMIT qmgr.DISPLAY.QMGR CLASS(MQCMDS) ID(MQSCXADM) ACCESS(READ)

PERMIT qmgr.SYSTEM.ADMIN.COMMAND.QUEUE CLASS(MQQUEUE) ID(MQSCXADM)
ACCESS(UPDATE)

  * either *

PERMIT qmgr.SYSTEM.DEFAULT.MODEL.QUEUE CLASS(MQQUEUE) ID(MQSCXADM)
ACCESS(UPDATE)

PERMIT qmgr.MQSCX.* CLASS(MQQUEUE) ID(MQSCXADM) ACCESS(UPDATE)

  * or *

PERMIT qmgr.MY.LOCAL.REPLYQ CLASS(MQQUEUE) ID(MQSCXADM) ACCESS(UPDATE)
```

24 Return Code

MQSCX will return an integer value to indicate success or failure of the program. The following table lists the possible values.

Please note that the list of return code is subject to change. If you are writing code to examine the reason code then please use a range comparison in addition to an exact comparison.

Success Reason Codes (n < 64)	**Meaning**
0	One or more commands processed successfully, no failures
Warning Reason Codes (64 <= n < 128)	
64	No failures but no messages processed
65	One or more commands processed, one or more MQSC commands failed
Error Reason Codes (128 <= n < 256)	
128	Could not connect to Queue Manager
129	An MQI call failed
130	Out of memory error
131	No MQSC commands were successful.
132	Parameter error
133	Input file error
255	Internal error

25 Problem Determination

If you have a problem when using MQSCX please spend a little time reading the Frequently Asked Questions (FAQ) to see whether your situation is covered in there. It is usually faster to solve your problem yourself. Remember that software fails for a reason and most often it is environmental. This means that most of the time there is some configuration or setting or installed code on your machine which is causing the failure.

Look for error messages produced by MQSCX, by IBM MQ itself and your OS console to see whether there is a message explaining the failure to you. Remember particularly that IBM MQ has a number of places it writes error message. The client error log plus error logs for each of the Queue Managers.

If your situation is not covered in the FAQ then please do raise a question with Support, details of how to do this are given below.

25.1 General frequently asked questions

25.1.1 It says 'Please set required location of configuration file.....' ?

Normally the configuration file would live in the same directory as the MQSCX program itself and that is the default location the program used. However, on some platforms, for example AIX, it is not possible for the program to query the directory it is running from[25]. On these platforms therefore it is necessary for the user to use the environment variable MQSCXCFG to explicitly give the location where they would like the configuration file stored.

25.2 MQ Related frequently asked questions

25.2.1 What does 'Can not find IBM MQ on this machine' mean ?

MQSCX dynamically loads MQ. It does this so that is can dynamically load either the local or client bindings as required. However, that does mean that although the program may load and run it is not able to then subsequently load the required MQ libraries at run time. This can be for a number of reasons:

- You don't actually have MQ installed on the machine. This may seem obvious but you wouldn't be the first person who has tried to run an MQ program without first installing MQ. MQSCX requires either the MQ server or the MQ client or both are installed.

- You are trying to use the 'wrong' binding. Make sure that if you have the server installed, then you are trying to do a local connect. If you have the client installed that you are doing a client connect. Check the '-l' parameter or 'client' parameter on the =conn command.

- You have not issued the setmqenv command. MQ version 7.1 introduced the possibility to install multiple versions of the product and to specify the location. As a consequence you should run setmqenv to notify MQ and applications such as MQSCX where it should find the libraries.

- Check your library path (LIBPATH on Unix and PATH on Windows). It may be necessary to change your library path to include the relevant MQ libraries.

  For example on Unix, export LIBPATH=/usr/mqm/lib64

  Make sure you specify the right directory and point to the 32bit or 64bit libraries depending on the version of MQSCX you are using. For example, if you are running a 32bit MQSCX then you must point it to the 32bit version of the MQ libraries. It is not possible to mix and match, that is not supported by the OS.

If all else fails you can see what library MQSCX is trying to load by issuing the following command sequence:

[25]This is quite surprising. There are many discussion on the web about this 'omission'.

```
export MQACCESS_DEBUG=yes
mqscx -f -Dg
```

The environment variable tells the program to output what files are being dynamically loaded or any error message returned by the system. Since these errors are written to stdout it is necessary to run the MQSCX program in file mode, hence the -f parameter. Finally we run the MQSCX program in debug mode so that it prints out whether it is a 32bit or 64bit program after the initial message, for example:

```
MQSCX Extended MQSC Program - Version 8.0.0
Build Date:Jun  1 2013 64Bit BigEndian
```

25.2.2 Why is my connect failing ?

MQSCX will output the reason code which broadly explains the failure. Unfortunately many of the reason codes can cover many situations which can make diagnosis more tricky. The key thing to remember is that MQ will write out errors to its error logs. So, if you have a failure that you weren't expecting looking at the MQ error logs is a good place to start. Remember that there is a different error log for client connections than for server connections. However, it is probably best to check both just to be sure. Look in the following files.

- <*MQ Installation Directory*>/errors/AMQERR01.LOG
- <*MQ Installation Directory*>/Qmgrs/<*QMNAME*>/errors/AMQERR01.LOG

Common causes for a connection failure include:

- Misspelling or forgetting to set the Queue Manager name
- Connecting using server rather than client bindings (or vice versa)
- Not setting up the necessary client connection security
- Not setting up the correct multi-install environment, using setmqenv,

25.2.3 Why does MQSCX make two connections to the Queue Manager ?

Making two MQ connections is a common approach taken by many MQ applications. Essentially one connection is used for putting messages to the Queue Manager and the other connection is used for consuming messages. This allows the application to remain responsive at all times.

If you are connecting over a client connection then setting SHARECNV to a value of 2 or more will ensure that only a single socket is used for the two connections and therefore overhead is kept to a minimum.

25.2.4 Why do I see 'Command Server or network slow in responding' ?

The answer to this question will probably depend on how often it happens.

- **It has never worked, this is the first time it has been set up**
  The most likely cause here is that you don't have the appropriate authority at the command server and the command messages are being written to the Dead Letter Queue. Examine the Dead Letter Queue for any message indicating a processing failure.

  Another common reason is that the channels to the target Queue Manager are either not started or have been incorrectly configured. Again there could be messages on the Dead Letter Queues indicating a routing failure. Remember to check all Queue Managers involved and channels in both directions.

- **It usually works but I see it occasionally**
  If you only occasionally see this message then it could be that your channels are either not started or are taking some considerable time to start.

- **It has worked but now I see this message all the time**
  The most likely reason here is that one or other of the channels to or from the Queue Manager is down and is not restarting. Check that your channels are configured for triggering.

25.3 Keyboard related frequently asked questions

25.3.1 Why is there an unnatural delay when I press the <ESCAPE> key ?

This is one of those vagaries of Linux which is hard to justify in this day and age. Some Unixes still make extensive use of escape sequences to indicate special characters from the keyboard. To distinguish between these escape sequences and just the user pressing the escape button the curses library will wait some period of time after the escape character to see whether any more characters are forthcoming.

You can modify this wait time using the ESCDELAY environment variable. In my experience setting a fairly low value produces the desired effect. For example:

```
export ESCDELAY=10
```

Of course you may find your environment needs a higher value so set this variable accordingly.

25.3.2 Why do F1, F2, F3 and F4 not work but the other function keys do ?

This can happen when using an emulator which doesn't send the key sequences which the curses library expects. Try changing the terminal emulator to 'xterm-R6' which often gives the correct result.

25.3.3 Why do I get told key or key sequence is unrecognised ?

MQSCX uses the curses library for keyboard input. Although curses has a fairly extensive knowledge of all the different Unix emulators and terminal types it is not feasible for it to know them all. Consequently it concentrates on the most common key combinations. The less common combinations are still given to the **MQSCX** program however they are not decoded by curses but are instead passed in raw form. These key codes are just a sequence of one or more bytes and therefore don't have an inherent name. For example there is no way for **MQSCX** to implicitly know what the escape sequence 0x'1B1B5B347E' means. However, you, the user, know what it means because you pressed the key to generate it. Consequently you can issue an appropriate =keyname command and give this key code a sequence an explicit name.

25.3.4 Paste is not working on Linux

Please see section 6.9 Copy and Paste on page 31 for a discussion on Copy/Paster.

25.4 Display related frequently asked questions

25.4.1 Why is there a delay for a second or two when I resize the screen ?

You may also see the screen clear to a colour during this period. Some of the old versions of the curses library did not support terminal resizing. This means that when the terminal is resized curses itself doesn't know about it. On these systems the way resizing is done is for **MQSCX** itself to catch the resize signal and then inform curses. Unfortunately though there is no simple mechanism to wake the curses keyboard function. The solution to the problem is to upgrade your version of the curses library if possible.

25.4.2 Why are Red and Blue interchanged ?

This is a bug in early versions of curses but was fixed in Version 4.1. If you can, install a later version of curses.

25.4.3 My screen shows everything in the wrong place, why is that ?

Some versions of curses on some platforms don't react to a screen resize dynamically. If you resize the screen then the actual position of the characters on the screen are not where curses believes they are. To fix this all you need to do is hit the ESC key. This causes **MQSCX** to query the actual size of the screen and tell curses explicitly how big it is.

25.5 Command Script file problems

25.5.1 Command script not returning

Check whether you have any while() statements in the program. If you do then ensure that it is correctly coded and that the boolean expression does evaluate to FALSE at the appropriate moment. The most common cause of a script not returning is a badly coded loop.

25.5.2 Responses not as expected

Bear in mind that response variables are always defined but may not have a value. For example, if you issued the following statement:

```
print xxxx
```

This statement would not cause an error, it would just print out a NULL string. The reason being that response variables are always defined but if they are not in the latest response then there are just set to the NULL string. This means you have to be careful that you don't misspell response variables.

25.5.3 Don't forget the debugger

Don't forget that you have simple line Debugging at your disposal. This can be very useful for stepping through your program and inspecting variables and results as you go. Please see Debugging on page 74 for further information.

25.5.4 Debugging functions

If you want to debug a function the don't forget to include the call to the function in your script. You script would look something like this:

```
func MyFunc(.....)
    … … … …
    … … … …
endfunc

MyFunc()
```

25.6 Support

I am sorry you are having problems and need support. Please feel free to email me the details of the problem and I will do my best to help you. Remember that the more complete you make the description of the problem the better chance I have of solving it. The kind of information you should include in your email is:

- Your full name
- The exact version, including build date, of the **MQSCX** you are using.
- The email address and issue date from within your licence file (if you have one)
- The OS platform and version you are using
- A complete description of the problem and how to recreate it. Please include as much detail as possible such as frequency of occurrence. For example, does the problem happen every time or just occasionally. Are there any error messages produced by **MQSCX** or IBM MQ at the time of the problem ?

Once you have gathered this information please email it to support@mqgem.com. I shall reply as soon as possible. Note that priority will be given to customers based on the severity of the problem and the type of licence held.

26 Changes in previous versions

This chapter will give you an idea of the changes that have been made if you have used a previous version.

26.1 Changes made in Version 9.3.1

1. **Support for IBM MQ Command Levels up to 932**

2. **New command =count**
   This new command allows the user to control what is considered a 'failed' command as well as control which values are output upon program termination.
   For a description of this command please see section 18.10 =count on page 121.

3. **Ability to specify comphdr and compmsg on the =conn command**
   You can now directly specify the compression options which should be used for a client connection.
   For a description of this command please see section 18.9 =conn on page 118.

4. **New functions 'trimstr' and 'attrstr'**
   For a description of these new functions please see Appendix B. Expression Functions on page 154.

5. **New functions 'random' and 'seedrand'**
   For a description of these new functions please see Appendix B. Expression Functions on page 154.

6. **Change to the way JSON CCDT escaped characters are displayed.**
   Please see the migration changes described in section 27.1.1 Backward search on page 151

26.2 Changes made in Version 9.3.0

1. **Support for IBM MQ Command Levels up to 930**
   This is includes object changes in queue manager objects at IBM MQ V9.3.0 and a new attribute in MQEV's DISPLAY ACCTQ output as a result of an enhancement to Queue Accounting records in IBM MQ V9.3.0.

2. **Allow setting the temporary reply queue prefix**
   You can now set the prefix which should be used to control the temporary queue which is created from your model queue definition. For more information about Reply Queues please see section 19.3 Reply Queue on page 136.

26.3 Changes made in Version 9.2.1

1. **Support for IBM MQ Command Levels up to 921**

2. **New command =CLEAR**
   This command, similar to the normal `CLEAR` command, will clear a queue of messages. However, this command will resort to removing messages by `MQGET` is the queue is in-use.
   For more information please refer to Chapter 18.6 =clear on page 116.

3. **Support added for hexadecimal numbers in expressions and WHERE clause**
   MQSCX expression can now contain numbers such as 0xAB43D

4. **New system variable _mqevcmdlevel**
   Returns the command level of the MQEV processor.

26.4 Changes made in Version 9.2.0

1. **JSON CCDT editing**
   MQSCX has always been able to create and edit standard IBM MQ CCDT files. However, now you can display, edit and credit CCDT using the JSON format. For more information please see the =ccdtj command and *Chapter 9.3. JSON CCDT file specifics* on page 85.

2. **Support for IBM MQ Command Levels up to 921**

3. **New and changed expression functions**
   Please see *Appendix B.Expression Functions* on page 154 for more information.
   - `valueof()`
   - `power()`
   - `system()` function now has an optional second parameter to determine whether to run synchronously or asynchronously

4. **New system variable, _lastcmdresp**
   This system variable allows you to access any error message associated with the last command issued.

5. **Command substitution improvements**
   You can now have a command in the script such as: `dis ql(<@Name[@index]>)`
   In other words the substitution need not be a simple variable but can now be an array value with a calculated index.

6. **ESC resizes the screen**
   Depending on the curses support on the platform there can be issues resizing the screen where curses does not notice that the screen has been resized and the output is therefore messed up. Pressing the ESC button will now force MQSCX to re-query the screen size and explicitly tell curses what screen size to use.

7. **z/OS Support**
   MQSCX is now available to run locally on z/OS. To enable this you require a z/OS specific licence. A distributed platform licence will not enable MQSCX on z/OS to run.

8. **Force the config file to be written**
   Normally the config is not saved when running in file mode, so that running a script does not affect your settings. However, you can now force it to save even in file mode by using the **`-S`** parameter.

9. **Remove weak cipher values from tabbing in the SSLCIPH field**
   Avoid being presented with weak cipher values when tabbing through all the possibilities in the **`SSLCIPH`** field. Change this setting using **`=set weakciph(on|off)`**

26.5 Changes made in Version 9.1.0

If you are already familiar with MQSCX then you are probably mainly interested in what's new in this version. So, briefly, here is a summary of the changes for this version.

1. **Support reading MO71 Configuration file**
   If you have an MO71 configuration file with all the definitions of how to connect to your favourite queue managers then you can now pick up those definitions by pointing MQSCX at your configuration file. Please see 'Using your MO71 Configuration File' on page 121 for more information.

2. **Connection pooling**
   MQSCX will not implicitly disconnect from a queue manager when you connect to a different one. This makes it far more efficient if your script bounces between two (or more) connections.

3. **MQEV Command Support**
   MQSCX now understands MQEV commands. See Chapter 11 MQEV Event Monitoring on page 94.

4. **MQSCX Program update notification**
   It is always recommended that you run the latest version of MQSCX and get the benefit of new features. To make this easier MQSCX will now automatically check whether there is a new version. You can control this with the `=set vercheck` command. See 18.24 =set on page 130 for more details. Please note that this feature is on by default.

   You can alternatively check for a new version by issuing the command, `=checkver`. See page 115.

5. **Constant variables**
   If you prefer to used named constants instead of numbers to refer to MQ constants, you can now use the `const.MQRC_Q_MGR_AVAILBLE` format. See 7.2.4 Constant Variables on page 39.

26.6 Changes made in Version 9.0.1

1. **Support RESET QSTATS command**
   For some reason lost in the mists of time the Distributed product only supports the RESET STATS command when issued in PCF. This is a shame because, despite it's faults, it can be really handy from time to time. So, prompted by a customer request, MQSCX now will allow you to issue this command in MQSC form.

2. **Support added for certlabl on connect**
   You can now specify a certificate label on a connect command or in the channel definition as a parameter to the program.

3. **New function getkey(<interval>)**
   It is sometimes handy to be able to ask the user a question while the script is running. This function will return the next key press (if any).

26.7 Changes made in Version 9.0.0

1. **IBM MQ Command Level 902 Support**
   MQSCX now knows about attributes which were introduced in command level 902.

2. **foreach(...) loop now operates in CCDT mode**

3. **New iteration system variables loops**
   New system variables **_idxEach**,**_idxItem**,**_idxWhile**,**_numEach**,**_numItem** and **_numWhile** which can make processing loops easier.

4. **Support of CCDT URL**
   MQ V9 allows a connecting application to specify the URL location of the CCDT file to use. This field can now be specified on the **=conn** command.

5. **Support for functions**
   You can now define lists of commands which can be invoked from the command line, other functions or expressions.

6. **Support for GOTO**
   You can now jump to labelled parts of your code.

7. **Automatically loads bootstrap.mqx file**
   So, you can define useful functions which are always available.

8. **Various improvements to the usability of the debugger**
   In addition commands to support functions such as 'sf' to set your current stack frame.

9. **New eval() function**
   This function allows the user to create more dynamic expressions by having the contents of strings evaluated as an expression. For example, print eval(“curdepth > 0”).

26.8 Changes made in Version 8.0.1

1. **IBM MQ Command Level 801 Support**
   MQSCX now knows about attributes which we introduced in command level 801.

2. **IBM MQ Appliance Support**
   MQSCX now knows about the new IBM MQ Appliance platform.

3. **=output command**
   A new command to allow the user to write a portion of the command stream (and responses) to a file

4. **=import parms(...) parameter**
   A parms(...) parameter has been added to the =import command. This allows enabled setting of the _Parm[] array.

5. **Default import file path**
   It is now possible to set the default file path which will be used by the =import command. The path can be set either by a parameter to MQSCX or with the =set filepath(...) command.

6. **New debug trace**
   The parameter -Dt will invoke debug trace in filemode. Debug trace will output the command being processed to the stderr file stream.

7. **wait() parameter added to =conn command**
   This new parameter allows the user to control how long MQSCX should wait for the initial exchange with the command server to determine platform and version.

8. **Cancelling a continuation command**
   In interactive mode you can now cancel a continuation command by clearing the command line when it is already empty. By default then pressing ESCAPE twice will cancel a continuation command.

9. **New =set respwait() command**
   This new command allows the user to set how long MQSCX will wait for a response from the command server.

10. **New _os system variable**
    This new system variable will return a value which indicates which platform MQSCX is running on.

11. **New _cmdlevel system variable**
    This new system variable will return the command level of the currently connected Queue Manager.

12. **New _platform system variable**
    This new system variable will return the platform string of the currently connected Queue Manager.

13. **New functions floor(), ceil() and round()**
    New functions which allow the conversion of a real number to an integer.

14. **New function strreplace()**
    A new function which makes it easier to replace one substring with another.

27 Migration from previous versions

We always try to ensure that, as each version is shipped, all the features that were working on the previous versions remain intact. When installing a new version we always recommend you backup the previous **MQSCX** program and the configuration file MQSCX.CFG. If you do find a problem then please send us a problem report and we will try to fix your issue as soon as possible.

27.1 Migrating from a version prior to Version 9.3.2

27.1.1 Backward search

The command to search backwards through the output has changed from **`/searchtext/-`** to **`/searchtext/b`**. This has been done to reduce the confusion with attribute modifiers.

27.1.2 MQSCX on Windows is now 64-Bit.

IBM MQ has announced that they are removing support for 32-Bit programs in the near future. **MQSCX** on Windows has therefore been converted to a 64-Bit program to ensure continued operation. This does mean, however, that the 64-Bit versions of the IBM MQ DLLs must be made available to **MQSCX** at runtime and therefore this could impact the program operation.

27.2 Migrating from previous builds of Version 9.3.1

27.2.1 JSON CCDT values

In the JSON specification characters such as '\' and '"' must be 'escaped' which means they are written with a preceding '\' character. Earlier versions of **MQSCX** used to display these characters when you displayed the channel object. However, these escape characters take up space and they meant that a field, such as a Security Exit, could not use all of the available space since there were additional escape characters. For this reason, the policy was changed and escape characters are now removed from the data as soon as the field is read from the JSON file. This means that, by default, you no longer see these special characters escaped. It follows, therefore, that the old environment variable of `MQSCX_HIDE_ESCAPE_CHAR` has no effect.

You can, if you wish, get the previous behaviour, and show escape characters, by setting an environment variable `MQSCX_SHOW_ESCAPE_CHAR=y`, before you run **MQSCX.**

27.3 Migrating from a version prior to Version 9.3.1

When running in file mode the command summary has changed with respect to empty **`DISPLAY`** commands. Prior to version 9.3.1 an empty **`DISPLAY`** response was considered a failure. However, from version 9.3.1 onwards an empty **`DISPLAY`** response is considered a success. Instead failures are actual failures such as a syntax error. The `=count` command can be used to change the processing so that empty **`DISPLAY`** responses are treated as a failure as before..

27.4 Migrating from a version prior to Version 9.3.0

The default dynamic reply queue prefix has changed from **MQSCX.*** to **MQSCX.%U.***

This is not expected to cause problems and has the advantage that it allows the Administrator to easily see who is currently using **MQSCX**. However, if you wish to change the prefix you can do so a number of ways. For more information about the prefix please see section 19.3.1 Temporary Reply Queue Prefix on page 136.

27.5 Migrating from a version prior to Version 9.1.0

The **MQSCX** program version checking feature was added in this release, and the default value is build, which means that **MQSCX** will check, on a daily basis, whether there is a newer version of the program. If you don't want to automatically check for new program versions, or if you cannot reach www.mqgem.com from the machine where you run **MQCX**, then you can turn off this feature using the **`=set`** command.

Appendix A. Expression Operators

Here is a list of the available operators. The majority of them can be used in both the =WHERE() clause and in normal control language expressions. Where there are restrictions they will be noted against the operator itself.

Operator	Meaning	Synonyms
	Standard WHERE clause operators	
EQ	Equals	=
NE	Not Equals	<> !=
GE	Greater Than or equals	>=
LE	Less Than or equals	<=
GT	Greater Than	>
LT	Less Than	<
LK	Like – wildcard comparison. (see wildcard note below)	==
NL	Not Like – wildcard comparison. (see wildcard note below)	
CT	Contains	
EX	Does not contain	
CTG	Contains generic (see wildcard note below)	
EXG	Does not contain generic (see wildcard note below)	
	Additional Operators	
=	Equals	**EQ**
==	Wildcard comparison	**LIKE**
!=	Not equals	**NE** <>
<>	Not equals	**NE** !=
OR	Logical OR	|
|	Logical OR	**OR**
||	Bitwise OR	
AND	Logical AND	**&**
&	Logical AND	**AND**
&&	Bitwise AND	
>	Greater Than	**GT**
>=	Greater Than or Equals	**GE**
<	Less Than	**LT**
<=	Less Than or Equals	**LE**
-	Minus	
+	Plus	
*	Multiply	
/	Divide	
%	Modulus	
NOT	Logical NOT	**!**
!	Logical Not	**NOT**

Appendix A.1. Wildcards

Wherever wildcards are allowed the restriction of only having a single wildcard at the end of the string has been lifted. Any wildcarded string can contain as many wildcards as you like and in any position. The character '*' is used to refer to zero or more characters. In addition the character '?' can be used to signify exactly one character.

So the expression below is perfectly valid:

```
=where(descr == '*TEST?? Machine*')
```

Appendix A.2. Operator Synonyms

A number of the operators have synonyms. This is, to a large extent a matter of taste but it can also save you typing. Consider a simple command asking to see all queues with more than 10 messages. The standard WHERE clause would be:

```
display queue(*) where(curdepth gt 10)
```

Note that we have to type spaces around the operator 'gt' to prevent the command processor thinking that there might be an attribute called 'curdepthgt'! If you use symbol rather than letter based operators then there is no such problem.

Using the =WHERE clause we could use exactly the same expression however we can also use this command:

```
display queue(*) =where(curdepth>10)
```

Now, personally I find this easier to understand and it is also less typing since I don't need add spaces around all my operators. Of course if we're trying to save typing we could even do this:

```
dis q(*) =where(cur>10)
```

As I said, it's personal taste. Use which ever style you feel is more natural.

Appendix B. Expression Functions

Here is a list of the available functions the majority of them can be used in both the =WHERE() clause and in normal control language expressions. Where there are restrictions they will be noted against the function itself.

Function	Meaning

attrstr(<string>,<string>)	Returns a string containing the full definition of an attribute or the empty string if the attribute is not present. Suppose we have issued the following command:
	
	`DISPLAY Q(Q1) MAXMSGL MAXDEPTH`
	
	It is entirely possible that `_lastresp` contains
	
	`QUEUE(Q1) TYPE(QLOCAL) MAXDEPTH(50000) MAXMSGL(1000000)`
	
	We can now use `attrstr` to access various parts of this string.
	
	For example, `attrstr("MAXDEPTH",_lastresp)` would have the value
	
	`"MAXDEPTH(50000) "` (note it includes trailing spaces)
	
	The usefulness of this function is that it makes it easy to construct other commands from this response. For example we can say something like:
	
	`@cmd="ALTER QLOCAL(Q2) "+attrstr("MAXDEPTH",_lastresp)`
	
	The variable `@cmd` now would have the value
	
	`ALTER QLOCAL(Q2) MAXDEPTH(50000)`
	
	An alternative use of this function is to be able to remove sections of the original response. For example consider the following:
	
	`@a = _lastresp`
	`@a = strreplace(@a, attrstr("MAXDEPTH", @a), "")`
	
	This sequence will have the effect of removing the attribute from the original string. `@a` will now contain:
	
	`QUEUE(Q1) TYPE(QLOCAL) MAXMSGL(1000000)`
	
	This can be useful when modifying one response to turn it into a new command.

ceil(<value>)	Returns the highest equivalent integer. The main use of this function is to convert real numbers to integers. For example `ceil(7.3)` has the value `8`.
cmpver(<version1>,<version2>)	Compares two version strings and return 1, 0 or -1 depending on whether **version1** is greater, equal or less than **version2**. Examples `print cmpver(“9.3.2”,”9.3.1”)` outputs 1. `print cmpver(“9.3”,”9.3.0”)` outputs 0. `print cmpver(“8.1”,”9.3.0”)` outputs -1

date()
date(<time>)
date(<time>,<format string>)

This function takes 0, 1 or 2 parameters.

The <time> parameter is the number of seconds since January 1st 1970.

If no parameters are given the current time is returned in the default format. If just the time parameter is passed then that time is returned in the default format. If both a time and format is given then the returned value is the time in a formatted string according to the following values for the format string:

Format	Meaning
H	Two digit hour (24 hour clock)
HH	Hour (24 hour clock)
h	Two digit hour (12 hour clock)
hh	Hour (12 hour clock)
M	Two digit minutes
S	Two digit seconds
d	Two digit day of month
dd	Day of month including suffix eg.1st, 2nd, 3rd
j	Julian day of year (zero based)
J	Julian day of year (one based)
m	Three character month name eg.Jan,Feb,Mar
mm	Two digit month
mmm	Full month name eg. January, February,March
P	AM/PM
p	am/pm
y	Four digit year
yy	Two digit year
D	Three character day of week eg.Mon,Tue,Wed
DD	Full character day of week eg. Monday, Tuesday, Wednesday
t	Simple time format eg. 18:14:03
\<char>	Escape character sequence. eg. \m will print 'm'

The default format is : **H:M d/mm/y** eg 17:05 02/04/2014

For example, to print out the current day of the week enter the command:

```
print date(_time,"DD")
```

delvar(<variable>

Will delete the given variable. The variable can be a normal user variable, an array name or an individual element of an array. For example delvar(@a) and delvar(@a[3,4]) are both valid.

eval(<expression>)

This function takes a single expression and returns its calculated value. The normal use of this function is by passing a string which should be evaluations.
For example, **eval("curdepth > 0")**

exists(<variable>)

Returns TRUE if the given variable exists and has a defined value. The variable can be a user or system variable or a user variable array element.
You can not use this function on a response variable.

fclose(<file identifier>)	Closes the file given by the file identifier. The file identifier must have been previously return by fopen()

fgets(<file identifier>,<variable>)	Read the next line in the file given by the file identifier.
	
	The function returns:
	• the length of the line which is read
	• -1 to indicate end of file
	• -2 to indicate a file read error
	
	If a line is read then its contents is placed in the given variable. For example:
	
	`if (fgets(@hf,@line)>=0)`
	`  print @line`
	`endif`

findstr(<string>,<search string>)	Returns the offset in the string of the given search string. The search is case sensitive. If the string is not found 0 is returned. A search for the NULL string (“”) will always return 1
findstri(<string>,<search string>)	Returns the offset in the string of the given search string. The search is case insensitive. If the string is not found 0 is returned. A search for the NULL string (“”) will always return 1
floor(<value>)	Returns the lowest equivalent integer. The main use of this function is to convert real numbers to integers. For example floor(7.3) has the value 7.
fopen(<filename>,<mode>)	Opens a file and returns the file identifier. A value of -1 is returned if the file is not opened. The function is essentially just a wrapper around the C runtime fopen() function so the same modes should work. Essentially though the modes are: • “r” - open file for read • “w”- open file for write - note than any current content will be destroyed • “a” - open file for append If the call returns -1 then the system variable _errno can be used to determine the cause of the failure. The returned file identifier can be passed to fprint, as the first parameter, to write to the file. On z/OS the name can be a DD name, an MVS file or a z/OS UNIX file in HFS. See 15 z/OS File name format on page 107 for more details.

getkey(<interval>)	Returns the next key code typed or 0 if no key is typed within the given interval. The interval is specified in seconds. Note that it is the numeric code in decimal, not the character, that is returned.
	For example on ASCII platforms 'a' =97, 'b'=98, 'c'=99 and so on. The escape key returns 27.
	On EBCDIC platforms, 'a' = 129, 'b'=130, 'c'=131 and so on. The escape key returns 39.
	If you wish to detect the Escape key in an OS independent way, you can use the supplied MQSCX constant as follows:
	`@k = getkey(10)`
	`if (@k = const.KEY_ESC) ...`

lower(<string>)	Return the lower case version of the given string
max(a,b,c,....)	Return the maximum of the parameters There may be any number of parameters or any type.
maxsub(<array variable>) **maxsub(<array variable>,<index>)**	Returns the maximum subscript use on the array for the given index. If an index is not specified the first index, 1, is assumed.

min(a,b,c,....)	Return the minimum of the parameters There may be any number of parameters or any type.
minsub(<array variable>) **minsub(<array variable>,<index>)**	Returns the minimum subscript use on the array for the given index. If an index is not specified the first index, 1, is assumed.
mqtime(<date>,<time>)	Takes the date and time as strings in the format of: "2004-06-17" and "20.28.08" or "20040617" and "202808" and returns the number of seconds since January 1st 1970 For an example of the use of this function please see "Calculating the age of a queue (age.mqx)" on page 61.
numsubs(<array variable>)	Returns the number of subscripts used by the given array. For example, suppose you define two variables @a[10] = 3 and @a[14,27] = "Hi" then nusubs(@a) will return 2 since the maximum number of subscripts used by the variable @a array is 2.
power(x,y)	Returns a real number with the value of x^y For example `power(2,3) = 8.00` and `power(16,0.5) = 4.00`
random(x,y)	Returns a randomly generated number between x and y (inclusive of both x and y). The random generation can be seeded by first calling **seedrand(x)** if required.
round(<value>)	Returns the nearest equivalent integer. The main use of this function is to convert real numbers to integers. For example round(7.3) has the value 7.
seedrand(<value>)	Seeds random number generation. Generate a random number using the **random(x,y)** function.
sort(<array>) **sort(<array>,<column>)**	The sort function will sort a one or two dimensional array into ascending order. With two dimensional arrays you can optionally pass a column parameter giving the column you want to sort by.
sortd(<array>) **sortd(<array>,<column>)**	A sort function just like 'sort()' above but it sorts in descending order.
str(<value>)	Returns the string representation of the value. The main use of this function is to be able to concatenate strings with numbers since without the function the strings are coerced into numbers. For example **`"Hi"+5`** has the value **7** since the length the of string **`"Hi"`** is **2**. However, **`"Hi"+str(5)`** is **`"Hi5"`**.
strlen(<string>)	Returns the length of the given string
strreplace(<string>,<old>,<new>)	Returns a string where all the occurrences of substring <old> are replaced with substring <new> in the string <string>. For example **strreplace("Q.TST",".TST",".PRD")** has the result of "Q.PRD". Passing an empty <old> substring will return <string> as the result. Passing an empty <new> substring will effectively just delete occurrences of <old>.
substr(<string>,<offset>,<length>	Returns the substring for the given length at the given offset. The first character of a string has an offset of 1. If the length parameter is greater than the available number of characters then the returned string is shortened accordingly.
sqrt(<number>)	Return the square root of the given parameter as a real number

<table>
<tr><td>system(<command>)
system(<command>,<options>)</td><td>This function takes 1 or 2 parameters.

Will invoke the system with the given command string. This is used to start other programs. For example:

'q -oLOG -M"event.summary"'

could be used to send an MQ message containing the event summary string, to a logging queue.

The options provided in the second parameter can be:

<table><tr><td>Option</td><td>Meaning</td></tr><tr><td>const.SYNC</td><td>The command is run synchronously and control does not return to the script until the command completes.</td></tr><tr><td>const.ASYNC</td><td>The command is run asynchronously and control is immediately returned to the script, while the command continues.</td></tr></table>
If the second parameter is not specified, the command is run asynchronously.

For more information about using the system() function, see 7.11 Invoking other programs from your script on page 58.</td></tr>
<tr><td>trimstr(<string>)

trimstr(<string>,<string>)

trimstr(<string>,<string>,<string>)</td><td>Returns the string with leading and trailing blanks removed
trimstr(" Hello ") returns "Hello"

Returns the string with leading and trailing characters removed where the characters to be removed are specified in the second parameter.
trimstr(" ---Hello--- ","- ") returns "Hello"

Returns the string with leading and trailing characters removed where the leading characters are given by the second parameter and the trailing characters are given by the third parameter.
trimstr(" <Name> "," <","> ") returns "Name"</td></tr>
<tr><td>upper(<string>)</td><td>Return the upper case version of the given string</td></tr>
</table>

<table>
<tr><td>valueof(<string>,<string>)</td><td>A function which will parse a set of fields of the format “...field1(value)...” and return the value associated with that field. The returned value is always a string regardless of its contents. This can be useful for processing parameters to the program or indeed parsing the response from the queue manager such as:
<code>valueof(“curdepth”, _lastresp)</code>

If any value is itself contained in quotes then these quotes will be stripped from the returned value.

If you need to convert a returned string into an integer then you can use the function <code>eval()</code>. For example <code>eval(“123”)</code> has the value 123. Of course <code>eval()</code> can take any expression so you can pass any valid expression this way.

To determine whether a value has been returned at all you can use the function <code>exists()</code>.

For examples, suppose you had the following string variable :

<code>@s = “a(Hello) b(123) c() d('another')”</code>

The the result of the of the function is as follows:
<table>
<tr><td><code>valueof(“a”,@s)</code></td><td>Has the value “Hello”</td></tr>
<tr><td><code>valueof(“b”,@s)</code></td><td>Has the value “123”
Note that this is a string, not a number.</td></tr>
<tr><td><code>valueof(“c”,@s)</code></td><td>Has the value “”</td></tr>
<tr><td><code>valueof(“d”,@s)</code></td><td>Has the value “another”
Note that the single quotes are stripped.</td></tr>
<tr><td><code>valueof(“x”,@s)</code></td><td>Is <Not Set></td></tr>
</table>
</td></tr>
<tr><td>wait(<number>)</td><td>Waits the given number of seconds before returning. The function returns TRUE.</td></tr>
</table>

Appendix C. Key Names

Below is the full list of key names known to the program. MQSCX uses the curses library and is therefore limited to the keys which that library exposes as known combinations. So, although pressing a certain key on the keyboard may produce a unique key code it may not have a predefined name nor is it guaranteed to be the same on each terminal emulator. MQSCX allows the user to assign their own names to keys or over-ride the standard names using the *=keyname* command.

Key Name	Description (if not obvious)
Enter	Non-assignable
Insert	Non-assignable
Delete	Non-assignable
Backspace	Non-assignable
Escape	*(On Unix)* The CURSES library on Unix inserts a delay when processing an 'escape' key - this is to distinguish it from other function keys. In my experience the delay is excessively long. You can usually tell the curses library to use a different delay using the ESCDELAY environment variable. Before starting MQSCX ensure that you have set this variable something like: `export ESCDELAY=10`
PageDown	
PageUp	
Up	
Down	
Left	Non-assignable
Right	Non-assignable
Ctrl-up	
Ctrl-down	
Ctrl-left	(Windows only)
Ctrl-right	(Windows only)
Home	
End	
Tab	
Shift-Tab	
Num-A1 <===> Num-C3	(Windows only) The 9 keys of the numeric keypad Num-A1 being top left key, Num-C3 being bottom right key
Shift-left	
Shift-right	
Shift-Home	
Shift-End	
Shift-Ctrl-Home	
Shift-Ctrl-End	
Alt-left	(Windows only)
Alt-right	(Windows only)
Ctrl-a <===> Ctrl-z	
Alt-0 <===> Alt-9	(Windows only)
Alt-a <===> Alt-z	(Windows only)
F1 <===> F24	Function Keys

Appendix C.1. Emulator keys

In some cases you may find that a key already has a special meaning in the terminal emulator that you are using. For example, on some Unix emulators function key F1 is caught and used to display emulator help. In these circumstances you should either disable or change the assign key in your emulator or use the *=key* command in MQSCX to associate a different key with the required action.

Appendix D. Key Actions

Below is the full list of actions which can be used with the =key command.

Key Action	Description
Enter Command	Enter the command on the command line
Toggle Insert	Switch between insert and over-write mode
Delete Character	Delete character under the cursor
Backspace	Delete character immediately before the cursor
Clear command line	Clears the command line If the command line is already then empty then any continuation command will also be cancelled.
Next Page	conScrolls forward one page
Previous Page	Scrolls backwards one page
Previous Command	Shows previous command in command history
Next Command	Shows next command in command history
Next Value	Shows the next command value
Previous Value	Shows the previous command value
Undo last change	Undoes the last change
Redo last change	Redoes the last change
Cursor up	Moves cursor up
Cursor down	Moves cursor down
Cursor left	Moves cursor left
Cursor right	Moves cursor right
Cursor Home	Moves cursor to start of line
Cursor End	Moves cursor to end of line
Cursor Top	Moves cursor to top of display
Cursor Bottom	Moves cursor to bottom of display
Select Cursor up	Moves cursor up and selects intervening text
Select Cursor down	Moves cursor down and selects intervening text
Select Cursor left	Moves cursor left and selects intervening text
Select Cursor right	Moves cursor right and selects intervening text
Select Cursor Home	Moves cursor to start of line and selects intervening text
Select Cursor End	Moves cursor to end of line and selects intervening text
Select Cursor Top	Moves cursor to top of display and selects intervening text
Select Cursor Bottom	Moves cursor to bottom of display and selects intervening text
Select All	Select all text If cursor is in command line then selects all command line, otherwise it selects all text
Copy	Copy current selection to clipboard
Cut	Copy current selection to clipboard and remove text
Paste	Paste current selection at cursor position. Cursor must be on command line
Refresh	Refresh screen information including screen size
Find Next	Find the next search string forwards through the output
Find Previous	Find the previous search string backwards through the output
Show Help	Show the help panel
Show MQSC	Show the MQSC panel
Show Keys	Show the Key assignment panel
Show Settings	Show the current user settings panel

Appendix E. Valid Names

A names field must conform to the following rules.

- Names are case insensitive
- Names must not start with a numeric field
- Names may only valid characters, where valid characters are any of:
  - Alphanumeric characters
  - Underscore (_)
  - Minus sign (-)
  - Period (.)
  - Angle brackets (<) and (>)
  - Square brackets ([) and (])

End of document