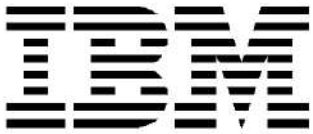

MA95: A Rexx Interface to WebSphere MQ
Version 1.0.2

Take Note!

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

This edition applies to the following product:

Version 1.0.2 of MA95: A Rexx Interface to WebSphere MQ

and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Contents ..... **iii**

Figures ..... **vii**

Tables ..... **viii**

Notices ..... **ix**

Preface ..... **x**

Prerequisites x
Acknowledgements x
Other SupportPacs x

Chapter 1. Introduction **1**

Chapter 2. Installing the SupportPac **3**

Installing on Windows *4*
Installing the DLLs 4
Compiling the Code for Windows 4

Installing on z/OS *6*
Compiling the Code for z/OS 6
TSO Support 7

Chapter 3. Interface Design Philosophy **9**

Chapter 4. General Points **10**

Compatibility with Previous SupportPacs *10*

Return Codes *11*

Last Operation *11*

Return Code Naming *11*

Message Lengths *11*

Header and Event processing *12*

ZLIST *13*

Stem Variables *15*

Rexx Execs *16*

Trace *17*

Chapter 5. Handling MQ Descriptors **18**

The Object Descriptor *20*

The Message Descriptor *21*

The Get Message Option Structure *23*

The Put Message Options Structure *24*

The Variable Length String Structure *25*

Specifying parameters for command interface (RXMQC) *26*

Chapter 6. Thread Support **27**

Initialization *27*

Termination *27*

Connection and Disconnection *27*

Access scope *27*

Shared Variables *27*

Chapter 7. The Interface **28**

Initialization *28*
Description 28
Parameters 28
Call 28
Additional Interface Return Codes and Messages 28
Example 28

Thread Initialization *29*
Description 29
Parameters 29
Call 29
Additional Interface Return Codes and Messages 29
Example 29

Termination *30*
Description 30
Parameters 30
Call 30
Additional Interface Return Codes and Messages 30
Example 30

RXMQCONN *31*
Description 31
Parameters 31
Call 31
Additional Interface Return Codes and Messages 31
Example 32

RXMQDISC *33*
Description 33
Parameters 33
Call 33
Additional Interface Return Codes and Messages 33
Example 33

RXMQOPEN *34*
Description 34
Parameters 34
Call 34
Additional Interface Return Codes and Messages 34
Example 35

RXMQCLOS *37*
Description 37
Parameters 37
Call 37
Additional Interface Return Codes and Messages 37
Example 38

RXMQINQ *39*
Description 39

Parameters 39
Call 39
Additional Interface Return Codes and Messages 39
Example 40

RXMQSET *41*
Description 41
Parameters 41
Call 41
Additional Interface Return Codes and Messages 41
Example 42

RXMQCMIT *43*
Description 43
Parameters 43
Call 43
Additional Interface Return Codes and Messages 43
Example 43

RXMQBACK *44*
Description 44
Parameters 44
Call 44
Additional Interface Return Codes and Messages 44
Example 44

RXMQGET *45*
Description 45
Parameters 45
Call 45
Additional Interface Return Codes and Messages 46
Example 47

RXMQPUT *48*
Description 48
Parameters 48
Call 48
Additional Interface Return Codes and Messages 48
Example 50

RXMQPUT1 ..... *51*
Description 51
Parameters 51
Call 51
Additional Interface Return Codes and Messages 52
Example 54

RXMQC *55*
Description 55
Parameters 55
Additional Interface Return Codes and Messages 55
Examples 56

RXMQBRWS *57*
Description 57
Parameters 57
Call 57
Additional Interface Return Codes and Messages 57
Example 58

RXMQHXT *59*
Description 59
Parameters 59
Call 59
Additional Interface Return Codes and Messages 59
Extracted information 61

Example 63

RXMQEVNT *64*
Description 64
Parameters 64
Call 65
Usage Notes 65
Additional Interface Return Codes and Messages 66
Example 67

RXMQTM *69*
Description 69
Parameters 69
Call 70
Additional Interface Return Codes and Messages 70
Trigger information 72
Examples 73

Appendix A. ISPF Interface 75

Appendix B. Sample REXX execs 78

Figures

Figure 1. TSO Batch JCL **7**
Figure 2. ZLIST and Event processing **14**
Figure 3. A Trigger Monitor **73**
Figure 4. A Rexx Triggered Process **74**
Figure 5. ISPF Exec (MA95T1) **76**
Figure 6. ISPF Panel (MA95P1) **76**
Figure 7. ISPF Panel (success) **77**
Figure 8. ISPF Panel (failure) **77**

Tables

Table 1. Function Names 10
Table 2. Object Descriptor (MQOD) Mappings 20
Table 3. Message Descriptor (MQMD) Mappings 21
Table 4. Get Message Option (MQGMO) Mappings 23
Table 5. Put Message Options (MQPMO) Mappings 24
Table 6. RXMQC Function Parameter Mappings 55
Table 7. Transmission Queue Message Header (MQXQH) Mappings 61
Table 8. Dead Letter Queue Message Header (MQDLH) Mappings 62
Table 9. Trigger Component (MQTM/MQTMC2) Mappings 72

Notices

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

INTERNATIONAL BUSINESS MACHINES CORPORATION 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.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of the intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM 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. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of the information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item has been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

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

WebSphere® MQ

IBM®

AIX®

MVS™

z/OS®

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

Windows® 95, 98, Me

Windows NT, 2000, XP

Preface

This MA95 SupportPac provides a Rexx Interface for IBM WebSphere MQ Version 6.0 for Windows and z/OS. It permits the usage of MQ functions within the Rexx Environment.

This interface is different to that described in the *WebSphere MQ Application Programming Reference Version 6.0 (SC34-6596)* book, as the API is customised for the Rexx environment. However, with a few exceptions, all the function described in the APR is available. Some extensions to the API are also provided to ease the usage of the interface.

The interface described within this SupportPac, should not be taken to be part of the official MQ Product API, nor should the interface itself be considered part of the official MQ product.

Prerequisites

The SupportPac requires:

- z/OS Version 1.6 or later plus WebShere MQ for z/OS Version 6 or 7
- Windows 2000 or Windows XP or Windows 2003 or later plus IBM Object Rexx for Windows Interpreter Edition plus WebShere MQ for Windows Version 6 or 7

What is in this SupportPac:

- REXX Function Pack load module provides Rexx support for WebSphere MQ for z/OS.
- Two DLLs which provide Rexx support for WebSphere MQ for Windows. One DLL supports access to a local Queue Manager, whilst the other DLL provides client access to a server Queue Manager.
- Rexx Execs which demonstrate usage of the interface.
- Source code of SupportPac modules as a reference for creating similar interfaces.
- This paper which documents the interface.

Acknowledgements

This SupportPac is based on the MA18, MA19, MA77 and MA78 SupportPacs written by Robert Harris of IBM Hursley.

Other SupportPacs

This SupportPac replaces the following outdated SupportPacs:

MA18 A SupportPac providing this interface for Rexx and MQSeries for MVS/ESA

MA19 A SupportPac containing a Rexx interface for Rexx and MQ for MVS/ESA for the issuing of **MQSC** commands.

MA77 A SupportPac providing this interface for Rexx and MQSeries for Windows NT

MA78 A SupportPac containing a Rexx interface to MQSeries for Windows NT for the issuing of **MQSC** commands.

Chapter 1. Introduction

This SupportPac provides a Rexx interface, within the z/OS and Windows environment, for WebSphere Message Queueing access.

One **IRXFUSER** load module is provided for z/OS environment. It is the so-called user REXX function pack which, when placed to the commonly available load library, allows the user to call all the provided interface functions without additional preparations. This load module is automatically placed in memory during LOGON, and all interface functions become readily available for use.

Alternative setup may be used, if it is not possible to exploit REXX function pack advantages. SupportPac load library contains alias names for every exported function, and user should issue **TSOLIB** command against this library to make them available for REXX procedures. The first call to **RXMQINIT** function takes care of loading the executable into memory, and the last call to **RXMQTERM** unloads the interface.

Two Dynamically Linkage Libraries (DLLs) are provided for Windows environment:

RXMQN A DLL that provides access to a Queue Manager located on the same workstation

RXMQT A DLL that provides access to a remote Queue Manager located on the different workstation via WebSphere MQ Client

Both DLLs contain exactly the same function, and has the same interface, however, both of them cannot be used in one REXX program simultaneously, because they are linked with different MQ API libraries. See below how to properly install the interface.

A full implementation of the API as described in the *WebSphere MQ Application Programming Reference Version 6.0 SC34-6596* (referenced below as APR) is provided, so this book will be needed to use the Rexx Interface. However, there are a few restrictions:

- *MQINQ* only permits a single attribute to be examined, as support for multiple access is too complicated in the Rexx environment
- *MQSET* only permits the setting of a single attribute
- *MQCONNX* and *MQBEGIN* functions are not implemented

In addition to the standard API functions, the Rexx Interface provides a number of extensions to the API to ease the coding with the Exec:

- Browse function is provided
- Header Extraction function is provided to split up a message from a Transmission Queue or a Dead letter Queue into its components
- Event Interpretation function is provided to split up a message from an Event Queue into its components
- Trigger Message function is provided to split up a Trigger message from an Initiation Queue and to generate/parse execution parameters
- Rexx Interface for manipulation of MQ objects (the function that is provided via the MQSC command).

Implementation of the last function is according with the *WebSphere MQ Script (MQSC) Command Reference Version 6.0 SC34-6597*, so this book will be needed to use the Rexx Interface. However, bear in mind:

- The + - line extender symbols are not required or supported
- The underlying function is provided by use of the PCF ESCAPE command (see *WebSphere MQ*

Programmable Command Formats and Administration Interface Version 6.0 SC34-6598 for details)

- Some non-printable characters are returned.

Chapter 2. Installing the SupportPac

Take the following actions to install the SupportPac

1. Save the file **MA95.zip** to a temporary directory.
2. Uncompress using any unzip program keeping the original directory structure. This will produce the following directories:

MA95\

bin\	contains the binary ready-to-use files of SupportPac
doc\	contains this **MA95.pdf** file that you are reading
MVS\	contains MVS specific files
ASM\	MVS Assembler source module (REXX function package)
Include\	MVS specific include files to use during recompile
JCL\	job samples for recompiling and running tests
samples\	contains sample REXX programs to show the interface usage
src\	contains source code of SupportPac modules common to Windows and MVS
Windows\	contains Windows specific files
Compile\	contains files necessary to compile on Windows
Runtests\	contains batch files for running samples
Util\	contains files necessary to recreate REXX constants definitions

Installing on Windows

Take the following actions to install the SupportPac on Windows:

Copy **rxmqn.dll** or **rxmqt.dll** from **MA95\bin** into a suitable directory contained in the **PATH** entry of the System -> Advanced -> Environment Variables (obtained via the Control Panel). If you want to restrict usage, then place in the User variables section, but it is recommended to place DLL in the System Variables section. Do not try to place both DLLs in the same commonly accessed directory. If both DLLs are called from the same REXX program the results are unpredictable.

Installing the DLLs

To use the Rexx WebSphere MQ function within a Rexx Exec, the relevant DLL must be made known to Rexx, and then the interface initialized.

To load the Local Queue Manager interface, code:

```
rcc = RxFuncAdd('RXMQINIT', 'RXMQN', 'RXMQINIT' )
rcc = RXMQINIT()
```

To load the Client/Server interface, code:

```
rcc = RxFuncAdd('RXMQINIT', 'RXMQT', 'RXMQINIT' )
rcc = RXMQINIT()
```

Once this initialization sequence has been done, then the Rexx MQ interface is ready for use.

Please note these operational characteristics:

- The **RxFuncAdd** operations need only be done once for the whole of the Object Rexx environment.
- The **RXMQINIT** call need to be done for each REXX EXEC run (ie: each Process using the interface).
- The **RXMQTERM** call removes the definitions of the functions (so, if two EXECs were running together, the second would start failing after the first issued the **RXMQTERM** call). Consequently, these may be omitted under normal circumstances.

  In a thread based environment, a thread should be initialized via **RXMQCONS** (**RXMQINIT** should only be called in the initial process thread)

Compiling the Code for Windows

If you want to add function to the interface (or bring it up to current WebSphere MQ level), then you may choose to compile the code. You will need the Object Rexx Toolkit along with the WebSphere MQ libraries and includes.

Two batch files are provided to compile the new version of the interface using the Microsoft Development Studio for Visual C++:

- **MA95\Windows\Compile\make_n.bat** to create a DLL that provides access to a Queue Manager located on the same workstation
- **MA95\Windows\Compile\make_t.bat** to create a DLL that provides access to a remote Queue Manager located on the different workstation via WebSphere MQ Client

Before starting the compile ensure that the C++ source code and headers in **MA95\src** are in the INCLUDE path as well as WebSphere MQ and Object Rexx Toolkit headers and library files.

Another task you may want to do is upgrading WebSphere MQ constants defined in REXX to a higher level. These constants are defined in **MA95\src\RXMQCONS.hpp** file which can be recreated using the following instructions:

- Place current WebSphere MQ include files **CMQC.H** and **CMQCFC.H** to the same directory as **MA95\Windows\Util\makecons.bat**
- Make changes, if desired, to **MA95\Windows\Util\rxmqcons.cfg** which contains the list of prefixes of constants to be pulled out of the WebSphere MQ include files
- Run the batch file
- Recompile the interface DLLs (as described above) or MVS function pack (as described below)

Installing on z/OS

Take the following actions to install the SupportPac from the unpacked **MA95.ZIP** file:

- **MA95\bin\MA95.xmt** contains the load library to be transferred to z/OS system. It needs to be transferred to the destination TSO system as a sequential binary file with a record format of FB 80.
- To send the file via ftp open the session with the target host. Ensure the **BINARY** option is set, then use the following commands:
  quote site fixrecfm 80

  put MA95.xmt MA95xmit
- On TSO, issue this command:
  receive indsname(MA95XMIT)
- When prompted for a filename, reply

  dsn(MA95.LOAD)
  - This creates a PDS called mvsuserid.**MA95.LOAD** with SupportPac executables
  - Use ISPF 3.2 to delete the **MA95XMIT** file
  - Use ISPF 3.3 to copy all members from mvsuserid.**MA95.LOAD** into the load library of your choice. If you want other users to access the MA95 interface, select a common access load library (like LINKLIB concatenation). SupportPac is automatically recognized by TSO as a REXX function pack during LOGON, and interface functions become readily available for use. Otherwise use a private load library, which may be made accessible from your TSO/ISPF sessions by the use of **TSOLIB ACTIVATE DSNAME(MA95.LOAD)** command.
  - To permit tracing to appear, ensure that the following DD-statement is in your TSO LOGON Procedure (or dynamically issue a TSO **ALLOCATE FI(SYSPRINT) SYSOUT** command)
    //SYSPRINT DD SYSOUT=*

The MVS file names have been described are an example. Please use whatever conventions are suitable for your installation.

To use the Rexx WebSphere MQ function within a Rexx Exec, nothing special has to be done to make it known to Rexx (compared to Windows **RxFuncAdd** call).

Compiling the Code for z/OS

If you want to add function to the interface (or bring it up to current WebSphere MQ level), then you may choose to compile the code. You will need IBM C/C++ Compiler and Language Environment and High Level Assembler for z/OS along with the WebSphere MQ libraries and includes. Another task you may want to do is upgrading WebSphere MQ constants defined in REXX to a higher level. The latter task is described in Compiling the Code for Windows on page 4.

To assist with re-building the code, sample jobs are provided in **MA95\MVS\JCL** directory. They are:

- **ASM95** – use this one to assemble REXX function package part of SupportPac
- **CC95** – use this one to compile C++ part of SupportPac
- **LKED95** – use this job to bind Assembler, C++ parts and WebSphere MQ libraries into load module.

These sample jobs refer to WebSphere MQ, C/C++ and Language Environment libraries (which names you may need to update for your installation) as well as SupportPac libraries you will need to create. It is recommended to use PDSE type libraries. Suggested names and attributes for them are as follows:

- **WMQ.MA95.ASM - RECFM=FB,LRECL=80**

- **WMQ.MA95.CC - RECFM=VB,LRECL=255**
- **WMQ.MA95.OBJ - RECFM=FB,LRECL=80**
- **WMQ.MA95.LOAD - RECFM=U,LRECL=32760**

First, you will need to transfer the source code from **MA95\src** to **WMQ.MA95.CC** on your z/OS system. Then transfer MVS specific code from **MA95\MVS\ASM** to **WMQ.MA95.ASM** and **MA95\MVS\Include** to **WMQ.MA95.CC**. Now make necessary changes to the code and submit 3 sample jobs to build the load module.

TSO Support

TSO Batch Support

The **MA95** interface will run in Batch mode via **IKJEFT01**, using a setup such as:

```
//LIB        EXEC PGM=IEBGENER
//*
//*                 Create the exec library
//*
//SYSUT2     DD     DSN=&&LIB(SILLY),DISP=(NEW,PASS),
//                  UNIT=SYSDA,
//                  SPACE=(CYL,(1,1,10)),
//                  DCB=(DSORG=PO,RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSPRINT   DD     DUMMY
//SYSIN      DD     DUMMY
//SYSUT1     DD     DATA,DLM='##'
/* A Silly Exec */
RXMQINIT()
say 'WMQA is very very very very silly'
RXMQTERM()
exit 0
/* End of SILLY exec */
##
/*
//RUN        EXEC PGM=IKJEFT01
//SYSPROC    DD DSN=&&LIB,DISP=SHR
//STEPLIB    DD DSN=WMQ.MA95.LOAD,DISP=SHR
//SYSTSPRT   DD SYSOUT=*,DCB=(RECFM=F,LRECL=132,BLKSIZE=132)
//SYSPRINT   DD SYSOUT=*
//SYSUDUMP   DD SYSOUT=*
//SYSTSIN    DD *
SILLY
/*
```

Figure 1. TSO Batch JCL

TSO Native Support

When running within a native TSO Exec (ie: one issued outside of the ISPF environment, or via the TSO command within ISPF), the Rexx processor is attached to TSO as a separate TCB. Consequently, the connection to the Queue Manager will only last throughout the lifetime of the Exec. If processing is interrupted via PA1, then the Rexx processor TCB is terminated, and so the MQ Step termination routines will be driven to terminate all extant accesses.

TSO Split Screen Support

When running within a ISPF Split Screen, the Rexx processor is attached to each part as a separate TCB.

Consequently, the same considerations apply as in “TSO Native Support,” so the Queue Manager connection, and the Rexx variables cannot be shared between the two halves.

Chapter 3. Interface Design Philosophy

The Rexx MQ Interface API differs from that defined in the APR. This is because the call-type of API is not suitable for the Rexx environment. This has been replaced with a set of verbs that use Rexx Stem variables to contain the relevant information.

The opportunity has also be taken to remove some parameters due to the restriction that a single Windows thread or z/OS task (Exec in the Rexx environment) can only communicate with a single Queue Manager. Additionally, in order to simplify coding, Input and Output versions of object are provided (this saves deleting and rebuilding things like Message descriptors which are updated by a MQ Verb).

As part of the initialization call, a number of MQ Constants (as described in *WebSphere MQ Constants Version 6.0 SC34-6607*) are defined to the Rexx workspace. Thus, you will be able to code options according to the descriptions in this book. However, these values are not protected against change, so you should avoid using your own variables starting with MQ.

The command interface takes a string of characters in the format of a text command as described in the MQ Command reference. This is then placed in the Command Input Queue in the format of a PCF ESCAPE command. The Command server processes the command, and returns the results in a dynamically generated ReplyToQ (called **`RXMQC.*`**) based upon the *ModelReplyToQ* specified. Each message is returned as an element of a Rexx Stem variable, with the number of messages being in stem.0 in the usual Rexx fashion. These returned messages may have non-printable characters in them.

Chapter 4. General Points

Compatibility with Previous SupportPacs

The previous SupportPacs, namely, MA18/19 for MVS/ESA and MA77/78 for Windows, used different names for the same implemented WebSphere MQ functions. That is why it was not possible to write a single REXX exec to execute in both environments.

In order to provide this capability the function names were unified to provide for running the same REXX exec unchanged on z/OS and Windows. The MA95 Common Function names are presented below in the table below. Sample REXX execs which run equally in both environments are provided in **MA95\samples** directory and explained more in Appendix B. Sample REXX execs on page 78.

To maintain compatibility with previous SupportPacs the old function names are still serviced, so that existing REXX execs are not required to be changed. For example, the calls to **RXMQN…/RXMQT…** in Windows and **RXMQV** in z/OS are supported and should provide the same results as new Common Functions names. The parameter lists remain unchanged.

In the rest of this document only the new Common Function names (**RXMQxxxx**) are described. See the table below for relationship between new and old function names specified in older manuals.

Table 1. Function Names

MQ Function Name	MA95 Common Function	MA77/78 Windows Function	MA18/19 MVS/ESA Function	Page
Standard MQ functions:				
MQBACK	**RXMQBACK**	**RXMQNBACK**	**RXMQVBACK**	44
MQCLOSE	**RXMQCLOS**	**RXMQNCLOSE**	**RXMQVCLOSE**	37
MQCMIT	**RXMQCMIT**	**RXMQNCMIT**	**RXMQVCMIT**	43
MQCONN	**RXMQCONN**	**RXMQNCONN**	**RXMQVCONN**	31
MQDISC	**RXMQDISC**	**RXMQNDISC**	**RXMQVDISC**	33
MQGET	**RXMQGET**	**RXMQNGET**	**RXMQVGET**	45
MQINQ	**RXMQINQ**	**RXMQNINQ**	**RXMQVINQ**	39
MQOPEN	**RXMQOPEN**	**RXMQNOPEN**	**RXMQVOPEN**	34
MQPUT	**RXMQPUT**	**RXMQNPUT**	**RXMQVPUT**	48
MQPUT1	**RXMQPUT1**	**--**	**--**	51
MQSET	**RXMQSET**	**RXMQNSET**	**RXMQVSET**	41
Extension functions:				
Initialization	**RXMQINIT**	**RXMQNINIT**	**RXMQVINIT**	28
Thread initialization	**RXMQCONS**	**RXMQNCONS**	**RXMQVCONS**	29
Termination	**RXMQTERM**	**RXMQNTERM**	**RXMQVTERM**	30
Command function	**RXMQC**	**RXMQCNC**	**RXMQVC**	55
Browse	**RXMQBRWS**	**RXMQNBROWSE**	**RXMQVBROWSE**	57
Header Extraction	**RXMQHXT**	**RXMQNHXT**	**RXMQVHXT**	59
Event Extraction	**RXMQEVNT**	**RXMQNEVENT**	**RXMQVEVENT**	64
Trigger Extraction	**RXMQTM**	**RXMQNTM**	**RXMQVTM**	69

MQ Function Name	MA95 Common Function	MA77/78 Windows Function	MA18/19 MVS/ESA Function	Page
Universal function caller	--	--	**RXMQV**	--

Warning. Some minor incompatibilities may still exist between the old and new SupportPacs due to optimizations made during the code merge. For example, small changes were made to improve EVENT queue message parsing algorithm (see RXMQEVNT on page 64). If you encounter a problem, please contact the authors to get workaround or have your suggestion included in the next release of SupportPac.

Return Codes

All the **RXMQ** functions return a standard Rexx return string. This is structured so that the numeric Return Code (which may be negative) is obtained by a **word(RCC,1)** call.

The Return Code for an operation can be negative to show that the interface has detected an error, otherwise it will be the MQ Completion Code (not the uninformative Reason Code).

The Return String is in text format as follows:

Word 1 Return Code

Word 2 MQ Completion Code (or 0 if successful)

Word 2 MQ Reason Code (or 0 if successful)

Word 4 **RXMQ** function being run

Word > OK or an helpful error message

Last Operation

In addition, the current (ie: the settings last set) values are available in these variables:

RXMQ.LASTRC current operation Return Code

RXMQ.LASTCC current operation MQ Completion Code

RXMQ.LASTAC current operation MQ Reason Code

RXMQ.LASTOP current operation **RXMQ** function name

RXMQ.LASTMSG current operation Return String

Return Code Naming

A set of variables called **RXMQ.RCMAP.nn** are also placed in the workspace, where nn is the MQ Reason Code. These variables can be used to turn a return code number into the defining string.

Thus:

```
rcc =   '2048 2 2048 RXMQPUT ERROR'
interpret  'fcs = RXMQ.RCMAP.'word(rcc,1)
/* fcs = MQRC_PERSISTENT_NOT_ALLOWED */
```

Message Lengths

When a *MQGET* is performed, if the buffer size is too small for the message, then the returned message length is the **real** length of the message, not the smaller size which fits in the buffer (see *DataLength* for *MQGET* in the APR).

Consequently, if you specify a too small a message length, and do not take any notice of the return code

indicating truncation, then the length of the message in stem.0 will be **bigger** than the message in stem.1. This will result in a mysterious loss of data in the message.

Header and Event processing

Functions **RQMQHXT** and **RXMQEVNT** will take messages and split them up into the contained components. These exploded components may clash with those for the Message Descriptor (or other like things). Therefore, use different **`stem.`** names to avoid this possibility.

ZLIST

One of the problems with REXX **stem.** variables is that it is difficult to know what components (things after the .) are associated with the **stem.** You have to know which ones might be around, and then test with something like:

```
if ( stem.comp1 <>  'STEM.COMP1'  ) then say  'comp1 =/'stem.comp1'/'
if ( stem.comp2 <>  'STEM.COMP2'  ) then say  'comp2 =/'stem.comp2'/'
```

To get around this problem, the **output** descriptors will contain a component called **ZLIST**. **ZLIST** will contain a list of words, each word a component name which is attached to the stem variable. You can then use the Rexx words (to get the number of elements) and word (to extract the component name) functions to manipulate the stem. variable. **ZLIST** does not contain itself (ie: **ZLIST** is not within **stem.ZLIST**).

The presence of an item in **ZLIST** implies that the relevant Stem.Component is **defined** as a Rexx Variable. However, the contents may be null (a length of zero or set to '') depending upon what the underlying MQ object contains.

This facility is not of much use for the **RXMQOPEN**, **RXMQGET** and **RXMQPUT** calls (wherein **ZLIST** is provided for the Output Object Descriptor, Output Message Descriptor, Output Get Message Options and Output Put Message Options) as the contents of the Output **stem.** variable is of fixed format. However, it can be used to display the **stem.** variable and can also be useful in copying operations.

For **RXMQHXT** and **RXMQEVNT** processing, **ZLIST** is of variable format, containing things relevant to the Message or Event being processed. **ZLIST** for **RXMQHXT** processing contains components 0 and 1 (the original message) as well as **NAME** and **TYPE**. For **RXMQEVNT** processing, **NAME**, **TYPE** and **REA** are always present; the rest of the list will depend upon the event being processed.

For example to display an Object Descriptor:

```
drop iod. ; drop ood.
iod.on = 'N1'
iod.ot = MQOT_Q

rcc = RXMQOPEN('iod.', mqoo_inquire, 'h1', 'ood.')
say 'RC=' rcc 'H=' h1
do j=1 to words(ood.zlist) ; k = word(ood.zlist,j) ; say k '/'ood.k'/' ; end
```

ZLIST can be used for Event processing:

```
drop bm. ; drop ed.
rcc = RXMQBRWS( he, 'bm.')
say 'Browse RC=' rcc

rcc = RXMQEVNT('bm.', ' ed.')
say 'Event RC =' rcc
say '.zlist  /'ed.zlist'/'

/* Protect against bad function by being very cautious! */
if ( (ed.zlist <> 'ED.ZLIST') & (words(ed.zlist) <> 0) ) then ,
    do j=1 to words(ed.zlist)
       k = word(ed.zlist,j)
       say 'ed.'k' /'ed.k'/'
       end
     end
/* I'm only interested in Channel Stopped Events  */
/*                                                */
/* However, do not want to access undefined       */
/*         components.                            */
/*                                                */

if ( ed.name = 'CHANNEL_STOPPED' ) then do
      uvars = 'Q_MGR_NAME CHANNEL_NAME REASON_QUALIFIER ERROR_IDENTIFIER'
      do i=1 to words(uvars)
           uv = word(uvars,i)
           if ( wordpos(uv,ed.zlist) <> 0 ) then ,
               say uv '= <'ed.uv'>'
      end
end

/* So, if PN is not set within the Event  */
/* (it's an optional parameter), it will  */
/*    not be accessed.                     */
```

Figure 2. ZLIST and Event processing

Stem Variables

As described in “Chapter 5. Handling MQ Descriptors” on page 18, Stem variables are extensively used in this interface. A Stem variable is one that has various bits separated by dots (such as **a.b.c**). Everything after the first dot is called a component; so in the above example, ***a*** is the Stem variable, and ***b*** & ***c*** are components.

You should be aware that you can cause conflicts if you use Rexx variables with the same name as components. This is because Rexx will **substitute** the values of component names as if they were variables before usage.

```
a.1 = 15
a.2 = 3

b  = 2
say a.b     /* -> 3 due to substitution */
```

This can cause problems if you use any of the returned component names from this utility as native variables because you will get an 'unknown' setting due to the substitution.

```
qn = 'WMQA'
ud = 'some userish data'

rcc = RXMQ..(...data_which_will_set_.qn=A , 'out.')

say out.qn   /*    tries to resolve out.WMQA            */
             /*                     -> A                */
             /* as the utility does the substitution    */

say out.ud  /* tries to resolve out.some userish data    */
             /* -> a Rexx error due to invalid var name  */
```

Unless you are deliberately doing this sort of processing, I suggest you avoid using variables which are returned as components.

Rexx Execs

An Object Rexx exec in the Windows environment behaves differently from that in the z/OS environment due to the way the function is placed in regard to the operating system. One of these is in the way one runs an exec.

Under Windows a Rexx exec is run via

```
REXX exec.rexx
```

whereas under z/OS the exec is run via

```
exec
```

to make the latter style available under Windows, you can code up a one line batch file as follows:

```
@REXX exec.rexx %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Trace

Tracing is provided by settings in the **RXMQTRACE** Rexx variable. Note that the tracing is sent to the currently open STDOUT stream, and some of the settings can produce a lot of output. The settings are:

CONN *MQCONN*

DISC *MQDISC*

OPEN *MQOPEN*

CLOSE *MQCLOSE*

GET *MQGET*

PUT *MQPUT*

PUT1 *MQPUT1*

INQ *MQINQ*

SET *MQSET*

CMIT *MQCMIT*

BACK *MQBACK*

COM *MQSC*

BRO Browse extension

HXT Header extraction extension

EVENT Event expansion extension

TM Trigger message extension

MMD Rexx stem var -> **MQMD**

MOD Rexx stem var -> **MQOD**

MPO Rexx stem var -> **MQPMO**

MGO Rexx stem var -> **MQGMO**

BMD **MQMD** -> Rexx stem var

BOD **MQOD** -> Rexx stem var

BPO **MQPMO** -> Rexx stem var

BGO **MQGMO** -> Rexx stem var

SK Return Code processing

TR Thread based processing

INIT Initialization processing (including **RXMQCONS**)

TERM Deregistration processing

* Trace everything!!!

So, to trace Gets and Puts, one would code

```
RXMQTRACE = 'PUT GET'
```

Chapter 5. Handling MQ Descriptors

The API defined for MQ in the APR Manual uses various structures to pass information both into and out of the Queue Manager. These structures are:

MQOD The Object Descriptor, used by *MQOPEN* and *MQPUT1* verbs to specify the MQ Object being processed, and return various attributes of the accessed item

MQMD The Message Descriptor, used by *MQGET* and *MQPUT* verbs to specify (for the *MQPUT*) attributes for the emplaced message, and return these attributes (for the *MQGET*)

MQGMO This structure controls the operation of the *MQGET* verb

MQPMO This structure controls the operation of the *MQPUT* verb

MQCHARV This structure represents variable length strings used in **MQOD** structure

These structures are input/output for the MQ Verbs.

In order to supply these structures to the underlying MQ Verbs within this Rexx MQ Interface, Rexx **stem** stem variables are used. In order to reduce complexity, and enhance the ease of usage of the interface, **separate** Stem variables are used for input and output. This reduces the complexity of the Rexx code, as the input Stem variable may be reused without completely reinitializing it.

This approach allows, for simple applications, the initial setup of the stem variables representing the requested options; these are then repeatedly reused, the output versions simply not being accessed.

The **structure** of the stem variables is **fixed**. That means that the name of the stem variable (before the dot) can be chosen by the caller, whilst the latter part (after the dot) is fixed by the interface. The things after the dot are called the **Components** of the stem variable.

The normal Rexx rules apply to these Stem variables, in particular they are case invariant (Rexx treats all variables as being of Upper case), and substitution may occur within the name. Therefore, take care to avoid using variables that could clash with the naming conventions of these interface requirements (see “Stem Variables” on page 15).

When supplying these stem variables to the interface, you have to pass the **name** of the stem variable (including the trailing dot). Thus, one would normally specify this information as a literal (`RXMQ...( ..., 'AGMO.',...)`).

However, you are at liberty to use the normal Rexx substitutions on an interface call (so `Z = 'AGMO.'; RXMQ...(..., z )` is correct), and even abandon the stem variable convention completely (but this will lead to unwieldy execs). This abandonment, however, does not apply to one of the **RXMQOPEN** parameters.

When you build the stem variable, component abbreviations for the full name of the relevant structure's field is used (eg: **CCID** for *CodedCharSetId*) to improve legibility of the Exec. You only specify those fields of interest - the others should be **omitted**. The omitted components will default to the relevant settings as defined in the APR (usually a value or nulls).

However, although some fields of the descriptors are only used for input or output, this interface will utilize all of the information within the Stem variable - even if it is not used by the underlying MQ code (such as the Destination Count fields within the **PMO** descriptor - these are not used by the underlying MQ code, but this interface will process them if so supplied).

When the interface returns a structure to the exec, in the named Stem variable, **all** the components (fields) will be placed within the stem.structure.

The actual settings for these component variables are documented in the MQ APR to which you should refer. As the interface places within the Rexx workspace all `MQ_` numeric values, the stem components can be set using the normal MQ conventions (eg: `stem.PER = MQMD_NOT_PERSISTENT`). The interface does not check that the values are relevant for the field.

In the case of text fields, the interface will truncate supplied data that is too long for the MQ structure without notification. Fields that are to be null should not be supplied to the interface, and are returned as nulls (`''`).

Actual message data to/from the Queue Manager is passed via the usual Rexx convention (see “Message Lengths” on page 11 for a warning about truncation):

stem.0 contains the length of the data

stem.1 contains the message data

Functions **RXMQHXT** and **RXMQEVNT** will take messages and split them up into the contained components. These exploded components may clash with those for the Message Descriptor (or other like things). Therefore, use different stem. names to avoid this possibility.

ZLIST processing (see “ZLIST” on page 13) is available for the **Output** Stems representing a **MQOD**, **MQMD**, **MQGMO** or **MQPMO**. If present within an Input Stem. variable, **ZLIST** is ignored.

The Object Descriptor

The Object descriptor is used by **RXMQOPEN** and **RXMQPUT1** calls (*MQOPEN* and *MQPUT1* verbs) to specify the MQ Object being processed, and return various attributes of the accessed item.

If you are accessing a Queue, then the short cut form of **RXMQOPEN/RXMQPUT1** can be used, and so the Object Descriptor is only of interest upon completion of the call. The only interesting part of the **MQOD** in this case is the name of the 'real' queue generated when a Model queue is opened.

Table 2. Object Descriptor (MQOD) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
	Version 1		
Version	**.VER**	I	**MQLONG**
ObjectType	**.OT**	I	**MQLONG**
ObjectName	**.ON**	B	**MQCHAR48**
ObjectQMgrName	**.OQM**	B	**MQCHAR48**
DynamicQueueName	**.DQN**	I	**MQCHAR48**
AlternateUserID	**.AUID**	I	**MQCHAR12**
	Version 2		
RecsPresent	**.RP**	I	**MQLONG**
KnownDestCount	**.KDC**	O	**MQLONG**
UnknownDestCount	**.UDC**	O	**MQLONG**
InvalidDestCount	**.IDC**	O	**MQLONG**
	Version 3		
AlternateSecurityID	**.ASID**	I	**MQBYTE40**
ResolvedQName	**.RQN**	O	**MQCHAR48**
ResolvedQMgrName	**.RQMN**	O	**MQCHAR48**
	Version 4		
ObjectString	**.OS.**	I	**MQCHARV**
SelectionString	**.SS.**	I	**MQCHARV**
ResObjectString	**.ROS.**	O	**MQCHARV**
ResolvedType	**.RT**	O	**MQLONG**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names
- Components **RP** and **ASID** are supported only in Windows

The Message Descriptor

The Message Descriptor details the type of the message being processed. It also has a meaning where messages are obtained from a queue - whereat it is used to select messages for obtention from the queue. The interface does not check that combinations of components are valid.

As separate versions of a Message Descriptor are required by the interface for Input and Output on each call, the input **MQMD** can be reused for subsequent accesses. Components omitted will take the defaults as defined in the APR.

Table 3. Message Descriptor (MQMD) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
Version 1			
Version	**.VER**	I	**MQLONG**
Report	**.REP**	O/I	**MQLONG**
MsgType	**.MSG**	O/I	**MQLONG**
Expiry	**.EXP**	O/I	**MQLONG**
Feedback	**.FBK**	O/I	**MQLONG**
Encoding	**.ENC**	O/I	**MQLONG**
CodedCharSetId	**.CCSI**	O/I	**MQLONG**
Format	**.FORM**	O/I	**MQCHAR8**
Priority	**.PRI**	O/I	**MQLONG**
Persistence	**.PER**	O/I	**MQLONG**
MsgId	**.MSGID**	B/B	**MQBYTE24**
CorrelId	**.CID**	B/I	**MQBYTE24**
BackoutCount	**.BC**	B/-	**MQLONG**
ReplyToQ	**.RTOQ**	O/I	**MQCHAR48**
ReplyToQMgr	**.RTOQM**	O/I	**MQCHAR48**
UserIdentifier	**.UID**	O/B	**MQCHAR12**
AccountingToken	**.AT**	O/B	**MQBYTE32**
ApplyIdentityData	**.AID**	O/B	**MQCHAR32**
PutApplType	**.PAT**	O/B	**MQLONG**
PutApplName	**.PAN**	O/B	**MQCHR28**
PutDate	**.PD**	O/B	**MQCHAR8**
PutTime	**.PT**	O/B	**MQCHAR8**
ApplOriginData	**.AOD**	O/B	**MQCHAR4**
Version 2			
GroupId	**.GID**	B/B	**MQBYTE24**
MsgSeqNumber	**.MSN**	B/B	**MQLONG**

Structure name	Stem. Component	Input, Output or Both	Format
Offset	**.OFF**	B/B	**MQLONG**
MsgFlags	**.MF**	B/B	**MQLONG**
OriginalLength	**.OL**	O/B	**MQLONG**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names

The Get Message Option Structure

The Get Message Option Structure requests what message is to be obtained from a queue via the *MQGET* verb. As it is updated by this operation, **RXMQGET** uses an Input and Output Stem variable to hold this information.

Table 4. Get Message Option (MQGMO) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
Version 1			
Version	**.VER**	I	**MQLONG**
Options	**.OPT**	I	**MQLONG**
WaitInterval	**.WAIT**	I	**MQLONG**
ResolvedQueueName	**.RQN**	O	**MQCHAR48**
Version 2			
MatchOptions	**.MOPT**	I	**MQLONG**
GroupStatus	**.GS**	O	**MGCHAR**
SegmentStatus	**.SS**	O	**MGCHAR**
Segmentation	**.SEG**	O	**MGCHAR**
Version 3			
MsgToken	**.MT**	B	**MQBYTE16**
ReturnedLength	**.RL**	O	**MQLONG**
Version 4			
MsgHandle	**.MH**	I	**MQHMSG**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names

The Put Message Options Structure

The Put Message Option Structure requests what type of message is to be placed in a queue via the *MQPUT* or *MQPUT1* verb. As it is updated by this operation, **RXMQPUT** and **RXMQPUT1** uses an Input and Output Stem variable to hold this information.

Table 5. Put Message Options (MQPMO) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
Version 1			
Version	**.VER**	I	**MQLONG**
Options	**.OPT**	I	**MQLONG**
Timeout	**.TIME**	I	**MQLONG**
Context	**.CON**	I	**HOBJ**
KnownDestCount	**.KDC**	O	**MQLONG**
UnKnownDestCount	**.UDC**	O	**MQLONG**
InvalidDestCount	**.IDC**	O	**MQLONG**
ResolvedQueueName	**.RQN**	O	**MQCHAR48**
ResolvedQueueMgrName	**.RQMN**	O	**MQCHAR48**
Version 2			
RecsPresent	**.RP**	I	**MQLONG**
Version 3			
OriginalMsgHandle	**.OMH**	I	**MQHMSG**
NewMsgHandle	**.NMH**	B	**MQHMSG**
Action	**.ACT**	I	**MQLONG**
PubLevel	**.PL**	I	**MQLONG**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names
- Component **RP** is supported only in Windows.

The Variable Length String Structure

The Variable Length String Structure (**MQCHARV**) designates character string of variable length and specific character set identifier.

Actual string data to/from the Queue Manager is passed via the usual Rexx convention:

stem.0 contains the length of the string

stem.1 contains the string itself

stem. name is constructed of the name of the higher structure and the name of this structure component, like, for example, OD.ROS.1 stands for ReturnedObjectString in Object Descriptor structure. Before the call appropriate OD.ROS.0 variable should be set to the biggest of string length and buffer size. During the call VSBufSize is set to OD.ROS.0 and VSLength is set to actual string length. String buffer is automatically created and it's pointer is placed to **MQCHARV** VSPtr. On return from the call VSLength value is placed to OD.ROS.0 variable, returned string is placed to OD.ROS.1 and buffer is released.

Coded character set identifier is assigned to **stem.CCSI** variable.

Specifying parameters for command interface (RXMQC)

The interface requires some parameters to control operation:

- The name of the queue Manager (if omitted, the default Queue Manager is accessed)
- The name of the Command Input Queue (defaults to **SYSTEM.ADMIN.COMMAND.QUEUE**)
- The name of the model ReplyToQ (defaults to **SYSTEM.MQSC.REPLY.QUEUE**)
- A Timeout for the operation (defaults to 5 seconds)

In order to supply this information to the underlying MQ Verbs within this Rexx MQ Interface, Rexx **stem** variables are used.

The **structure** of the stem variables is **fixed**. That means that the name of the stem variable (before the dot) can be chosen by the caller, whilst the latter part (after the dot) is fixed by the interface. The things after the dot are called the **Components** of the stem variable.

The normal Rexx rules apply to these Stem variables, in particular they are case invariant (Rexx treats all variables as being of upper case), and substitution may occur within the name. Therefore, take care to avoid using variables that could clash with the naming conventions of these interface requirements.

When supplying these stem variables to the interface, you have to pass the **name** of the stem variable (including the trailing dot). Thus, one would normally specify this information as a literal (**`RXMQC( ..., 'ASTEM.',...)`**).

However, you are at liberty to use the normal Rexx substitutions on an interface call (so **`Z = 'ASTEM.'; RXMQC(..., z)`** is correct), and even abandon the stem variable convention completely (but this will lead to unwieldy execs). This abandonment, however, does not apply to one of the **RXMQC** parameters.

When you build the stem variable, component abbreviations for the full name of the relevant structure's field is used (eg: **CQ** for Command Queue) to improve legibility of the Exec. You only specify those fields of interest - the others should be **omitted**. The omitted components will default to the relevant settings as defined in the interface.

Actual message data in response to the Command is passed via the usual Rexx convention:

stem.0 contains the length of the data

stem.1 contains the data

Chapter 6. Thread Support

Some non-IBM Versions of Rexx which run in the Windows environment may support Rexx activity within threads. IBM Object Rexx for Windows does not support threads. There is no multitask support in z/OS REXX environment, so all the thread-based issues in this documentation will be applicable only for Windows Rexx/Threaded environment.

This utility supports access to the API in a Rexx/Threaded environment; however, this function is untested and usage is at your own risk.

Initialization

The initialization of via **RXMQINIT** should be done under the **Process** (main) thread. This will ensure that the functions are available for all the threads which are subsequently created within the process.

Within each thread, **RXMQINIT should not** be called (as this would remove all Global information stored within the interface). The thread should be initialized with a **RXMQCONS** call which simply creates the MQ mappings within the Rexx Variable space for the thread.

Termination

The usage of **RXMQTERM** will remove all access to the interface whomsoever is using it with the Process. If this is done before all threads within the Process have stopped accessing MQ facilities, this will result in errors (as the function will affect threads other than the issuing one).

No special processing needs to be done at thread termination, as MQ facilities will terminate any current access within the thread.

Connection and Disconnection

Within the thread, the **RXMQCONN** and **RXMQDISC** call should be made as normal to establish and terminate access to the Queue Manager. Each Thread can contact a different Queue Manager. The scope of the Connection is only within the issuing Thread. Access made in the original processes 'main' thread do **not** spread to the subsequent threads within the process.

When the **RXMQDISC** call is issued, all access to the Queue Manager within the issuing thread are effected. The graceful processing described in “Termination” on page 30 applies to all accesses within the current thread.

Access scope

The handle returned by the **RXMQCONN** and used for most of the access functions of the interface has a scope of the issuing thread. This is policed within the interface to prevent MQ errors.

Functions like **RXMQCMIT** and **RQMQBACK** have a scope of **all** accessed MQ objects **within** the issuing thread; they do not effect accesses outside the thread.

Shared Variables

The interface implicitly assumes that Rexx Variables within a thread **are not shared** across threads. If your version of Rexx supports shared Rexx variables across threads, then ensure that access is suitably restricted (ie: the variables used are unique to the threads) across these threads.

Chapter 7. The Interface

The functions provided by this Rexx MQ interface roughly follow those provided by the underlying MQ API, with some extensions and the calls required by Rexx to initialize the interface.

All the parameters specified for a call are required; none can be omitted.

When the interface detects an error, a **negative** return code will be provided as the first word in the return string. These are documented with the associated message under the individual calls.

Initialization

Description

This function initializes the interface, defines all the functions for Rexx usage, and places a number of the **MQ*** constants into the Rexx workspace. These constants are described in *WebSphere MQ Constants Version 6.0 SC34-6607*. See **MA95\src\RXMQCONS.hpp** for a list of the defined constants.

The **RXMQINIT** call needs to be done within each EXEC. In a thread-based environment, issue this call only in the 'owning' thread for the process.

Parameters

None

Call

```
rcc = RXMQINIT()
```

Additional Interface Return Codes and Messages

None

Example

```
rcc = RXMQINIT()
```

Thread Initialization

Description

This function places a number of the **MQ*** constants into the Rexx workspace for a thread. These constants are described in *WebSphere MQ Constants Version 6.0 SC34-6607*. See **MA95\src\RXMQCONS.hpp** for a list of the defined constants. This function can be called when there is no Queue Manager activity.

The **RXMQCONS** call needs to be the first call within a thread to setup the **MQ*** constants. It can be called generally to do the same function.

Parameters

None

Call

```
rcc = RXMQCONS()
```

Additional Interface Return Codes and Messages

None

Example

```
rcc = RXMQCONS()
```

Termination

Description

This function simply removes the access to the interface functions from Rexx. It does **not** initiate MQ Termination processing. If a prior **RXMQDISC** has not been done, then the usual End-of-Process MQ function will (eventually) stop access to the Queue Manager.

The **MQ*** definitions are left in the Rexx workspace, so that new commands can be composed using the 'real' notations.

As the **RXMQTERM** call removes the definitions of the functions, if two EXECs were running together, the second would start failing after the first issued the **RXMQTERM** call. Consequently, this call should be omitted under normal circumstances.

In a thread-based environment, issue this call **only** in the 'owning' thread for the process when all other accesses have ended.

Parameters

None

Call

```
rcc = RXMQTERM()
```

Additional Interface Return Codes and Messages

None

Example

```
rcc = RXMQTERM()
```

RXMQCONN

Description

This function connects the Rexx Interface to the Queue Manager. Note that there is a MQ restriction such that only one Queue Manager can be contacted from a Windows thread or z/OS task (the Rexx processor, in this case).

This call **has** to be made after the **RXMQINIT** call, and only be made once (unless a **RXMQDISC** is made).

Owing to the above restriction, the Queue Manager Handle returned by the use of *MQCONN* within the interface is not a useful thing, and so is not returned to the Rexx Exec.

Parameters

1. The name of the Queue Manager to connect to. (Input only).

Call

```
rcc = RXMQCONN( QM )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQCONN Bad number of parms
```

Explanation You must specify only **one** parameter to **RXMQCONN**; this parameter being the name of the Queue Manager to contact.

```
-2 0 0 RXMQCONN Null QM name
```

Explanation The Queue Manager name supplied contained only nulls, not a proper name.

```
-3 0 0 RXMQCONN Zero length QM name
```

Explanation The Queue Manager Name supplied was of zero length (ie: `''`).

```
-4 0 0 RXMQCONN QM name too long
```

Explanation The maximum length of a valid Queue Manager Name is **`MQ_Q_MGR_NAME_LENGTH`** bytes.

```
-5 0 0 RXMQCONN No available Q objects
```

Explanation The number of concurrent MQ Objects accessed with the process has reached its limit.

```
-95 0 0 RXMQCONN Mutex <n> Release failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

```
-96 0 0 RXMQCONN Mutex <n> Acquisition failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

```
-98 0 0 RXMQCONN Already Connected to a QM
```

Explanation The current thread has already Connected to a Queue Manager

```
-99 0 0 RXMQCONN UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQCONN(   'WMQA'   )
```

This call will contact the local Queue Manager called **WMQA**. If this Queue Manager is not defined, or not running, then the call will fail.

If the **RXMQCONN** Client/Server interface is being used, then the MQ Client/Server Communications must be configured and active for the Server Queue Manager to be contacted. A Client/Server *MQCONN* takes longer to complete than a local contact.

RXMQDISC

Description

This function disconnects (*MQDISC*) from the currently connected Queue Manager. As an extension to the function, the interface will issue a *MQCLOSE(…,MQCO_NONE)* for any still open queue accessed via the interface (this is to cope with Rexx Tracing, and so give the user a simple way of 'gracefully' exiting when in test mode).

Parameters

None.

Call

```
rcc = RXMQDISC()
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQDISC Bad number of parms
```

Explanation You cannot specify any parameters to this call.

```
-95 0 0 RXMQDISC Mutex <n> Release failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

```
-96 0 0 RXMQDISC Mutex <n> Acquisition failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

```
-98 0 0 RXMQDISC Not Connected to a QM
```

Explanation The current thread was not connected to a Queue Manager

```
-99 0 0 RXMQDISC UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQDISC()
```

This call will disconnect from the currently accessed Queue Manager doing a *MQCLOSE(None)* on any Queues still open at this point.

If the **RXMQDISC** Client/Server interface is being used, then the MQ Client/Server Communications will stop after this call has successfully completed.

RXMQOPEN

Description

This verb provides access to a MQ Object via a *MQOPEN* call. Upto 100 Objects can be accessed via this interface in any one process (ie: 100 spread throughout all the threads owned by the process). Although one will normally be accessing a Queue, any of the allowed MQ objects can be accessed.

Parameters

1. The name of a Stem variable (including the dot) specifying the Object Descriptor for the MQ Object to access. This is an input only field. The format of this Stem variable is described in “The Object Descriptor” on page 20.
   If the name given does not end in a dot, then the data is taken to be the name of a Queue (or Model Queue) to access. This short cut removes the requirement to fully format up a stem variable for 'normal' Queue access; but note that you supply the name of the Queue, not the name of the variable containing the name of the Queue.
2. The *MQOPEN* Options (as described in the APR). This is an input only field, and should resolve into a number (not the name of a field containing the Options).
3. The name of a variable to contain a handle for the MQ Object being accessed. This is an output field, and should be the name of the field to receive the handle.
   The handle returned is **not** the handle returned by the underlying *MQOPEN* verb; this latter value is not accessible outside of the interface. This handle must be quoted on all subsequent accesses to the Object.
4. The name of a Stem variable (including the dot) into which is placed the Object Descriptor returned by the underlying *MQOPEN* verb. This is an output only field.
   The format of this Stem variable is described in “The Object Descriptor” on page 20; **ZLIST** processing is provided.

Call

```
rcc = RXMQOPEN( 'Stem.Input.OD.', OpenOptions, 'VarHandle', 'Stem.Output.OD.' )
```

or

```
rcc = RXMQOPEN( QueueName       , OpenOptions, 'VarHandle', 'Stem.Output.OD.' )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQOPEN Bad number of parms
```

Explanation You **must** specify four parameters to the **RXMQOPEN** call.

```
-2 0 0 RXMQOPEN Null Input OD/Qname
```

Explanation A null has been supplied for the first parameter, the name of a stem variable for an input Open Descriptor or the name of a Queue to access.

```
-3 0 0 RXMQOPEN Zero length Input OD/Qname
```

Explanation No value has been keyed for the first parameter, the name of a stem variable for an input Open Descriptor or the name of a Queue to access.

```
-4 0 0 RXMQOPEN Null options
```

Explanation A null has been supplied for the second parameter, a number representing the Open Options. To specify No Options, supply a 0.

-5 0 0 RXMQOPEN Zero length options

Explanation No value has been keyed for the second parameter, a number representing the Open Options. To specify No Options, supply a 0.

-6 0 0 RXMQOPEN Null handle name

Explanation A null has been supplied for the third parameter, the name of a variable which will be set to the obtained handle for the accessed MQ Object.

-7 0 0 RXMQOPEN Zero length handle name

Explanation No value has been keyed for the third parameter, the name of a variable which will be set to the obtained handle for the accessed MQ Object.

-8 0 0 RXMQOPEN Null Output OD

Explanation A null has been supplied for the forth parameter, the name of a stem variable which will be set to the obtained Object Descriptor for the accessed MQ Object.

-9 0 0 RXMQOPEN Zero length Output OD

Explanation No value has been keyed for the forth parameter, the name of a stem variable which will be set to the obtained Object Descriptor for the accessed MQ Object.

-10 0 0 RXMQOPEN No available Q objects

Explanation The limit of MQ Objects supported by this interface has been reached.

-95 0 0 RXMQOPEN Mutex <n> Release failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

-96 0 0 RXMQOPEN Mutex <n> Acquisition failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

-98 0 0 RXMQOPEN Not Connected to a QM

Explanation The current thread was not connected to a Queue Manager

-99 0 0 RXMQOPEN UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Example

```
opts =  MQOO_INQUIRE        + MQOO_INPUT_SHARED        ,
      + MQOO_BROWSE         + MQOO_SAVE_ALL_CONTEXT    ,
      + MQOO_FAIL_IF_QUIESCING

rcc = RXMQOPEN( N1, opts, 'hn1', 'od.' )
```

This call opens the Queue **N1** for a Browse access, and permits the inquiry of the queue's attributes. If the open succeeds, then the variable **hn1** is set to the handle for subsequent access to **N1**, and the stem variable **od.** is set to the contents of the Object Descriptor for **N1** (eg: **od.ON = 'N1'**).

```
iod.OT = MQOT_Q
iod.ON = 'N1'

rcc = RXMQOPEN( 'iod.', MQOO_BROWSE+MQOO_INQUIRE, 'hn1', 'ood.' )
```

This example shows how the Queue **N1** would be accessed if the full Object Descriptor method is used to specify the MQ Object to be accessed.

RXMQCLOS

Description

This verb stops access to a MQ Object, using the underlying *MQCLOSE* verb.

Parameters

1. The Handle for the object obtained from a prior **RXMQOPEN** call. This is an input parameter. After this call completes, the handle is no longer valid for use.
2. The Close options. This is an input parameter representing the type of *MQCLOSE* operation to be performed.

Call

```
rcc = RXMQCLOS( handle, CloseOptions )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQCLOS Bad number of parms
```

Explanation You **must** specify two parameters to the **RXMQCLOS** call.

```
-2 0 0 RXMQCLOS Null handle
```

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

```
-3 0 0 RXMQCLOS Zero length handle
```

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

```
-4 0 0 RXMQCLOS Null options
```

Explanation A null has been supplied for the second parameter, a number representing the Close Options. To specify No Options, supply a 0.

```
-5 0 0 RXMQCLOS Zero length options
```

Explanation No value has been keyed for the second parameter, a number representing the Close Options. To specify No Options, supply a 0.

```
-6 0 0 RXMQCLOS Handle out of range
```

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

```
-7 0 0 RXMQCLOS Invalid handle
```

Explanation The handle specified does not relate to an accessed MQ Object.

```
-95 0 0 RXMQCLOS Mutex <n> Release failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

```
-96 0 0 RXMQCLOS Mutex <n> Acquisition failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

```
-97 0 0 RXMQCLOS Handle not owned by Current Thread
```

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

```
-98 0 0 RXMQCLOS Not Connected to a QM
```

Explanation The current thread was not connected to a Queue Manager

```
-99 0 0 RXMQCLOS UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQCLOSE( hn1, MQCO_NONE )
```

This call closes the object referred to by the handle specified in the `hn1` variable, with no special closing actions being requested.

RXMQINQ

Description

This call will inquire upon a **single** attribute of a MQ object. This is a difference between this interface and the function of the underlying *MQINQ* verb.

The relevant data is returned in character format, so numeric attributes need not be converted for Rexx usage. The requested attribute is specified via **MQIA_, MQCA_...** variables.

Parameters

1. The **handle** for the object obtained from a prior **RXMQOPEN** call, whereat the object was opened for Inquiry. This is an input parameter.
2. The **Attribute** number to be inquired upon (setting starting with **MQIA, MQCA_...)**. This is an input parameter.
3. The name of a variable into which will be returned the current setting of the desired attribute. Numeric attributes (like Maximum Message Size) are presented as decimal strings (so '17' might be returned rather than '11'x). This is an output parameter.

Call

```
rcc = RXMQINQ ( handle, Attribute, VarAttributeValue )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQINQ Bad number of parms
```

Explanation You **must** specify three parameters to the **RXMQINQ** call.

```
-2 0 0 RXMQINQ Null handle
```

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

```
-3 0 0 RXMQINQ Zero data handle
```

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

```
-4 0 0 RXMQINQ Null data input attr
```

Explanation A null has been supplied for the second parameter, a number representing representing the attribute of the MQ object to be obtained.

```
-5 0 0 RXMQINQ Zero data input attr
```

Explanation No value has been keyed for the second parameter, a number representing the attribute of the MQ object to be obtained.

```
-6 0 0 RXMQINQ Null output attr
```

Explanation A null has been supplied for the third parameter, the name of a variable to receive the value of the requested attribute.

```
-7 0 0 RXMQINQ Zero length output attr
```

Explanation No value has been keyed for the third parameter, the name of a variable to receive the value of the requested attribute.

```
-8 0 0 RXMQINQ No attribute supplied
```

Explanation No value was supplied for the attribute under consideration.

```
-9 0 0 RXMQINQ Attribute out of valid range
```

Explanation The value of the attribute under consideration was outside of the ranges defined for Integer and Character attributes.

```
-10 0 0 RXMQINQ Handle out of range
```

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

```
-11 0 0 RXMQINQ Invalid handle
```

Explanation The handle specified does not relate to an accessed MQ Object.

```
-95 0 0 RXMQINQ Mutex <n> Release failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

```
-96 0 0 RXMQINQ Mutex <n> Acquisition failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

```
-97 0 0 RXMQINQ Handle not owned by Current Thread
```

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was RXMQOPENed by another thread).

```
-98 0 0 RXMQINQ Not Connected to a QM
```

Explanation The current thread was not connected to a Queue Manager

```
-99 0 0 RXMQINQ UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQINQ( hn1, MQIA_MAX_MSG_LENGTH,   'maxmsg'   )
      /* maxmsg = 3109856 */
```

This call obtains the current Maximum Message Length attribute for the queue referenced by the handle contained in **hn1**. In this case, the **maxmsg** variable is set to 3109856, the value of the desired attribute.

RXMQSET

Description

This call will set a **given** attribute of a MQ object. This is a difference between this interface and the underlying *MQSET* verb, whereat many attributes can be manipulated in a single execution.

The relevant data is specified in character format, so numeric attributes need not be converted for interface usage. The attribute is specified via **MQIA_, MQCA_...** variables.

Parameters

1. The **handle** for the object obtained from a prior **RXMQOPEN** call, whereat the object was opened for Setting. This is an input parameter.
2. The **Attribute** Number to be set (starting with **MQIA_, MQCA_...)**. This is an input parameter.
3. The value of the attribute which is to be set in the MQ Object. Numeric attributes (like Trigger Depth) are specified as a normal Rexx decimal string (so use '17' rather than '11'x). This is an input parameter.

Call

```
rcc = RXMQSET( handle, Attribute, AttributeSetting )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQSET Bad number of parms
```

Explanation You **must** specify three parameters to the **RXMQSET** call.

```
-2 0 0 RXMQSET Null handle
```

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

```
-3 0 0 RXMQSET Zero data handle
```

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

```
-4 0 0 RXMQSET Null data attribute
```

Explanation A null has been supplied for the second parameter, a number representing representing the attribute of the MQ object to be set.

```
-5 0 0 RXMQSET Zero data attribute
```

Explanation No value has been keyed for the second parameter, a number representing the attribute of the MQ object to be set.

```
-6 0 0 RXMQSET Null setting
```

Explanation A null has been supplied for the third parameter, the name of a variable to receive the value of the requested attribute.

```
-7 0 0 RXMQSET Zero length setting
```

Explanation No value has been keyed for the third parameter, the name of a variable to receive the value of the requested attribute.

```
-8 0 0 RXMQSET No attribute supplied
```

Explanation No value was supplied for the attribute under consideration.

```
-9 0 0 RXMQSET Attribute out of valid range
```

Explanation The value of the attribute under consideration was outside of the ranges defined for Integer and Character attributes.

```
-10 0 0 RXMQSET Handle out of range
```

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

```
-11 0 0 RXMQSET Invalid handle
```

Explanation The handle specified does not relate to an accessed MQ Object.

```
-95 0 0 RXMQSET Mutex <n> Release failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

```
-96 0 0 RXMQSET Mutex <n> Acquisition failed rc <n>
```

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

```
-97 0 0 RXMQSET Handle not owned by Current Thread
```

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

```
-98 0 0 RXMQSET Not Connected to a QM
```

Explanation The current thread was not connected to a Queue Manager

```
-99 0 0 RXMQSET UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQSET( hn1, MQIA_TRIGGER_DEPTH,  21)
```

This call sets the Trigger Depth for the Queue specified by **hn1** (which must have been opened with Set access) to 21 messages.

RXMQCMIT

Description

This verb will issue a *MQCMIT* verb. It syncpoints the current Queue Manager accesses. Note that this operation affects **all** the currently accessed queues which have extant operations within Unit of Work control **within** the current thread (ie: it does not effect other threads within the process).

Parameters

None

Call

```
rcc = RXMQCMIT()
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQCMIT Bad number of parms
```

Explanation You cannot specify any parameters to this call.

```
-98 0 0 RXMQCMIT Not Connected to a QM
```

Explanation The current thread is not connected to a Queue Manager

```
-99 0 0 RXMQCMIT UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQCMIT()
```

The accesses to all currently accessed queues (that are within Unit of Work control) are committed. Accesses outside of UOW control are unaffected by this call.

RXMQBACK

Description

This verb will issue a *MQBACK* verb. It rolls back the current Queue Manager accesses. Note that this operation affects **all** the currently accessed queues which have extant operations within Unit of Work control **within** the current thread (ie: it does not effect other threads within the process).

Parameters

None

Call

```
rcc = RXMQBACK()
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQBACK Bad number of parms
```

Explanation You cannot specify any parameters to this call.

```
-98 0 0 RXMQBACK Not Connected to a QM
```

Explanation The current thread is not connected to a Queue Manager

```
-99 0 0 RXMQBACK UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
rcc = RXMQBACK()
```

The accesses to all currently accessed queues (that are within Unit of Work control) are rolledback. Accesses outside of UOW control are unaffected by this call.

RXMQGET

Description

This call will obtain a message from a Queue, using the underlying *MQGET* verb. All the abilities of this verb are supported by this interface.

A quick way of issuing Browse calls is provided by “RXMQBRWS” on page 57.

Parameters

1. The Handle for the Queue obtained from a prior **RXMQOPEN** call, whereat the Queue was opened for Input (or Browse) access. This is an Input parameter.
2. The name of a Rexx Stem variable (including the dot) into which the obtained message will be placed. This is an input/output parameter. Upon the call, Component 0 must contain the Maximum length of the message to be received. After the call, Component 0 will contain the length of the message received (or would have been received if the initial setting was 0) and Component 1 will contain the obtained message (if any). See “Message Lengths” on page 11 for a warning about truncation.
3. The name of a Stem variable (including the dot) containing the Input Message Descriptor describing the Message to be obtained from the Queue. This is an input parameter.
4. The name of a Stem variable (including the dot) into which will be returned a Message Descriptor describing the message obtained by the call. This is an output parameter, so **ZLIST** processing is provided.
5. The name of a Stem variable (including the dot) containing the Get Message Options for the operation. This is an input parameter.
6. The name of a Stem variable (including the dot) into which will be placed the updated Get Message Options resulting from the call. This is an output parameter, so **ZLIST** processing is provided.

Call

```
rcc = RXMQGET( handle, 'Stem.Message.' , 'Stem.Input.MD.' , 'Stem.Output.MD. ' ,
                                         'Stem.Input.GMO.', 'Stem.Output.GMO.' )
```

Additional Interface Return Codes and Messages

-1 0 0 RXMQGET Bad number of parms

Explanation You must specify six parameters to the **RXMQGET** call.

-2 0 0 RXMQGET Null handle

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

-3 0 0 RXMQGET Zero data handle

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

-4 0 0 RXMQGET Null data stem var

Explanation A null has been supplied for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

-5 0 0 RXMQGET Zero data stem var

Explanation No value has been keyed for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

-6 0 0 RXMQGET Null Input MsgDesc

Explanation A null has been supplied for the third parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

-7 0 0 RXMQGET Zero length Input MsgDesc

Explanation No value has been keyed for the third parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

-8 0 0 RXMQGET Null Output MsgDesc

Explanation A null has been supplied for the forth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

-9 0 0 RXMQGET Zero length Output MsgDesc

Explanation No value has been keyed for the forth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

-10 0 0 RXMNGET Null input GMO

Explanation A null has been supplied for the fifth parameter, the name of a Stem Variable containing the Get Message Options for the operation.

-11 0 0 RXMNGET Zero length input GMO

Explanation No value has been keyed for the fifth parameter, the name of a Stem Variable containing the Get Message Options for the operation.

-12 0 0 RXMNGET Null output GMO

Explanation A null has been supplied for the sixth parameter, the name of a Stem Variable into which will be placed the resulting Get Message Options from the operation.

-13 0 0 RXMQGET Zero length output GMO

Explanation No value has been keyed for the sixth parameter, the name of a Stem Variable into which will be placed the resulting Get Message Options from the operation.

-14 0 0 RXMQGET Handle out of range

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

-15 0 0 RXMQGET Invalid handle

Explanation The handle specified does not relate to an accessed MQ Object.

-16 0 0 RXMQGET malloc failure, RC(<errno>)

Explanation An attempt to acquire storage for the number of bytes specified in the Message.0 (2nd parameter) variable failed. The return code is errno code from the malloc call.

-17 0 0 RXMQGET Zero length input data buffer

Explanation The Message.0 (2nd parameter) was zero, indicating no message to process.

-97 0 0 RXMQGET Handle not owned by Current Thread

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

-98 0 0 RXMQGET Not Connected to a QM

Explanation The current thread was not connected to a Queue Manager

-99 0 0 RXMQGET UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Example

```
message.0 = 100
message.1 = ''

igmo.opt = MQGMO_WAIT + MQGMO_SYNCPOINT + MQGMO_FAIL_IF_QUIESCING
igmo.wait = 1

imd.MSGID = ''
imd.CID  = ''

rcc = RXMQGET( hn1, 'message.', 'imd.', 'omd.', 'igmo.', 'ogmo.' )

/* on return, say…

message.0 = 13
message.1 = 'WMQ rules OK!'

omd.msg  = MQMT_DATAGRAM
omd.PER  = MQPER_PERSISTENT
…
ogmo.rqn = 'N1'

*/
```

This call destructively obtains the next message from the Queue. The message can be upto 100 bytes long - a bigger message is not obtained (as the options do not specify **MQGMO_ACCEPT_TRUNCATED_MSG**). The obtained message (which will not physically be removed from the Queue until a Syncpoint is issued, as it is obtained under Unit Of Work control) is 13 bytes long, and is persistent.

RXMQPUT

Description

This call will place a message into a Queue, using the underlying *MQPUT* verb. All the abilities of this verb are supported by this interface.

Parameters

1. The Handle for the Queue obtained from a prior **RXMQOPEN** call, whereat the Queue was opened for Output access. This is an Input parameter.
2. The name of a Rexx Stem variable (including the dot) containing the message to be placed on the Queue. This is an input parameter. Component 0 must contain the length of Component 1, which is the message to be put into the Queue.
3. The name of a Stem variable (including the dot) containing the Input Message Descriptor describing the Message to be placed on the Queue. This is an input parameter.
4. The name of a Stem variable (including the dot) into which will be returned a Message Descriptor describing the message placed by the call. This is an output parameter, so **ZLIST** processing is provided.
5. The name of a Stem variable (including the dot) containing the Put Message Options for the operation. This is an input parameter.
6. The name of a Stem variable (including the dot) into which will be placed the updated Put Message Options resulting from the call. This is an output parameter, so **ZLIST** processing is provided.

Call

```
rcc = RXMQPUT( handle, 'Stem.Message.' , 'Stem.Input.MD.' , 'Stem.Output.MD. ' ,
                                         'Stem.Input.PMO.', 'Stem.Output.PMO.' )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQPUT Bad number of parms
```

Explanation You **must** specify six parameters to the **RXMQPUT** call.

```
-2 0 0 RXMQPUT Null handle
```

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

```
-3 0 0 RXMQPUT Zero data handle
```

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

```
-4 0 0 RXMQPUT Null data stem var
```

Explanation A null has been supplied for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

```
-5 0 0 RXMQPUT Zero data stem var
```

Explanation No value has been keyed for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

`-6 0 0 RXMQPUT Null Input MsgDesc`

Explanation A null has been supplied for the third parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

`-7 0 0 RXMQPUT Zero length Input MsgDesc`

Explanation No value has been keyed for the third parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

`-8 0 0 RXMQPUT Null Output MsgDesc`

Explanation A null has been supplied for the forth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

`-9 0 0 RXMQPUT Zero length Output MsgDesc`

Explanation No value has been keyed for the forth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

`-10 0 0 RXMQPUT Null input PMO`

Explanation A null has been supplied for the fifth parameter, the name of a Stem Variable containing the Put Message Options for the operation.

`-11 0 0 RXMQPUT Zero length input PMO`

Explanation No value has been keyed for the fifth parameter, the name of a Stem Variable containing the Put Message Options for the operation.

`-12 0 0 RXMQPUT Null output PMO`

Explanation A null has been supplied for the sixth parameter, the name of a Stem Variable into which will be placed the resulting Put Message Options from the operation.

`-13 0 0 RXMQPUT Zero length output PMO`

Explanation No value has been keyed for the sixth parameter, the name of a Stem Variable into which will be placed the resulting Put Message Options from the operation.

`-14 0 0 RXMQPUT Handle out of range`

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

`-15 0 0 RXMQPUT Invalid handle`

Explanation The handle specified does not relate to an accessed MQ Object.

`-16 0 0 RXMQPUT malloc failure, RC(<errno>)`

Explanation An attempt to acquire storage for the number of bytes specified in the data.0 (2nd parameter) variable failed. The return code is errno code from the malloc call.

`-17 0 0 RXMQPUT Zero length input data buffer`

Explanation The data.0 (2nd parameter), the Stem variable containing the length of message to be sent, was zero or not numeric.

`-18 0 0 RXMQPUT Data length is not equal to specified value`

Explanation The data.0 (2nd parameter), the Stem variable containing the length of message to be sent, is not equal to data.1 actual message length.

`-19 0 0 RXMQPUT Context handle out of range`

Explanation The value of the handle supplied in **`PMO.CON`** is not in the known range for a handle within

the interface.

```
-20 0 0 RXMQPUT Invalid Context handle
```

Explanation The handle specified in **PMO.CON** does not relate to an accessed MQ Object.

```
-96 0 0 RXMQPUT Context handle not owned by current thread
```

Explanation The object referred to by the given **PMO.CON** handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

```
-97 0 0 RXMQPUT Handle not owned by current thread
```

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

```
-98 0 0 RXMQPUT Not Connected to a QM
```

Explanation The current thread was not connected to a Queue Manager

```
-99 0 0 RXMQPUT UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
message.1 = 'WMQ''s wonderful interface!'
message.0 = LENGTH(message.1)

ipmo.opt    =      MQPMO_NO_SYNCPOINT     + MQPMO_NO_CONTEXT  ,
                 + MQPMO_FAIL_IF_QUIESCING

imd.MSG     = MQMT_DATAGRAM
imd.per     = MQPER_NOT_PERSISTENT

rcc = RXMQPUT( hn1,  'message.',  'imd.',  'omd.',  'ipmo.',  'opmo.' )

/* on return, say……

omd.PD   = 20080831
…

opmo.rqn = 'N1'

*/
```

This call places the given non-persistent message on the Queue outside of a Unit of Work.

RXMQPUT1

Description

This call will open the Queue, place one message into it and close the queue using the underlying *MQPUT1* verb. All the abilities of this verb are supported by this interface.

Parameters

1. The name of a Stem variable (including the dot) specifying the Object Descriptor for the MQ Object to access. This is an input only field.

   The format of this Stem variable is described in "The Object Descriptor" on page 20.

   If the name given does not end in a dot, then the data is taken to be the name of a Queue (or Model Queue) to access. This short cut removes the requirement to fully format up a stem variable for 'normal' Queue access; but note that you supply the name of the Queue, not the name of the variable containing the name of the Queue.

2. The name of a Stem variable (including the dot) into which is placed the Object Descriptor returned by the underlying *MQOPEN* verb. This is an output only field.

   The format of this Stem variable is described in "The Object Descriptor" on page 20; **ZLIST** processing is provided.

3. The name of a Rexx Stem variable (including the dot) containing the message to be placed on the Queue. This is an input parameter. Component 0 must contain the length of Component 1, which is the message to be put into the Queue.
4. The name of a Stem variable (including the dot) containing the Input Message Descriptor describing the Message to be placed on the Queue. This is an input parameter.
5. The name of a Stem variable (including the dot) into which will be returned a Message Descriptor describing the message placed by the call. This is an output parameter, so **ZLIST** processing is provided.
6. The name of a Stem variable (including the dot) containing the Put Message Options for the operation. This is an input parameter.
7. The name of a Stem variable (including the dot) into which will be placed the updated Put Message Options resulting from the call. This is an output parameter, so **ZLIST** processing is provided.

Call

```
rcc = RXMQPUT1( 'Stem.Input.OD.', 'Stem.Output.OD.', 'Stem.Message.',
                'Stem.Input.MD.', 'Stem.Output.MD.', 'Stem.Input.PMO.',
                'Stem.Output.PMO.' )
```

or

```
rcc = RXMQPUT1( Queue Name,       'Stem.Output.OD.', 'Stem.Message',
                'Stem.Input.MD.', 'Stem.Output.MD.', 'Stem.Input.PMO',
                'Stem.Output.PMO' )
```

Additional Interface Return Codes and Messages

-1 0 0 RXMQPUT1 Bad number of parms

Explanation You must specify seven parameters to the **RXMQPUT1** call.

-2 0 0 RXMQPUT1 Null Input OD/Qname

Explanation A null has been supplied for the first parameter, the name of a stem variable for an input Object Descriptor or the name of a Queue to access.

-3 0 0 RXMQPUT1 Zero length Input OD/Qname

Explanation No value has been keyed for the first parameter, the name of a stem variable for an input Object Descriptor or the name of a Queue to access.

-4 0 0 RXMQPUT1 Null Output OD

Explanation A null has been supplied for the second parameter, the name of a stem variable which will be set to the obtained Object Descriptor for the accessed MQ Object.

-5 0 0 RXMQPUT1 Zero length Output OD

Explanation No value has been keyed for the second parameter, the name of a stem variable which will be set to the obtained Object Descriptor for the accessed MQ Object.

-6 0 0 RXMQPUT1 Null data stem var

Explanation A null has been supplied for the third parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

-7 0 0 RXMQPUT1 Zero data stem var

Explanation No value has been keyed for the third parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

-8 0 0 RXMQPUT1 Null Input MsgDesc

Explanation A null has been supplied for the fourth parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

-9 0 0 RXMQPUT1 Zero length Input MsgDesc

Explanation No value has been keyed for the fourth parameter, the name of a Stem Variable containing the Input Message Variable for the operation.

-10 0 0 RXMQPUT1 Null Output MsgDesc

Explanation A null has been supplied for the fifth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

-11 0 0 RXMQPUT1 Zero length Output MsgDesc

Explanation No value has been keyed for the fifth parameter, the name of a Stem Variable into which will be placed the resulting Message Descriptor from the operation.

-12 0 0 RXMQPUT1 Null input PMO

Explanation A null has been supplied for the sixth parameter, the name of a Stem Variable containing the Put Message Options for the operation.

-13 0 0 RXMQPUT1 Zero length input PMO

Explanation No value has been keyed for the sixth parameter, the name of a Stem Variable containing

the Put Message Options for the operation.

-14 0 0 RXMQPUT1 Null output PMO

Explanation A null has been supplied for the seventh parameter, the name of a Stem Variable into which will be placed the resulting Put Message Options from the operation.

-15 0 0 RXMQPUT1 Zero length output PMO

Explanation No value has been keyed for the seventh parameter, the name of a Stem Variable into which will be placed the resulting Put Message Options from the operation.

-16 0 0 RXMQPUT1 No available Q objects

Explanation The limit of MQ Objects supported by this interface has been reached.

-17 0 0 RXMQPUT1 malloc failure, RC(<errno>)

Explanation An attempt to acquire storage for the number of bytes specified in the data.0 (3rd parameter) variable failed. The return code is errno code from the malloc call.

-18 0 0 RXMQPUT1 Zero length input data buffer

Explanation The data.0 (3rd parameter), the Stem Variable containing the length of message to be sent, was zero or not numeric.

-19 0 0 RXMQPUT1 Data length is not equal to specified value

Explanation The data.0 (3rd parameter), the Stem variable containing the length of message to be sent, is not equal to data.1 actual message length.

-20 0 0 RXMQPUT1 Context handle out of range

Explanation The value of the handle supplied in **PMO.CON** is not in the known range for a handle within the interface.

-21 0 0 RXMQPUT1 Invalid Context handle

Explanation The handle specified in **PMO.CON** does not relate to an accessed MQ Object.

-95 0 0 RXMQPUT1 Mutex <n> Release failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

-96 0 0 RXMQPUT1 Mutex <n> Acquisition failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

-97 0 0 RXMQPUT1 Context handle not owned by current thread

Explanation The object referred to by the given **PMO.CON** handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

-98 0 0 RXMQPUT1 Not Connected to a QM

Explanation The current thread was not connected to a Queue Manager

-99 0 0 RXMQPUT1 UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Example

```
message.1 = 'WMQ''s wonderful interface!'
message.0 = LENGTH(message.1)

ipmo.opt    =      MQPMO_NO_SYNCPOINT     + MQPMO_NO_CONTEXT  ,
                 + MQPMO_FAIL_IF_QUIESCING

imd.MSG     = MQMT_DATAGRAM
imd.per     = MQPER_NOT_PERSISTENT

rcc = RXMQPUT1( ‘iod.’, ‘ood.’,  'message.',  'imd.',  'omd.',  'ipmo.',  'opmo.' )

/* on return, say……

omd.PD   = 20080831
…

opmo.rqn = 'N1'

*/
```

This call places the given non-persistent message on the Queue outside of a Unit of Work.

```
message.1 = 'WMQ''s wonderful interface!'
message.0 = LENGTH(message.1)

ipmo.opt    =      MQPMO_NO_SYNCPOINT     + MQPMO_NO_CONTEXT  ,
                 + MQPMO_FAIL_IF_QUIESCING

imd.MSG      = MQMT_DATAGRAM
imd.per      = MQPER_NOT_PERSISTENT

rcc = RXMQPUT1( N1, ‘ood.’,  'message.',  'imd.',  'omd.',  'ipmo.',  'opmo.'  )

/* on return, say…

omd.PD   = 20080831
…

opmo.rqn = 'N1'

*/
```

This call places the given non-persistent message on the Queue outside of a Unit of Work using Queue Name as a parameter in **RXMQPUT1** function

RXMQC

Description

This function issues a **MQSC** command to a Queue Manager, returning the results. Each invocation checks whether the connection to Queue Manager exists. If the connection exists, the function uses it. If the connection does not exist, the function creates a connection to Queue Manager. If the connection is created by this function, it is released after the call.

As **RXMQC** operates within the Rexx environment, all the Rexx variables used are available for use within ISPF in the normal fashion. See Appendix A. ISPF Interface on page 75 for an example.

Parameters

1. The controlling parameters (Input only)

   either The name of the Queue Manager to connect to

   or The name of a Stem. variable containing the name of the Queue Manager (`.QM`), the name of Command Queue (`.CQ`), the name of Model ReplyToQ (`.RQ`) and the Timeout (`.TO`)

2. The command to issue
3. The name of a Stem. variable into which the results of the `command` will be placed

Call

```
rcc = RXMQC (qmname,        'command', 'Stem.Reply.' )
```

or

```
rcc = RXMQC ('Stem.Parms.', 'command', 'Stem.Reply.' )
```

Table 6. RXMQC Function Parameter Mappings

Name	Stem. Component	Input, Output or Both	Format
Queue Manager	`.QM`	I	`MQCHAR48`
Command Queue	`.CQ`	I	`MQCHAR48`
Model ReplyToQ	`.RQ`	I	`MQCHAR48`
Timeout	`.TO`	I	`MQLONG`

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- A timeout of 0 is an eternal wait

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQC Bad number of parms
```

Explanation You **must** specify three parameters to **RXMQC**

-2 0 0 RXMQC Null parms

Explanation A null has been supplied for the first parameter, the name of a stem variable which will be set to the controlling parms for the operation.

-3 0 0 RXMQC Zero parms

Explanation No value has been keyed for the first parameter, the name of a stem variable which will be set to the controlling parms for the operation.

-4 0 0 RXMQC Null command var

Explanation A null has been supplied for the second parameter, the text of the command to issue.

-5 0 0 RXMQC Zero command var

Explanation No value has been keyed for the second parameter, the text of the command to issue.

-6 0 0 RXMQC Null response stem var

Explanation A null has been supplied for the third parameter, the name of a stem variable which will contain the results of the command.

-7 0 0 RXMQC Zero response stem var

Explanation No value has been keyed for the third parameter, the name of a stem variable which will contain the results of the command.

-8 0 0 RXMQC No command supplied

Explanation The length of the second parameter is zero, so no command was given.

-9 0 0 RXMQC Too big a command supplied

Explanation The length of the second parameter is bigger than 5000 bytes, which is the maximum supported Command length.

-10 0 0 RXMQC malloc failure, RC(<errno>)

Explanation An attempt to acquire storage for the response buffer. The return code is errno code from the malloc call.

-99 0 0 RXMQC UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Examples

```
res.0 = 0
rcc = RXMQC('WMQA',   'DISPLAY QUEUE(*)  TYPE(QLOCAL)',   'res.'   )
```

This call will contact the local Queue Manager called **WMQA** and ask for a list of all the local Queues. The returned information is placed in the **res.n** stem variables, with **res.0** indicating the number of elements.

```
parm.QM = 'WMQA'
parm.TO = 0
com   = 'ping channel(CWTONBYTN)'
res.0  = 0
rcc   = RXMQC('parm.', com, 'res.' )
```

This call will contact the remote Queue Manager called **WMQA** and issue a ping on the given channel. The interface will wait until a reply to the ping is received.

RXMQBRWS

Description

This call is an extension to the MQ API as documented in the APR. This call will obtain the next message from a Queue via a **Browse** operation, using the underlying Browse function of the *MQGET* verb.

As this call is designed to be simple way to browse messages on a Queue, no Get Message Options or Message Descriptors are available. If access to these is required, then use the base "RXMQGET" on page 45.

Similarly, the position of the Browse cursor cannot be manipulated.

Parameters

1. The Handle for the Queue obtained from a prior **RXMQOPEN** call, whereat the Queue was opened for Browse access. This is an Input parameter.
2. The name of a Rexx Stem variable (including the dot) into which the obtained message will be placed. This is an input/output parameter. Upon the call, Component **0** must contain the Maximum length of the message to be received. After the call, Component **0** will contain the length of the message received (or would have been received if the initial setting was 0) and Component **1** will contain the obtained message (if any). See "Message Lengths" on page 11 for a warning about truncation.

Call

```
rcc = RXMQBRWS( handle, 'Stem.Message.' )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQBRWS Bad number of parms
```

Explanation You **must** specify two parameters to the **RXMQBRWS** call.

```
-2 0 0 RXMQBRWS Null handle
```

Explanation A null has been supplied for the first parameter, the handle representing the MQ object.

```
-3 0 0 RXMQBRWS Zero data handle
```

Explanation No value has been keyed for the first parameter, the handle representing the MQ object.

```
-4 0 0 RXMQBRWS Null data stem var
```

Explanation A null has been supplied for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

```
-5 0 0 RXMQBRWS Zero data stem var
```

Explanation No value has been keyed for the second parameter, the name of a Stem Variable containing the maximum length of message to be obtained.

```
-6 0 0 RXMQBRWS Handle out of range
```

Explanation The value of the handle supplied is not in the known range for a handle within the interface.

```
-7 0 0 RXMQBRWS Invalid handle
```

Explanation The handle specified does not relate to an accessed MQ Object.

-8 0 0 RXMQBRWS malloc failure, RC(<errno>)

Explanation An attempt to acquire Storage for the number of bytes specified in the Message.0 (2nd parameter) variable failed. The return code is errno code that for the malloc call, and will usually result if the message.0 value is not numeric.

-9 0 0 RXMQBRWS Zero length input data buffer

Explanation The Message.0 (2nd parameter) was zero, indicating no message to process

-95 0 0 RXMQBRWS Mutex <n> Release failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to release the lock for the given Reason (documented under CSingleLock::UnLock).

-96 0 0 RXMQBRWS Mutex <n> Acquisition failed rc <n>

Explanation The anonymous private Mutex used internally to single-thread object-based logic has failed to acquire for the given Reason (documented under CSingleLock::Lock).

-97 0 0 RXMQBRWS Handle not owned by Current Thread

Explanation The object referred to by the given handle was not accessed by the current Thread (ie: it was **RXMQOPEN**ed by another thread).

-98 0 0 RXMQBRWS Not Connected to a QM

Explanation The current thread was not connected to a Queue Manager .

-99 0 0 RXMQBRWS UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Example

```
message.0 = 100
message.1 =  ''

rcc = RXMQBRWS  ( hn1,   'message.')

/* on return, say… message.0 = 2 ; message.1 =  'M1'    */

message.0 = 100
message.1 =  ''

rcc = RXMQBRWS  ( hn1,   'message.')

/* on return, say… message.0 = 8 ; message. 1 =  '>>>M2<<<'    */
```

This example shows how a Browse is used to scan a Queue; observe that the message. Stem variable is cleared before each use.

RXMQHXT

Description

This call will take a message obtained from a Transmission Queue or a Dead Letter Queue (identified by the relevant header in the message) and split it up into its components.

This Header Extraction, therefore, permits the obtention of the 'real' message and an explanation of the control data associated with it.

The message to be split up is specified in the usual way as the name of the first stem. variable; with component **0** representing the length of the message which is supplied in component **1**. See “Message Lengths” on page 11 for a warning about truncated messages used with this function.

The Extracted data is placed in the second stem. variable (whose name is supplied); with component **0** representing the length of the 'actual' message which is placed in component **1**. The associated data is placed in other components, as shown in Table 7. Transmission Queue Message Header (MQXQH) Mappings on page 61 and in the Table 8. Dead Letter Queue Message Header (MQDLH) Mappings on page 62. It is **not** recommended that the input and output stem variables are the same (as this might loose information in the case of an error and additionally the component names clash with those generated as part of the Message descriptor).

In order to identify the type of header extracted, a component called **TYPE** is also created, taking the value of **XQH** or **DLH** (this is also provided in the **NAME** component).

Parameters

1. The name of a Rexx Stem variable (including the dot) containing a message to be splitup. This is an input parameter. Upon the call, Component **0** must contain the length of the message in Component **1**; the message must have been obtained from a Transmission Queue or a Dead Letter Queue. See “Message Lengths” on page 11 for a warning about truncation.
2. The name of a Rexx Stem variable (including the dot) into which the splitup message will be placed. This is an input/output parameter. After the call, Component 0 will contain the length of the 'actual' message and Component **1** will contain the 'actual' message (if any). Other components will be created (as documented in the Table 7. Transmission Queue Message Header (MQXQH) Mappings on page 61 and in theTable 8. Dead Letter Queue Message Header (MQDLH) Mappings on page 62) to return the extracted Header information from the input message. **ZLIST** processing is provided for this Stem variable.

Call

```
rcc = RXMQHXT( 'Stem.Message.', 'Stem.Splitup.' )
```

Additional Interface Return Codes and Messages

```
-1 0 0 RXMQHXT Bad number of parms
```

Explanation You **must** specify two parameters to the **RXMQHXT** call.

```
-2 0 0 RXMQHXT Null input stem var
```

Explanation A null has been supplied for the first parameter, the name of a Stem. variable representing the message to be splitup.

```
-3 0 0 RXMQHXT Zero input stem var
```

Explanation No value has been keyed for the first parameter, the name of a Stem. variable representing the message to be splitup.

-4 0 0 RXMQHXT Null output stem var

Explanation A null has been supplied for the second parameter, the name of a Stem. variable representing the splitup message.

-5 0 0 RXMQHXT Zero output stem var

Explanation No value has been keyed for the second parameter, the name of a Stem. variable representing the splitup message.

-6 0 0 RXMQHXT No input data

Explanation The input Stem.0 was zero, indicating no message to process

-7 0 0 RXMQHXT Zero input data

Explanation The length of the input Stem.1 was zero, indicating no message to process

-8 0 0 RXMQHXT Cannot locate Header

Explanation The input Stem.0 was <= 3, indicating no header in the message

-9 0 0 RXMQHXT Cannot find Header

Explanation The length of the input Stem.1 was <= 3, indicating no header in the message

-10 0 0 RXMNHXT Unknown Header

Explanation The first 4 bytes of the input Stem.1 was not **DLH** or **XQH**, so the message did not come from a Dead Letter Queue or a Transmission Queue, and so cannot be splitup

-11 0 0 RXMNHXT Too short for a DLH (<n> bytes!)

Explanation Although the input Stem.1 looked like a DLH, Stem.0 was too small for the message to originate from a Dead Letter Queue, and so cannot be splitup

-12 0 0 RXMNHXT Too short for a XQH (<n> bytes!)

Explanation Although the input Stem.1 looked like a XQH, Stem.0 was too small for the message to originate from a Transmission Queue, and so cannot be splitup

-13 0 0 RXMHXT malloc failure, RC(<errno>)

Explanation An attempt to acquire Storage for the number of bytes specified in the input Stem.0 (1st parameter) variable failed. The return code is errno code from the malloc call.

-14 0 0 RXMQPUT1 Data length is not equal to specified value

Explanation The Stem.0 (1st parameter), the Stem variable containing the length of message to be splitup, is not equal to Stem.1 actual message length.

-99 0 0 RXMQHXT UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Extracted information

Transmission Queue Messages

Table 7. Transmission Queue Message Header (MQXQH) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
Actual message length	`.0`	O	`MQLONG`
Actual message	`.1`	O	`MQCHAR`
RemoteQName	`.RQN`	O	`MQCHAR48`
RemoteQMgrName	`.RQM`	O	`MQCHAR48`
MsgDesc	`.XXX`	O	`MQMD1`

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- `ZLIST` is set to relevant existing field Stem. Component names
- See MsgDesc component names in the Table 3. Message Descriptor (MQMD) Mappings

Dead Letter Queue Messages

Table 8. Dead Letter Queue Message Header (MQDLH) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
Actual message length	**.0**	O	**MQLONG**
Actual message	**.1**	O	**MQCHAR**
Reason	**.REA**	O	**MQLONG**
DestinationQMgrName	**.DQM**	O	**MQCHAR48**
DestinationQName	**.DQN**	O	**MQCHAR48**
Encoding	**.ENC**	O	**MQLONG**
CodedCharSetId	**.CCSI**	O	**MQLONG**
Format	**.FORM**	O	**MQCHAR8**
PutApplType	**.PAT**	O	**MQLONG**
PutApplName	**.PAN**	O	**MQCHAR28**
PutDate	**.PD**	O	**MQCHAR8**
PutTime	**.PT**	O	**MQCHAR8**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names

Example

```
/* A message has been obtained such that ... */

message.0 = 438

message.1 = <XQH>1234567890

/* Clear the result variable */

drop x.

/* Split the message */

rcc = RXMQHXT ( 'message.', 'x.' )

/* on return, the following (and more) are set */

say x.0      /* 10          */
say x.1      /* 1234567890  */
say x.RQM    /* WMQA        */
say x.RQN    /* CP1         */
say x.PER    /* 1           */
say x.TYPE   /* XQH         */
```

This example shows how a message obtained from a Transmission Queue is splitup, showing information extracted from the **XQH** and the actual message being transmitted.

RXMQEVNT

Description

This call will take a message obtained from an Event Queue and split it up into its components. In general the default system queues called are
SYSTEM.ADMIN.QMGR.EVENT,
SYSTEM.ADMIN.CONFIG.EVENT,
SYSTEM.ADMIN.PERFM.EVENT and
SYSTEM.ADMIN.CHANNEL.EVENT

This Event Extraction, therefore, permits the detection of the event and an explanation of the control data associated with it.

The message to be split up is specified in the usual way as the name of a stem. variable; with component **0** representing the length of the message which is supplied in component **1**. See "Message Lengths" on page 11 for a warning about truncated messages used with this function. This message will have come from a prior **RXMQBRWS** or **RXMQGET** operation.

The Extracted data is placed in another stem. variable (whose name is supplied), with the various components containing information about the event. Each component name provided is equal to event attribute constant name, which are described in *WebSphere MQ Monitoring WebSphere MQ Version 6.0 SC34-6593*. For ease of reading and to save space constant name prefix (like **MQCA**_ or **MQIA**_) is omitted. It is **not** recommended that the input and output stem variables are the same (as this might loose information in the case of an error and additionally the component names clash with those generated as part of the Message descriptor). Observe that some information is held in the event message's Message Descriptor (like Date and Time), so obtaining the message should be done via a Browse-type of **RXMQGET** rather than the **RXMQBRWS** call which does not return the Message Descriptor if this type of information is required.

In order to identify the type of event extracted, a component called **TYPE** is created and set to **EVENT**, and another called **NAME** which interprets the Event (it is presented in the same way as described above for the component names).

Information about Events is discussed in *WebSphere MQ Monitoring WebSphere MQ Version 6.0 SC34-6593* book which you should use to interpret the expansion.

Warning

The PCF Documentation on events sometimes does not agree with what is actually recorded in the Event Message. Please take care in this arena, and treat deviations from the Documentation pragmatically (ie: raise an APAR, but process as this interface returns). A general usage should test each component to discover whether or not this information is returned. Alternatively, use **ZLIST** processing (as described in "ZLIST" on page 13). A returned component may be null (or have a zero length) if the Event Field is present without any data.

Parameters

1. The name of a Rexx Stem variable (including the dot) containing an event message to be splitup. This is an input parameter. Upon the call, Component **0** must contain the length of the message in Component **1**; the message must have been obtained from an Event Queue. See "Message Lengths" on page 11 for a warning about truncation.
2. The name of a Rexx Stem variable (including the dot) into which the splitup message will be placed. This is an input/output parameter. After the call, components will be created (as described above) to return the extracted event information from the input message. **ZLIST** processing is provided for this Stem variable.

Call

```
rcc = RXMQEVNT( 'Stem.Message.', 'Stem.Splitup.' )
```

Usage Notes

Bear in mind the following when using **RXMQEVNT**:

- A component is returned when the relevant parameter is present in the PCF Event Message. The returned data may consist of binary zeros, a null string (") or all spaces if the contents do not exist (this is due to the way MQ builds the PCF Event message). Certain Rexx processors object to long strings of Binary zeros, so *you have been warned!*

- The PCF Event documentation may differ from the data actually returned. Always use **ZLIST** processing to see what is going on!

- The **EID, AEDI1, AEDI2 and CED** fields are not returned as numbers, but rather in Hex. This will aid problem determination for these Channel error codes.

- There may be more than one **CED** field. In this case, **.CED.0** will contain the number of fields, with the data being in **.CED.n**

- The Date and Time of an Event is **not** held within the event, but in the Message Descriptor for the event.

- **.TYPE** is set to '**EVENT**' for all events.

Additional Interface Return Codes and Messages

-1 0 0 RXMQEVNT Bad number of parms

Explanation You **must** specify two parameters to the **RXMQEVNT** call.

-2 0 0 RXMQEVNT Null input stem var

Explanation A null has been supplied for the first parameter, the name of a Stem. variable representing the message to be splitup.

-3 0 0 RXMQEVNT Zero input stem var

Explanation No value has been keyed for the first parameter, the name of a Stem. variable representing the message to be splitup.

-4 0 0 RXMQEVNT Null output stem var

Explanation A null has been supplied for the second parameter, the name of a Stem. variable representing the splitup message.

-5 0 0 RXMQEVNT Zero output stem var

Explanation No value has been keyed for the second parameter, the name of a Stem. variable representing the splitup message.

-6 0 0 RXMQEVNT No input data

Explanation The input Stem.0 was zero, indicating no message to process

-7 0 0 RXMQEVNT Zero input data

Explanation The length of the input Stem.1 was zero, indicating no message to process

-8 0 0 RXMQEVNT Cannot locate Header

Explanation The input Stem.0 was <= 3, indicating no header in the message

-9 0 0 RXMQEVNT Cannot find Header

Explanation The length of the input Stem.1 was <= 3, indicating no header in the message

-10 0 0 RXMQEVNT Not an Event Header

Explanation The first 4 bytes of the input Stem.1 was not <MQCFH_EVENT>, so the message did not come from an Event Queue, and so cannot be splitup

-11 0 0 RXMQEVNT Too short for an Event (<n> bytes!)

Explanation Although the input Stem.1 looked like an Event Message, Stem.0 was too small for the message to originate from an Event Queue, and so cannot be splitup

-12 0 0 RXMQEVNT Unknown Event Category (<n>)

Explanation Although the input Stem.1 looked like an Event Message, the PCF Command field did not contain a recognisable event category, and so the message cannot be splitup

-13 0 0 RXMQEVNT Unknown Event Type (<n>)

Explanation Although the input Stem.1 looked like an Event Message, the PCF Reason field did not contain a recognisable event identifier, and so the message cannot be splitup

```
-14 0 0 RXMQEVNT No elements in the Event
```

Explanation Although the input Stem.1 looked like an Event Message, there were no PCF fields within the Message, and so the message cannot be splitup

```
-15 0 0 RXMQEVNT malloc failure, RC(<errno>)
```

Explanation An attempt to acquire Storage for the number of bytes specified in the input Stem.0 (1st parameter) variable failed. The return code is errno code that for the malloc call, and will usually result if the input Stem.0 value is not numeric or negative integer.

```
-16 0 0 RXMQPUT1 Data length is not equal to specified value
```

Explanation The Stem.0 (1st parameter), the Stem variable containing the length of message to be splitup, is not equal to Stem.1 actual message length.

```
-99 0 0 RXMQEVNT UNKNOWN FAILURE
```

Explanation Some unknown error has occurred!

Example

```
/* A message has been obtained such that ... */

message.0 = n

message.1 = <EVENT Header><Event Data>

/* Clear the result variable */

drop x.

/* Split the message */

rcc = RXMQEVNT ( 'message.', 'x.' )

/* on return, the following (and more) are set */

say x.TYPE                /*     EVENT                */
say x.NAME                /*     CHANNEL_STOPPED      */
say x.REA                 /*     2283                 */
say x.Q_MGR_NAME          /*     WMQA                 */
say x.CHANNEL_NAME        /*     SYSTEM.ADMIN.SVRCONN */
say x.REASON_QUALIFIER    /*     8                    */
say x.ERROR_IDENTIFIER    /*     20009519             */
```

This example shows how a message obtained from **SYSTEM.ADMIN.CHANNEL.EVENT** is splitup, showing the information relating to the Channel Stop Event.

See Figure 2 on page 14 for an example using **ZLIST** processing to cope with the variable format component names.

```
/* Explurge an Event */

message.0 = n
message.1 = <EVENT Header><Event Data>
drop x.
rcc = RXMQEVNT ( 'message.', 'x.' )
```

```
/* Testing the returned information */

say x.TYPE   /* EVENT */
say x.NAME   /* INGET */
say x.REA    /* 2016  */

if ( x.qn <> 'X.QN' ) then say x.qn  /* works - returned comp */
if ( x.BQN <> 'X.BQN' ) then say x.bqn /* fails - not in event */
```

This example shows how the components of an exploded Event message can be tested to fully extract all the returned information if **ZLIST** processing is not used.

ZLIST processing is also useful to cope with situations were an event String Field is defined, but set to all binary zeros. These can easily be changed into blanks (with space truncation) as follows:

```
message.0 = n
message.1 = <EVENT Header><Event Data>
drop x.
rcc = RXMQEVNT ( 'message.', 'x.' )

do i=1 to words(x.zlist)
   ts = word(x.zlist,i)
   x.ts = translate(x.ts,' ','00'x)
   x.ts = strip(x.ts,'B')
end
```

RXMQTM

Description

This call will take a message obtained from an Initiation Queue (a Trigger Message) and split it up into its components. It will also parse the data passed to a started Rexx Exec (via a MQ Trigger Monitor).

This processing, therefore, permits the obtention of the control information associated with a Trigger: whether this is in the format of a MQ Message (garnered from an Initiation Queue) or passed as parameters to a Rexx Exec (as the Triggered Process).

The action of this function is controlled by the format of its first parameter, in particular whether or not it ends in a **dot**.

- If it **ends** in a dot, then **RXMQTM** is processing a message derived from an Initiation Queue.

  The message to be processed is specified in the usual way as the name of a stem. variable; with component **0** representing the length of the message which is supplied in component **1**. See "Message Lengths" on page 11 for a warning about truncated messages used with this function.

  This is called Message Mode.

- If it does **not end** in a dot, then **RXMQTM** is processing the parameter data passed via a Trigger Monitor to the Rexx Exec which is acting as a Triggered Process (ie: replaces the initial *parse arg* processing). It is the actual data, **not** a variable name that is supplied (ie: a substituted variable, not the variable name).

  This is called Data Mode.

The Extracted data is placed in another stem. variable (whose name is supplied); with components representing the various sub-fields of the Trigger Message or Trigger parms.

Sub-fields which are all blanks (or start with a Binary Zero) are not extracted. **ZLIST** processing (see "ZLIST" on page 13) is provided so that the various extant components can be determined.

In Message Mode (a Trigger Message provided to **RXMQTM** in a Stem. variable) an additional component (not in **ZLIST**) called **PL** is provided which is the Parameter list for a process to be invoked by the reception of the Trigger Message in the Initiation Queue (if the current thread is connected to a Queue Manager, its name will be present in **.PL**). You should ensure that this component is not truncated in any way (as this will may well effect the activity of the process which uses it).

You can, therefore, use a Rexx Exec as the Triggered Process, extracting the supplied information using **RXMQTM** in Data Mode.

The use of Message Mode permits the coding of your own Trigger Monitor (recall the Trigger Messages only get placed in an Initiation Queue if the priorities are right, the process exists, and the Initiation Queue is Open for Getting) in Rexx (see Figure 3 on page 73), and Data Mode permits the use of Rexx Execs as Triggered Processes (see Figure 4 on page 74).

Parameters

1. This parameter takes one of these formats:

In Message Mode	The name of a Rexx Stem variable (including the dot) containing a message to be splitup. This is an input parameter. Upon the call, Component **0** must contain the length of the message in Component **1**; the message must have been obtained from an Initiation Queue. See "Message Lengths" on page 11 for a warning about truncation.
In Data Mode	The actual data (not a variable name) representing the **MQTMC2** structure which is used to initiate a Triggered Process.

2. The name of a Rexx Stem variable (including the dot) into which the extracted data will be placed. This is an input/output parameter. After the call, components will be created (as documented in Table 9. Trigger Component (MQTM/MQTMC2) Mappings on page 72) to return the extracted information. **ZLIST** processing is provided for this Stem variable. In the case of Message Mode, component **PL** will contain an area suitable for use by a Triggered Process as its parameters.

Call

```
Message Mode:

   rcc = RXMQTM( 'Stem.Message.', 'Stem.Splitup.' )

Data  Mode:

   rcc = RXMQTM( MQTMC2_data    , 'Stem.Splitup.' )
```

Additional Interface Return Codes and Messages

-1 0 0 RXMQTM Bad number of parms

Explanation You **must** specify two parameters to the **RXMQTM** call.

-2 0 0 RXMQTM Null input stem var

Explanation A null has been supplied for the first parameter, the name of a Stem. variable representing the message to be splitup or data representing a **MQTMC2** structure to be parsed.

-3 0 0 RXMQTM Zero input stem var

Explanation No value has been keyed for the first parameter, the name of a Stem. variable representing the message to be splitup or data representing a **MQTMC2** structure to be parsed.

-4 0 0 RXMQTM Null output stem var

Explanation A null has been supplied for the second parameter, the name of a Stem. variable representing the splitup data.

-5 0 0 RXMQTM Zero output stem var

Explanation No value has been keyed for the second parameter, the name of a Stem. variable representing the splitup data.

-6 0 0 RXMQTM No input data

Explanation The input Stem.0 was zero, indicating no message to process

-7 0 0 RXMQTM Zero input data

Explanation The length of the input Stem.1 was zero, indicating no message to process

-8 0 0 RXMQTM Cannot locate Header

Explanation The input Stem.0 was <= 3, indicating no header in the message

-9 0 0 RXMQTM Cannot find Header

Explanation The length of the input Stem.1 was <= 3, indicating no header in the message

-10 0 0 RXMQTM Zero input data

Explanation The length of the input data was zero, indicating no **MQTMC2** structure to process

-11 0 0 RXMQTM Cannot find Header

Explanation The length of the input data was <= 3, indicating no header in the alleged **MQTMC2** structure

-12 0 0 RXMQTM Unknown Header

Explanation The first 4 bytes of the input Stem. 1 or data was not **'TM '**, so the message did not come from an Initiation Queue or a Triggered Process' parameter, and so cannot be splitup

-13 0 0 RXMQTM Unknown Version

Explanation The second 4 bytes of the input Stem.1 or data was not that for a recognised Version field, so the message did not come from an Initiation Queue or a Triggered Process' parameter, and so cannot be splitup

-14 0 0 RXMQTM Header mismatch (1<>1)

Explanation Message mode was selected, but the StrucId field was not correct

-15 0 0 RXMQTM Version mismatch (1<>1)

Explanation Message mode was selected, but the Version field was not correct

-16 0 0 RXMQTM Header mismatch (2<>C)

Explanation Data mode was selected, but the StrucId field was not correct

-17 0 0 RXMQTM Version mismatch (2<>C)

Explanation Data mode was selected, but the Version field was not correct

-18 0 0 RXMQTM Too short for a TM (<n> bytes!)

Explanation Although the input Stem.1 looked like a Trigger Message, Stem.0 was too small for the message to originate from an Initiation Queue, and so cannot be splitup

-19 0 0 RXMQTM Too short for a TMC (<n> bytes!)

Explanation Although the input data looked like Trigger Parms (MQTMC2), the data was too small (not 732 bytes long) for it to be a Triggered Process' Parameter and so cannot be splitup

-20 0 0 RXMQTM malloc failure, RC(<errno>)

Explanation An attempt to acquire Storage for the number of bytes specified in the input Stem.0 (1st parameter) variable failed. The return code is errno code from the malloc call.

-21 0 0 RXMQPUT1 Data length is not equal to specified value

Explanation The Stem.0 (1st parameter), the Stem variable containing the length of message to be splitup, is not equal to Stem.1 actual message length.

-99 0 0 RXMQTM UNKNOWN FAILURE

Explanation Some unknown error has occurred!

Trigger information

Table 9. Trigger Component (MQTM/MQTMC2) Mappings

Structure name	Stem. Component	Input, Output or Both	Format
QName	**.QN**	O	**MQCHAR48**
ProcessName	**.PN**	O	**MQCHAR48**
TriggerData	**.TD**	O	**MQCHAR64**
ApplType	**.AT**	O	**MQLONG**
ApplId	**.AID**	O	**MQCHAR256**
EnvData	**.ED**	O	**MQCHAR128**
UserData	**.UD**	O	**MQCHAR128**
QMgrName	**.QM**	O	**MQCHAR48**
MQTMC2 parameter	**.PL**	O	**MQTMC2**

Notes:

- Input, Output and Both show how the field is used
- Format shows the type of the field
- **ZLIST** is set to relevant existing field Stem. Component names
- Text items which are all Blanks (or start with a Binary Zero) are not generated
- **.AT** and **.PL** are only available in Message Mode
- **.QM** is only available in Data Mode
- **.PL** is not placed in **ZLIST**

Examples

```
/* A message has been obtained from an Initiation Queue */

message.0 = 684
message.1 = <MQTM>

/* Clear the result variable */

drop t.

/* Split the message */

rcc = RXMQTM( 'message.', 't.' )

/* on return, the following are set */

say t.QN        /* L3N1     */
say t.PN        /* P3TO46N   */

/* Truncated non-parm areas for usage */

do j=1 to words(t.zlist)
   item   = word(t.zlist,j)
   t.item = strip(t.item,'B')
end

/* Some processing to decide on something to do */

/* Start a Process to service the Queue         */

'@START CMD someproc.CMD' t.pl exit
```

Figure 3. A Trigger Monitor

This example shows how a message obtained from an Initiation Queue is splitup, showing how the PL component is used to start a process to service the Queue which generated the Trigger. Note that all the parameters passed in the Message can be used however one wants when one codes ones own Trigger Monitor.

```
/* Get the parm */

parse arg parm

/* Clear the result variable */

drop p.

/* Split the parm */

rcc = RXMQTM( parm,   'p.'  )

/* on return, the following are set */

say p.QM     /* WMQA        */
say p.QN     /* L3N1        */
say p.PN     /* P3TO46N     */
```

```
/* Truncate areas for usage */

do j=1 to words(p.zlist)
   item       = word(p.zlist,j)
   p.item    = strip(p.item,'B')
   end
```

Figure 4. A Rexx Triggered Process

This example shows how a Rexx Exec being initiated via a Trigger Monitor accesses its passed data.

Appendix A. ISPF Interface

As **RXMQC** operates within the Rexx environment, all the Rexx variables used are available for use within ISPF in the normal fashion. Figure 5. ISPF Exec (MA95T1) on page 76 shows an ISPF exec using **RXMQC** to issue a Command via the panel shown in Figure 6. ISPF Panel (MA95P1) on page 76. If everything works, then Figure 7. ISPF Panel (success) on page 77 is displayed, or else something like Figure 8. ISPF Panel (failure) on page 77.

```
/* REXX ***************************************************************  */
"ISPQRY"                          /* Check ISPF availability          */
lastrc = rc
if lastrc = 20 then do            /* rc=20 is no ISPF,so start ISPF   */
    address TSO 'ISPSTART CMD(MA95T1) NEWAPPL(MA95)'
    exit
  end

address ispexec

qm          = '' ; com    = '' ; rcm    = '' ; rcc    = '' ; rcc1 = ''
reply1      = '' ; reply2 = '' ; reply3 = '' ; reply4 = '' ; rcc2 = ''

"VPUT (qm com rcm rcc rcc1 rcc2 reply1 reply2 reply3 reply4 ) SHARED"

Do mainloop = 1 by 1              /* Do mainloop forever              */

  "DISPLAY PANEL("MA95P1")" ; lastrc = rc

  if ( lastrc <> 0 ) then leave mainloop

  "VGET (qm com rcm rcc rcc1 rcc2 reply1 reply2 reply3 reply4) SHARED"

                                rcm = '' ; rcc          = '' ; rcc1 = ''
  reply1 = ''; reply2 = '' ; reply3 = '' ; reply4       = '' ; rcc2 = ''


drop rep.

rcci = RXMQINIT
rcc = RXMQC(qm, com, 'rep.' )
rcct = RXMQTERM

rcm = strip(word(rcc,1),'B')
rcm = rxmq.rcmap.rcm
rcc1 = rep.cc
rcc2 = rep.ac

if ( (rep.0 <> 'REP.0') & (rep.0 <> 0) ) then do
     if (rep.0 >= 1) then reply1 = rep.1
     if (rep.0 >= 2) then reply2 = rep.2
     if (rep.0 >= 3) then reply3 = rep.3
     if (rep.0 >= 4) then reply4 = rep.4
     end
end
```

```
exit 0
/********** End of MA95T1 ****************************************/
```

Figure 5. ISPF Exec (MA95T1)

```
)ATTR
/*********************************************************************/
      # TYPE(INPUT) COLOR(WHITE)
      @ TYPE(OUTPUT) COLOR(TURQ)
      { TYPE(OUTPUT) CAPS(OFF)
)BODY SMSG(MSG)
%------------------- MA95 Rexx/MQ/MVS/ -----------------------
%COMMAND ==> #ZCMD                                          +

                            +MA95P1 Commands+

 +QM+#qm                                                     +
 +Command+#com                                               +
 +Reply1:+{reply1


 +Reply2:+{reply2


 +Reply3:+{reply3


 +Reply4:+{reply4


 +Com.cc {rcc1           +Com.rc {rcc2                   +
 +rc     {rcm
 +rcc    {rcc

 {msg
 )INIT
 /*****************************************/
 /* INITIALIZATION SECTION */
 /*****************************************/
 )PROC
 /*****************************************/
 /* PROCESSING SECTION */
 /*****************************************/
 )END
```

Figure 6. ISPF Panel (MA95P1)

```
--------------------------------- MA95 Rexx/MQ/MVS/ -----------------------------
                                  MA95P1 Commands
QM WMQA
Command DISPLAY QUEUE(N1) ALL
Reply1: CSQM401I  QUEUE(N1) TYPE(LOCAL)
DESCR(Notpersist)
PUT(ENABLED  ) DEFPRTY(       0) DEFPSIST(NO      ) OPPROCS(
Reply2: CSQ9022I > CSQMDRTS ' DISPLAY QUEUE' NORMAL COMPLETION


Reply3:


Reply4:

Com.cc       00000000       Com.rc 00000000
Rc           MQCC_OK
rcc          0 0 0 RXMQC OK
```

Figure 7. ISPF Panel (success)

```
--------------------------------- MA95 Rexx/MQ/MVS/ -----------------------------
                                  MA95P1 Commands
QM VRH1
Command DISPLAY QZ  (N1) ALL
Reply1: CSQ9021E > VERB ' DISPLAY' REQUIRED KEYWORD IS MISSING


Reply2: CSQ9023E > CSQ9SCND 'DISPLAY ' ABNORMAL COMPLETION


Reply3:


Reply4:


Com.cc       00000008       Com.rc FFFFFFFF
Rc           MQCC_OK
rcc          0 0 0 RXMQC OK
```

Figure 8. ISPF Panel (failure)

Appendix B. Sample REXX execs

MA95\samples directory contains a number of useful examples related to both z/OS and Windows environments. They are generally self-explaining and show how to exploit the functions provided by SupportPac. In order to run them you can use sample jobs in **MA95\MVS\JCL** or batch files in **MA95\Windows\Runtests**. Before running these samples please update Queue Manager and Queue names to comply with your WebSphere MQ installation names.

- **RXMQCMND.REXX** – sends a command to Queue Manager, receives a response and presents it to the REXX caller.
- **RXMQEVNT.REXX** – illustrates EVENT queue interface. Requests data from one of EVENT queues, interprets it by calling **RXMQEVNT**, and presents the results.
- **RXMQHXT.REXX** – illustrates transmission queue message header parsing. Gets a message from transmission queue, interprets is by calling **RXMQHXT**, and presents the results.
- **RXMQMA95.REXX** – perform basic WebSphere MQ functions, showing how to process the results and return/reason codes.
- **RXMQTM.REXX** – shows how to use Trigger Monitor function of SupportPac. Gets message from Initiation Queue, calls **RXMQTM** to interpret the message, and presents the results.