WebSphere software

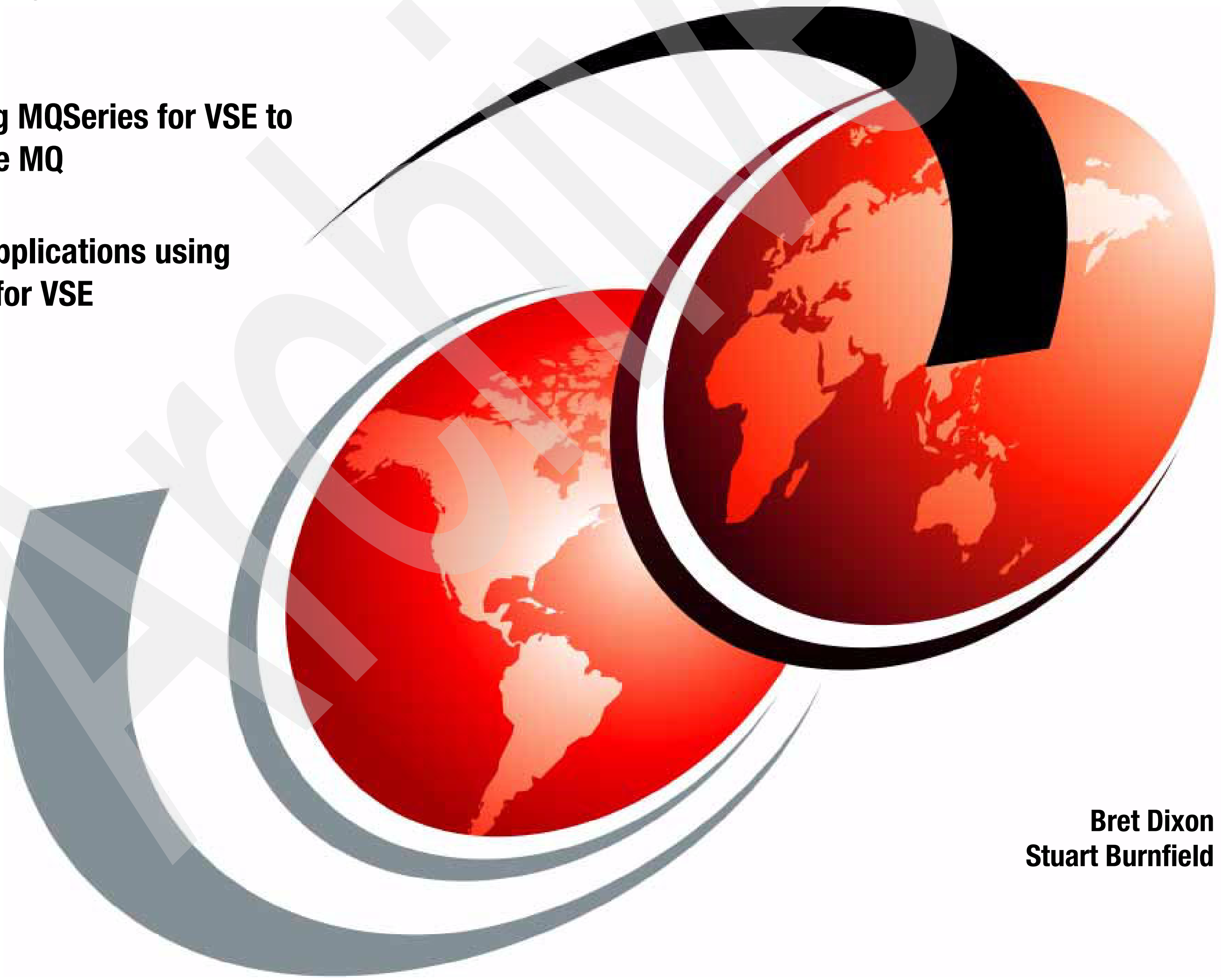
IBM

Using MQSeries for VSE

Implementing MQSeries for VSE

Connecting MQSeries for VSE to WebSphere MQ

Building applications using MQSeries for VSE



Bret Dixon
Stuart Burnfield

Redbooks

ibm.com/redbooks

International Technical Support Organization

Using MQSeries for VSE

January 2005

SG24-5647-01

Note: Before using this information and the product it supports, read the information in "Notices" on page xiii.

Second Edition (January 2005)

This edition applies to VSE/ESA Version 2, Release 5 and later, Program Number 5690-VSE for use with the MQSeries for VSE Version 2, Release1.2.

Contents

Figures

Notices

Trademarks

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

@server®
@server®
Redbooks (logo) ™
Redbooks™
ibm.com®
CICS/VSE®
CICS®
CUA®
DB2®
EPILOG®
ETE™
IBM®
Language Environment®
Lotus®
MQSeries®
MVS™
MVS/ESA™
Notes®
OS/2®
OS/390®
POWER™
QBIC®
S/390®
Sequent®
SupportPac™
SLC™
Tivoli®
VM/ESA®
VSE/ESA™
VTAM®
WebSphere®
XDE™
z/OS®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook will help you get started with MQSeries® for VSE V2R1.2. It explores why and how to use MQSeries with the VSE operating system and shows why MQSeries is more than just another queuing system. This book is meant for those who know VSE but have only a slight understanding of MQSeries and how to use it.

This redbook contains an overview of MQSeries installation and configuration, explains how to connect clients to servers and servers to servers, how to trigger applications, and the MQI programming interface. It describes the command interface, how the libraries are used, and how to edit, compile, test, and debug a program. This redbook also provides information about performance and security for the MQSeries for VSE V2.1.2. It explains what has to be done to use MQSeries for communication between VSE and other platforms, such as OS/390® and Windows® XP. It provides examples to guide you through the administrative process of defining connections between different systems; information regarding security, backup, and recovery is also provided.

The team that wrote this redbook

This edition was produced by IBM GSA's Australian Programming Centre, Perth, Western Australia, by:

Bret Dixon is a Software Engineer in IBM Australia. He has 25 years of experience in IBM systems and software development, including mainframe and distributed platforms. He holds a degree in management information systems, and has worked at IBM as a consultant for eight years. His areas of expertise include software engineering under OS/390, VSE, and UNIX®, using COBOL, C, and Assembler languages.

Stuart Burnfield is a freelance technical writer currently working at IBM Australia. He has over 12 years of experience in information development, as well as eight years of experience as a software developer.

The first edition of this redbook was produced by the International Technical Support Organization, Poughkeepsie Center. Primary contributors included Gerard Martinelli, a consultant systems engineer from IBM France, Simon Davitt, an independent consultant from IBM UK, and Bret Dixon.

Thanks to the following person for his invaluable contributions to this project:

Hanns-Joachim Uhl
IBM Böblingen

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM® Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-5647-01
for Using MQSeries for VSE
as created or updated on January 26, 2005.

January 2005, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Chapter 2, Configuration. Section 2.1, Global System Definition, has been expanded to explain the queue manager's log and trace settings, and its event settings.
- Chapter 2, Configuration. Section 2.3, Configuring channels, has been expanded to explain SSL and Exit configuration for channels.
- Chapter 5, System administration. This new chapter explains Programmable Command Formats (PCF), MQSeries Commands (MQSC), Instrumentation Events, and the automatic reorganization feature. Where applicable, the chapter provides and explains program examples.
- Chapter 7, Extended channel features. This new chapter explains SSL-enabled channels and steps through the process of generating public and private keys and digital certificates. It also explains channel exits and steps through a programming example.
- Chapter 8, Application programming. This chapter now includes a new section, 8.6 MQSeries–CICS® Bridge, which explains the MQSeries–CICS bridge support and steps through programming examples for DPL and 3270 bridging.
- Appendix B, Sample programs. This appendix now includes program samples for PCF, PCF Escape, instrumentation events, DPL bridging, 3270 bridging, and Java™.
- Appendix C, Sample JCL. This appendix now includes JCL samples for running the MQPMQSC utility, which supports MQSC commands from batch.

Changed information

- Chapter 1, Installation. Information describing installation has been updated to explain more accurately the installation process for MQSeries for VSE V2.1.2. This includes changes to nearly all sections of the chapter.
- Chapter 2, Configuration. Apart from new sections explaining extended queue manager and channel configuration, information describing configurable attributes of MQSeries objects has been updated to describe these attributes more accurately. This includes changes to nearly all sections of the chapter.
- Chapter 3, Operation. Information describing operational features has been updated to describe new or changed operational displays and settings.

- Chapter 4, Monitoring. Information describing monitoring has been updated to better describe monitor displays. Information explaining system queues has also been expanded.
- Chapter 6, Distributed queuing. This chapter has been updated to describe attributes involved in distributed queuing that are new or have changed since the previous edition of this redbook.
- Chapter 8, Application programming. This chapter has been extensively rewritten to better describe application programming for MQSeries for VSE V2.1.2. It provides more detailed information regarding copybook and include files, better detail on how triggering works, and how and when to use reply queues. It also contains some information about message expiry.
- Chapter 9, Security. This chapter has been extended to describe channel security exits and SSL-enabled channels as additional methods for securing message traffic between queue managers and clients.
- Chapter 10, Performance. This chapter now provides a better explanation of MQSeries attributes and settings that can affect MQSeries performance. It also contains more information about common MQSeries transactions that run and use resources in your CICS environment.
- Chapter 11, Problem determination. This chapter has been updated to better describe common and known problems with the MQSeries for VSE product.
- Some chapter titles have been renamed, and the position of some chapters has moved to provide a better overall structure to the redbook. The chapter describing the Batch Interface is now included in Chapter 8, Application programming.
- All screen images reproduced in the book have been updated to reflect the latest screens for MQSeries for VSE V2.1.2.

1

Installation

In this chapter, we explain how to install MQSeries for VSE V2.1.2. We provide a step-by-step explanation for installing MQSeries on VSE and document where the installation processes differ when installing MQSeries to run under CICS TS rather than CICS/VSE. We also provide hints and tips for setting up your MQSeries environment. We strongly recommend that you first read the *MQSeries for VSE System Management Guide*, GC34-5364.

1.1 MQSeries prerequisite software levels

The prerequisite software levels for MQSeries for VSE 2.1.2 are defined in the *MQSeries for VSE System Management Guide*, GC34-5364. These are:

- VSE/ESA™ 2.5 or later
- CICS/VSE 2.3 or CICS TS 1.1 or later
- VTAM® for VSE/ESA 4.2 or later
- Language Environment/VSE 1.4 Runtime Library or later

If you intend to use TCP/IP for cross-platform MQSeries communication or client connections, you must have:

- TCP/IP for VSE/ESA 1.4 or later

Each of these prerequisites should be at the latest maintenance level.

Although VSE/ESA 2.5 is the minimum prerequisite for MQ/VSE V2.1.2, at the time of publication, service for VSE/ESA 2.5 had been discontinued. It is always recommended that the latest versions of the prerequisites be installed because newer features of MQSeries may exploit new services only available with the latest prerequisite software.

1.2 MQSeries PSP bucket

Many problems experienced with MQSeries for VSE are related to the installation process. A careful reading of the MQSeries Product Service Planning (PSP) bucket can avoid some of these problems.

The MQSeries PSP bucket is a documentation file that contains installation information and high-impact, pervasive APAR information. The PSP bucket is not shipped with your MQSeries installation tape. You should request a copy of the bucket from your IBM representative.

You should read the PSP bucket before you begin installation to determine whether there are changes to the installation process or PTFs that must be applied prior to installation. MQSeries-specific PTFs cannot be applied until you have installed the product in MSHP.

The PSP bucket is also available online. It is kept up to date by IBM, contains the latest APAR/PTF information and reports the current content of active information APARs for the product. This information is available at:

https://techsupport.services.ibm.com/server/390.psp390

This link prompts you to request an IBM ID, which you can do online. Once you have signed on, a search window is displayed where you should enter:

```
MQSVSE 2ZZ
```

IBM also provides service information, including the content of active information APARs online at:

http://www.ibm.com/software/integration/mqfamily/support/summary

1.3 Installing under CICS/VSE® and CICS TS

Chapter 2 of the *MQSeries for VSE System Management Guide*, GC34-5364, documents the installation process. In this section, we provide a hands-on, step-by-step explanation of the installation process from start to finish. Unless otherwise noted, the installation process is identical for CICS/VSE and CICS TS.

Installation will generally be under CICS TS. This is because new features of MQSeries for VSE tend to be available only when running under CICS TS. However, if you currently run a CICS/VSE system, and do not want to introduce a CICS TS system, MQSeries can be installed under CICS/VSE with certain features disabled, including MQ security and the MQ CICS bridge.

1. The first decision to make is whether to install MQSeries in the default library or your own library. The library contains all the MQSeries objects that are needed to install and operate your MQSeries system and build Message Queue Interface (MQI) application programs.

   For this example, we want to create our own library. There are two ways to create the library:

   – An IDCAMS step
   – A LIBR step

   The IDCAMS step creates a VSAM managed file; the LIBR creates a BAM file. Your VSE system programmer may have a preference for which type you use.

   The LIBR step is probably easier. The library can be created using the following Job Control Language (JCL):

   ```
   * $$ JOB JNM=MQMSUBL,CLASS=0,DISP=D
   * $$ LST DEST=(host,user)
   // JOB MQMSUBL Define the MQSeries installation library
   // DLBL mylib,'l.f.i',yyyy/ddd
   // EXTENT ,volume,,,n,m
   // EXEC LIBR
   DEFINE L=mylib
   DEFINE S=mylib.sublib
   /*
   /&
   * $$ EOJ
   ```

   where:

   – `mylib` is the new library name, for example, VSE1
   – `sublib` is the new sublibrary name, for example, MQLIB
   – `l.f.i` is your local file ID, for example, VSE1.ID
   – `yyyy/ddd` is the file retention year and day, for example, 2005/366
   – `volume` is the local disk volume name, for example, VSEQ02
   – `n,m` is the start track and size required, for example, 2251,150

   See the IBM *VSE/ESA System Control Statements*, SC33-6613 documentation for further information about DLBL, EXTENT, and LIBR.

   Using this JCL, we create the library and sublibrary VSE1.MQLIB (you may choose a different name). Note that the number of tracks for the library we used is only 150. This is large enough for the MQSeries library, which, for MQSeries for VSE 2.1.2, is approximately 120 tracks on a 3390. However, if you want to use the library for other purposes and create multiple sublibraries, you need to specify a size that will accommodate your particular requirements.

   Alternatively, you may want to use an existing library for the new MQSeries sublibrary. In this case, the DLBL and EXTENT cards are irrelevant, and the JCL can be trimmed to the EXEC and DEFINE for the sublibrary.

2. At this point, it is convenient to add the new sublibrary to your VSE standard labels. Standard labels are common DLBLs used by your VSE system. By adding the sublibrary

to our standard labels definition now, we avoid having to include DLBL and EXTENT cards in all our following JCLs.

To temporarily add the sublibrary to standard labels, run the following JCL:

```
* $$ JOB JNM=MQSTDLAB,CLASS=0,DISP=D
// JOB MQSTDLAB
// OPTION STDLABEL=ADD
// DLBL mqlib,'l.f.i'
// EXTENT ,volume
/*
/&
* $$ EOJ
```

where:

- `mylib` is the new library name, for example, VSE1
- `l.f.i` is your local file ID, for example, VSE1.ID
- and `volume` is the local disk volume name, for example, VSEQ02

Note that this job must run in the background partition. This addition to the standard labels remains until your VSE system is IPL. To make the addition permanent, ask your VSE system programmer to add the DLBL and EXTENT to the appropriate standard labels procedure.

3. Our next step is to restore the MQSeries library from the installation tape. Once again, there are two ways of achieving this:

   - Using the VSE Interactive User Interface
   - Submittinga customized JCL

   In our example, we customize the JCL provided in the *MQSeries for VSE System Management Guide*, GC34-5364, as follows:

```
* $$ JOB JNM=MQMTAPE,CLASS=0,DISP=D
* $$ LST DEST=(host,user)
// JOB MQMTAPE Restore MQSeries from tape
// ASSGN SYS006,cuu
// MTC REW,SYS006
// EXEC MSHP,SIZE=1M
INSTALL PRODUCT FROMTAPE ID='MQSeries...2.1.2' -
PROD INTO=mylib.sublib
/*
/&
* $$ EOJ
```

   where:

   - `mylib` is the new library name, for example, VSE1
   - `sublib` is the new sublibrary name, for example, MQLIB
   - `cuu` is the tape drive address, for example, 181

   To verify the restore of the MQSeries library, you can run a LIBR job to examine the contents of your new sublibrary. For example:

```
* $$ JOB JNM=MQLIBDIR,CLASS=0,DISP=D
* $$ LST DEST=(host,user)
// JOB LIBR
// EXEC LIBR
 ACC SUBLIB=mylib.sublib
 LD *.*
/*
/&
* $$ EOJ
```

4. MQSeries is now defined to the system history file. It is a good idea to apply all prerequisite PTFs at this point. Some PTFs have specific actions that must be carried out after the PTF is applied but before MQSeries is started, so you should examine the associated APAR for details before applying each PTF. For a list of relevant PTFs, review the PSP bucket or online service page as explained above.
5. At this point, you need a VSAM catalog for your MQSeries VSAM files. These VSAM files will be used to store MQSeries configuration information, and more significantly, MQ message data. If you do not already have one, you can ask your VSE system programmer to create a VSAM user catalog, or you can define one yourself using the VSE Interactive User Interface or by writing your own JCL. You should create a user catalog that controls sufficient VSAM data space for your MQSeries system files and queues.

   MQSeries for VSE system data does not exceed 10 MB. This can increase if you do not maintain your system logs. The size of your queues will depend on your expected application data volume. One method of estimating the size of your queues is to multiply the average message length by the maximum message traffic expected on the queue between reorganizations. The estimate should not be based on the expected queue depth alone. This is because MQSeries for VSE retains messages after they have been read (using MQGET). The expected queue depth represents the maximum of unread messages, not the total number of messages that can be in your queue at any time. You may also want to confer with your VSE system programmer to include VSAM considerations in your estimate.

   Before you can continue with the MQSeries installation, you need:

   – A user catalog name.

   – The volume where the user catalog resides.

   – A list of volumes with dataspaces recognized by the user catalog (there may be only one volume in your list).

   Once again, it is convenient to add a DLBL and EXTENT for the VSAM user catalog to your standard labels definition. To add the user catalog to your standard labels definition, run the following JCL in the BG:

   ```
   * $$ JOB JNM=MQSTDLAB,CLASS=0,DISP=D
   // JOB MQSTDLAB
   // OPTION STDLABEL=ADD
   // DLBL catnam,'u.c.n',0,VSAM
   // EXTENT ,volume
   /*
   /&
   * $$ EOJ
   ```

   where:

   – `catnam` is the DLBL filename, for example, MQMCAT
   – `u.c.n` is your VSAM user catalog name, for example, MQ.USER.CATALOG
   – `volume` is the local disk volume name, for example, VSEQ02
6. We can now allocate and initialize the required MQSeries files. Samples for jobs to achieve this are found in the new MQSeries sublibrary. The steps involved include:

   – Creating the MQSeries setup file
   – Creating the MQSeries configuration file
   – Creating cluster definitions for MQSeries queues

   You can get a copy of these samples using LIBRP via the VSE Interactive User Interface, or punch the file to your reader using the following JCL:

   ```
   * $$ JOB JNM=MQPUN,CLASS=0
   * $$ LST DEST=(host,user)
   ```

```
// JOB LIBPUN
// DLBL mylib,'l.f.i'
// EXTENT ,volume
// EXEC LIBR
ACC SUBLIB=mylib.sublib
* $$ PUN DEST=(host,user)
PUNCH MQJSETUP.Z F=NOHEADER EOF=N
* $$ PUN DEST=(host,user)
PUNCH MQJCONFG.Z F=NOHEADER EOF=N
* $$ PUN DEST=(host,user)
PUNCH MQJQUEUE.Z F=NOHEADER EOF=N
/*
/&
* $$ EOJ
```

If you are using LIBRP, the sample files are the bold names in the preceding JCL.

7. To create the MQSeries setup file, we need to edit the MQJSETUP.Z sample JCL file. A comment at the beginning of this file tells you exactly what you need to change. Changes include the following:

   – POWER™ cards from * ** to * $$
   – Catalog ID for VSAM DELETE and DEFINE
   – Dataspace volume ID for VSAM DEFINE
   – Catalog DLBL name for LOADFL DLBL

   For our example, we also need to change:

   – MQSeries sublibrary name to our new sublibrary name

   Having made these changes, we can run the job to create a VSAM cluster for the MQSeries setup file and load it with MQSeries setup data. You can expect a maximum return code of 0 for this job.

8. To create the MQSeries configuration file, we need to edit the MQJCONFG.Z sample JCL file. Once again, comments at the beginning of this file tell you exactly what needs to be changed. Changes include the following:

   – POWER cards from * ** to * $$
   – Catalog ID for VSAM DELETE and DEFINE
   – Dataspace volume ID for VSAM DEFINE

   The MQSeries configuration file contains, among other things, queue and channel definitions. Consequently, we need to create it with a size that matches the number of queues and channels we intend to define. This is for definitions only, not actual queue messages. For our example, we use only 10 queues and four channels. The sample JCL specifies a RECORDS definition of 300 primary and 100 secondary, which is sufficient for our purposes. Consequently, we do not need to change the RECORDS definition in the sample JCL.

   Having made these changes, we can run the job to create a VSAM cluster for the MQSeries configuration file. You can expect a maximum return code of 0 for this job.

9. To create cluster definitions for MQSeries queues, we need to edit the MQJQUEUE.Z sample JCL file. Once again, we need to change the following:

   – POWER cards from * ** to * $$
   – Catalog ID for VSAM DELETE and DEFINE
   – Dataspace volume ID for VSAM DEFINE

   The MQJQUEUE job provides definitions for system queues and sample definitions for application queues. The system queues include:

– MQFERR - Dead letter queue file
– MQFLOG - System log queue file
– MQFMON - MQI monitor queue file
– MQFREOR - Auto-reorganization file

The MQJQUEUE job contains // SETPARM cards that determine whether or not additional files are created for the Programmable Command Formats (PCF) and Instrumentation Events (IEVT) features. To create files for associated system queues, set these cards as follows:

```
// SETPARM PCF=Y
// SETPARM IEVT=Y
```

Setting these parameters to `Y` results in the creation of the following additional system queue files:

– MQFACMD - PCF system command queue file
– MQFARPY - PCF system reply queue file
– MQFIEQE - Queue manager event queue file
– MQFIECE - Channel event queue file
– MQFIEPE - Performance event queue file

Unless you want to change the names of these files, the definitions should remain unchanged. The only exception to this is if you intend to allow your system logs to grow to a significant size (in this case, more than 300 entries). If so, you may want to increase the RECORDS definition for these system files. If you do not take this precaution, these files may involve numerous secondary extents that ultimately impact performance.

For the sample definitions for application queues, we need to consider the following:

– Preferred CLUSTER name for queue files
– Preferred FILE name for queue files
– Expected length of queue messages
– Expected queue message depth
– How many queues are needed

We do not need to change the default CLUSTER and FILE names; however, you may want to make these names conform to your own naming standards. Note that the FILE names will match the FCT entries in CICS later. (Do not be confused by the sample names MQFI001, MQFO001, MQFI002, and so forth. This naming is historic and does not imply that some files are for input and others for output.)

More importantly, we need to consider the expected length of the queue messages. This affects our RECORDSIZE specification. The default RECORDSIZE in the sample JCL is not mandatory. For each queue file, the RECORDSIZE should be changed to the expected length of queue messages for queues we intend to define in that file. For example, if we intend to define a queue in file MQFI001 that will contain message data as short as 200 bytes and as long as 1000 bytes, we should change the RECORDSIZE to, for example:

```
RECORDSIZE(1000 1736)
```

In this example, we chose 1000 bytes as the average record length because each message is appended with a 736-byte MQSeries context header. If the average message data written to this queue is around 264 bytes, we can expect an average record length of 1000 bytes. A maximum RECORDSIZE of 1736 will allow message data of up to 1000 bytes plus the MQSeries context header.

Once you have decided on your RECORDSIZE, you should consider the associated Control Interval Sizes (CISZ) to optimize disk usage. You may want to check with your VSE system programmer to determine the best CISZ for your VSAM definitions.

The *MQSeries for VSE System Management Guide*, GC34-5364, strongly recommends that you define one queue per VSAM file. There are significant performance reasons for this recommendation. Another consideration is that each queue may have its own average and maximum message sizes. Consequently, it makes sense to define a VSAM file that accommodates the specific dimensions of an intended queue.

We also need to consider the expected message traffic on each queue between reorganizations. This affects the RECORDS definition in the sample JCL. For example, if we expect that a queue may grow to 10,000 messages (5,000 read and 5,000 unread), we should ensure our RECORDS definition for the VSAM file of that queue is large enough to accommodate the message load. In this case, for example:

```
RECORDS(10000 1000)
```

The 5,000 read messages are cleared during the MQSeries queue reorganization.

The number of VSAM files you define depends on the number of application queues you need. You can increase or decrease the number of defines in the sample JCL to match your needs.

The KEYS definition is for MQSeries use and should not be changed.

Having made all of these changes to the MQJQUEUE file, we can run the job to create VSAM clusters for the MQSeries system and application queues. You can expect a maximum return code of 4 for this job because the **`DELETE`** commands will generate a warning if the files do not already exist.

10. We now need to create MQSeries-specific definitions in CICS. These include:

    – CSD definitions
    – DCT definitions
    – FCT definitions

    Sample files to facilitate each of these are provided in the MQSeries sublibrary. The files we need include:

    – MQCICDCT.Z for DCT definitions

    – If you are installing MQSeries in CICS/VSE, you need sample file MQJCSD.Z for CICS/VSE CSD definitions

    – If you are installing MQSeries in CICS TS, you need sample file MQJCSD24.Z for CICS TS CSD definitions

    – If you are installing MQSeries in CICS/VSE, you also need MQCICFCT for FCT definitions

    To get a copy of these samples, use LIBRP via the VSE Interactive User Interface, or punch them to your reader using the following JCL:

```
* $$ JOB JNM=MQPUN,CLASS=0
* $$ LST DEST=(host,user)
// JOB LIBPUN
// EXEC LIBR
ACC SUBLIB=mylib.sublib
* $$ PUN DEST=(host,user)
PUNCH MQJCSD.Z   F=NOHEADER EOF=N
* $$ PUN DEST=(host,user)
PUNCH MQJCSD24.Z F=NOHEADER EOF=N
* $$ PUN DEST=(host,user)
PUNCH MQCICDCT.Z F=NOHEADER EOF=N
* $$ PUN DEST=(host,user)
PUNCH MQCICFCT.Z F=NOHEADER EOF=N
/*
/&
* $$ EOJ
```

If you are using LIBRP, the sample files are the bold names in the preceding JCL.

11. To add the MQSeries CSD definitions, we need to edit the MQJCSD.Z or MQJCSD24.Z sample JCL file (whichever is applicable in your case). We need to change the POWER cards from * ** to * $$. The only other changes we *may* need to make are:

    – CSD group name
    – CICS SIT list name

    The default CSD group name is MQM. The default CICS SIT list name is VSELIST. You should check with your CICS system programmer regarding these names before running the MQJCSD job.

    For our example, the target CICS system for MQSeries uses DFHLIST. This particular list is IBM protected, so we need to define a new list or use an existing list other than DFHLIST. Your CICS system programmer can tell you which list to use and guarantee that your CICS SIT GRPLIST identifies this list. For our purposes, we use VSELIST, and consequently, we do not need to change the sample JCL. Because our SIT GRPLIST is DFHLIST, this needs to change to VSELIST and CICS must be restarted in COLD mode (there is no need to stop and restart CICS at this point). If you are creating a new list, remember to append the contents of DFHLIST (and any other relevant list) in your new list. Your CICS system programmer can perform this task.

    For a CICS TS installation, you may also need to change the CSD FILE definitions in the sample JCL. These must match the VSAM clusters you defined earlier for system and application queues.

    Before running the sample JCL, you need to check that your standard labels include a DLBL for DFHCSD. If so, your CSD definitions are added to the CICS system identified by this DLBL. If there is no DFHCSD in your standard labels definition, or it points to the wrong CICS system, you must add a DLBL to the sample JCL. For example:

    ```
    // DLBL cicsuct,'usercat',,VSAM
    // DLBL DFHCSD,'user.dfhcsd',,VSAM,CAT=cicsuct
    ```

    The startup JCL for your CICS system must include these DLBLs if they are not in your standard labels definition.

    Having made the relevant changes to the MQJCSD or MQJCSD24 file, we can run the job to create the MQSeries CSD definitions. You can expect a maximum return code of 4 for this job.

12. The MQCICDCT.Z sample file contains definitions for MQSeries transient data queues. For MQSeries for VSE 2.1.2, there are three transient data queues:

```
MQER
MQXP
MQIE
```

To apply this definition to your CICS system, you need to provide the MQCICDCT.Z sample file to your CICS system programmer. Your system programmer appends the sample definition to existing DCT definitions and rebuilds the DCT phase. This phase must be rebuilt and available to CICS before you can start MQSeries.

If you are activating the MQ security feature, you need to modify the MQCICDCT.Z file before you rebuild your CICS DCT phase. When security is active, MQ requires that the DCT definitions include the USERID parameter and identify a user who will have sufficient authority relevant to the purpose of the particular transient data queue. Information regarding this requirement, and MQ security in general, is available in the *MQSeries for VSE System Management Guide*, GC34-5364.

13. The MQCICFCT.Z sample file is only applicable when you are installing MQSeries in CICS/VSE. It contains definitions for MQSeries files.

For CICS TS, the CICS File Control Table (FCT) is modified by CSD definitions. These definitions are applied when you apply your CSD definitions covered previously. Consequently, if you are installing MQSeries in CICS TS, you can skip this step.

For a CICS/VSE installation, you need to ensure that the FCT dataset names in the MQCICFCT.Z sample file match the VSAM clusters you defined earlier for system and application queues.

When the sample file is ready, you should provide it to your CICS system programmer. Your system programmer appends the sample definition to existing FCT definitions and rebuilds the FCT phase. This phase must be rebuilt and available to CICS before you can start MQSeries.

14. We are now ready to modify our CICS startup JCL. To achieve this, MQSeries is shipped with another sample file MQJLABEL.Z. This file contains DLBLs for MQSeries system and application files. Once again, the contents of this file must be modified to match the VSAM clusters defined earlier for system and application queues.

Once this file has been modified, you can either include the contents in your CICS startup JCL or catalog the contents as a PROC and EXEC the procedure from your CICS startup JCL. Your CICS system programmer makes this decision.

15. The LIBDEF SEARCH path in your CICS startup JCL also must be modified to include the MQSeries sublibrary. The MQSeries sublibrary can appear anywhere in the chain. For example:

```
// LIBDEF *,SEARCH=(PRD2.CONFIG,
                    mylib.sublib,
                    PRD1.BASE,
                    PRD2.SCEEBASE,
                    ...)
```

This ensures that CICS can find the MQSeries programs that reside in the MQSeries sublibrary. (At this point, it is still unnecessary to stop and restart your CICS system.)

16. There is a warning in the *MQSeries for VSE System Management Guide*, GC34-5364, that your CICS Journal Control Table (JCT) definition should have a BUFSIZE greater than the largest physical VSAM record you expect in your MQSeries queue files. This is to avoid AFCL abends during normal operation of MQSeries. MQSeries requires that you have the JCT active in CICS.

Once again, your CICS system programmer can confirm that this is the case or modify the existing JCT definition to meet MQSeries requirements.

The *MQSeries for VSE System Management Guide*, GC34-5364, also recommends that you define your FCT entries with LOG=YES for recovery and restart purposes. However, the sample JCL to define your FCT entries includes several definitions where LOG=NO. This is the case for VSAM files not used for message queues. For application message queues, you should always specify `LOG=YES`.

17. MQSeries for VSE 2.1.2 includes Language Environment® C, COBOL, and Assembler modules. For this reason, your CICS system must be correctly defined to run these programs.

    Refer to 1.5.1, “Language Environment definitions” on page 14 for Language Environment setup requirements for CICS. These requirements must be met before you start MQSeries.

18. We are now ready to start CICS and initialize MQSeries. If your CICS system is already running, you must stop and restart it. Once your CICS system is running, you need to establish a session in native CICS and run the MQSeries system installation transaction:

    ```
    MQSU
    ```

    This transaction populates the MQSeries configuration file with initial values. On successful completion, this transaction displays:

    ```
    MQSU: MQSeries install completed,  nnnn input records read.
    ```

    The `nnnn` in this message represents the number of configuration records processed by the MQSU transaction. This number can change depending on your product's level of service. The actual number should match the number of records loaded by the MQJSETUP installation job. Note that this job is also rerun for some PTFs. Whenever you run MQJSETUP, you should also rerun the MQSU transaction in CICS.

    If you experience an AFCL or AFCP abend running this transaction, check your CICS JCT definition.

    You can also run the MQSU transaction with the UC parameter, that is:

    ```
    MQSU UC
    ```

    This converts all configuration to uppercase when it is loaded into the MQSeries configuration file. Messages and screen displays will be in uppercase.

    You only need to run the MQSU transaction once during installation. You do not need to run it each time you start MQSeries. At this point, your MQSeries installation is complete.

1.4 Initial configuration

Once you have installed MQSeries for VSE 2.1.2, you must do some initial configuration before you can start your MQSeries system. Initial configuration includes:

- Defining the local queue manager
- Defining system queues

Complete the following steps:

1. Before we can start MQSeries, we must define the local queue manager. This is done using the MQMT master terminal transaction. However, before we can run the master terminal transaction, we must run the MQSE transaction in native CICS. For example:

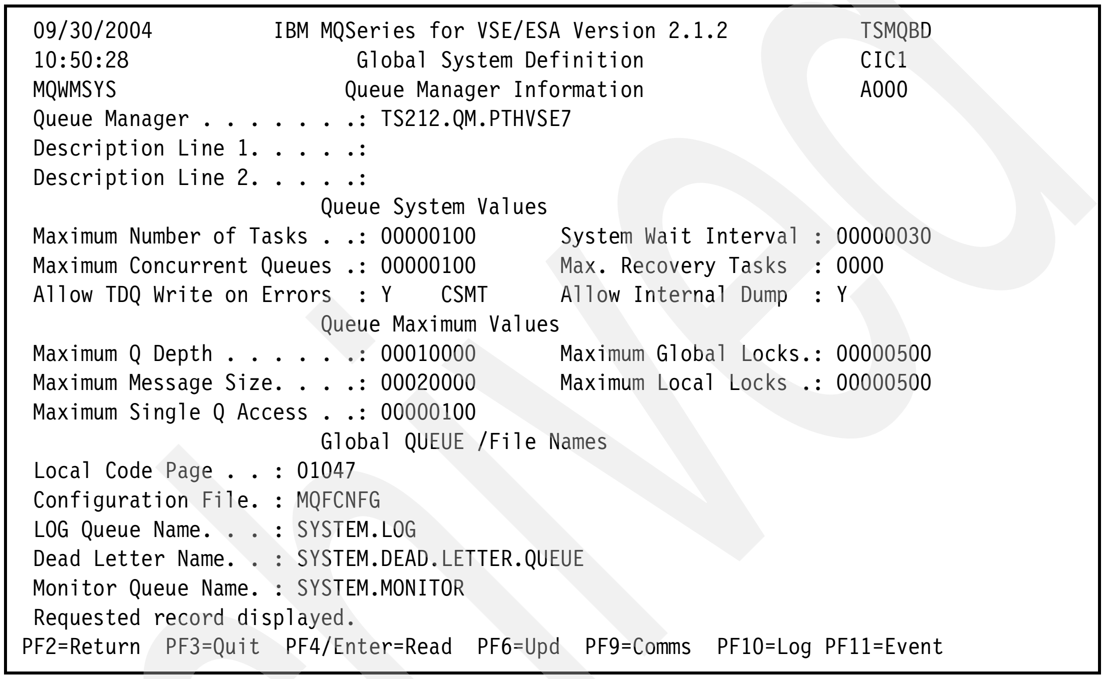
    ```
    MQSE
    ```

The MQSE transaction establishes the MQSeries environment within CICS. A successful completion of this transaction displays:

```
MQSE: MQSeries environment setup completed.
```

We can now run the MQMT transaction to define the local queue manager. The master terminal transaction displays a main menu and allows the selection of options. Option 1 allows you to configure MQSeries objects including the local queue manager.

To define the local queue manager, select option **1** (Configuration) and suboption **1** (Global System Definition). This displays the screen shown in Figure 1-1.



```
 09/30/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
 10:50:28                  Global System Definition                 CIC1
 MQWMSYS                  Queue Manager Information                 A000
 Queue Manager . . . . . . .: TS212.QM.PTHVSE7
 Description Line 1. . . . .:
 Description Line 2. . . . .:
                        Queue System Values
 Maximum Number of Tasks . .: 00000100      System Wait Interval : 00000030
 Maximum Concurrent Queues .: 00000100      Max. Recovery Tasks  : 0000
 Allow TDQ Write on Errors  : Y    CSMT     Allow Internal Dump  : Y
                        Queue Maximum Values
 Maximum Q Depth . . . . . .: 00010000      Maximum Global Locks.: 00000500
 Maximum Message Size. . . .: 00020000      Maximum Local Locks .: 00000500
 Maximum Single Q Access . .: 00000100
                        Global QUEUE /File Names
 Local Code Page . . : 01047
 Configuration File. : MQFCNFG
 LOG Queue Name. . . : SYSTEM.LOG
 Dead Letter Name. . : SYSTEM.DEAD.LETTER.QUEUE
 Monitor Queue Name. : SYSTEM.MONITOR
 Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF6=Upd  PF9=Comms  PF10=Log PF11=Event
```

Figure 1-1 Initial local queue manager definition screen

At this point, you must change the default values to your specific requirements. Most importantly, you should select a suitable queue manager name.

You can also define your queue manager's communications setting using **PF9**, your log and trace settings with **PF10**, and your event settings with **PF11**. Each of these is described in greater detail in Chapter 2, “Configuration” on page 27.

You may choose to keep all the initial defaults. You can always change your queue manager definition at a later stage using MQMT.

Once you have finalized your queue manager definition, press **PF6** to permanently update your queue manager definition.

2. Before starting MQSeries, you should create your system queues. There are several system queues pertaining to general operation, PCF command support, instrumentation events, and the MQ CICS bridge. Of immediate importance are those system queues pertaining to general operation. These include:

   – SYSTEM.LOG
   – SYSTEM.DEAD.LETTER.QUEUE

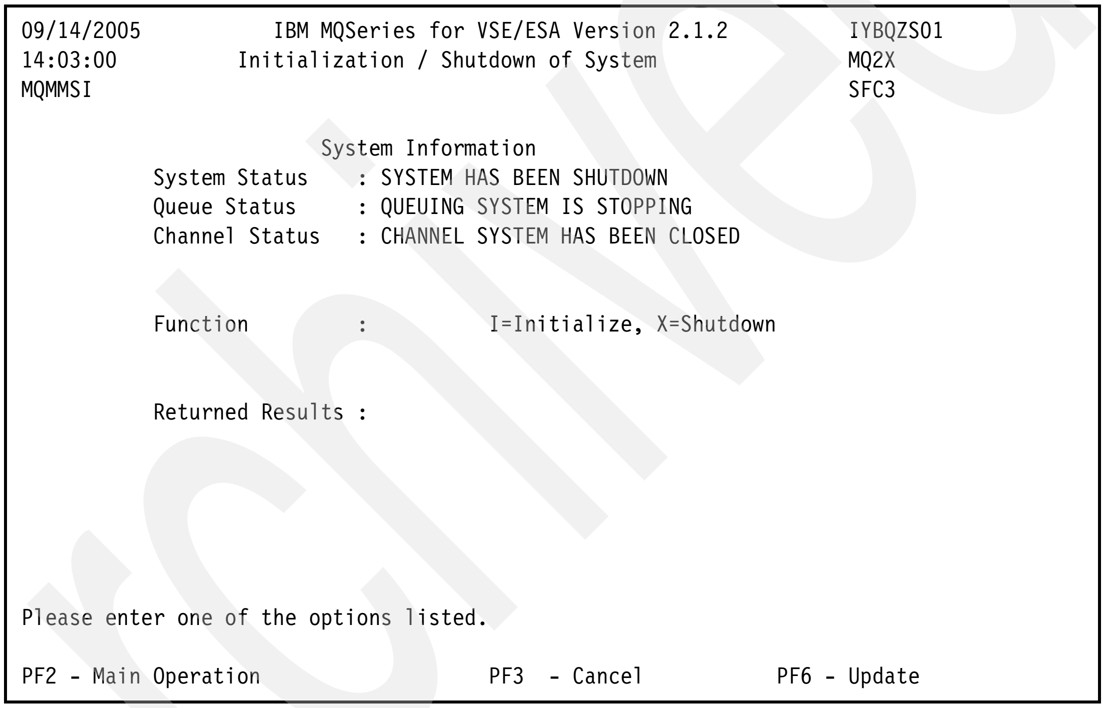
   You may have chosen different names rather than these defaults. You should create these queues before starting MQSeries for the first time. Queues can be created using the master terminal transaction MQMT and option 1.2. Queue creation is explained in section

"Configuring queues" on page 40 and in the *MQSeries for VSE System Management Guide*, GC34-5364.

Local queue messages ultimately reside in VSAM KSDS files. These files were defined by installation job MQJQUEUE.Z, described above. The MQJQUEUE.Z sample includes definitions for files MQFLOG and MQFERR. These files are intended to host the SYSTEM.LOG and SYSTEM.DEAD.LETTER.QUEUE, respectively. The host VSAM file is part of the local queue definition.

3. We can now start MQSeries for the first time. There are several ways to start MQSeries:
   - Define your CICS PLTPI to start MQSeries during CICS system startup.
   - Use the MQMT master terminal program, option 2.4.
   - Run the MQSE and MQIT transactions in native CICS.

   For our example, we use MQMT option **2** (Operations) with suboption **4** (Initialization). This displays the screen shown in Figure 1-2.



```
09/14/2005          IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS01
14:03:00         Initialization / Shutdown of System                MQ2X
MQMMSI                                                              SFC3

                         System Information
           System Status    : SYSTEM HAS BEEN SHUTDOWN
           Queue Status     : QUEUING SYSTEM IS STOPPING
           Channel Status   : CHANNEL SYSTEM HAS BEEN CLOSED


           Function         :          I=Initialize, X=Shutdown


           Returned Results :







Please enter one of the options listed.

PF2 - Main Operation                  PF3  - Cancel          PF6 - Update
```

Figure 1-2 Initialization/Shutdown of System screen

To start your MQSeries system, select **I** (Initialize) at Function and press **PF6**. This should display the following result on the screen:

```
System initialized on   mm/dd/yyyy at hh:mm:ss
```

You may receive a warning message if you do not have queues or channels defined.

Starting MQSeries also causes messages to be displayed on the VSE console. These messages vary depending on your local queue manager definition, but should include:

```
MQI0030I - MQSeries for VSE system starting
MQI0035I - MQSeries for VSE licensed support for 0000 clients
MQI0040I - MQSeries for VSE system started
```

The console message regarding licensed clients can be ignored. It is intended to identify the maximum number of concurrent client connections that the queue manager will accept at any one time. The restriction, however, is not enforced. At the time of publication, no additional license for clients is required, and the maximum number of concurrent clients is not capped.

Your MQSeries system is now installed and active. Before it is used for application programs, you should define application queues specific to your needs.

1.5 CICS environment for MQSeries

MQSeries for VSE is a set of CICS application programs. These programs have been developed in different programming languages such as Assembler, COBOL, and C. Therefore, all CICS requirements for running such programs must be met.

Although some of these requirements are discussed in 1.3, “Installing under CICS/VSE® and CICS TS” on page 2, the overall requirements for your CICS system are provided here for your convenience. The areas where your CICS system is affected include:

- Language Environment definitions
- Systems Network Architecture (SNA) definitions
- Journal Control Table (JCT) definitions
- File Control Table (FCT) definitions
- Destination Control Table (DCT) definitions
- MQSeries definitions
- CICS system startup JCL

Each of these is discussed in the following sections.

1.5.1 Language Environment definitions

Language Environment for VSE provides common services and language-specific routines in a single runtime environment for applications written in Language Environment/VSE-conforming versions of C, COBOL, and PL/I high-level languages, as well as the High Level Assembler language. For applications using Language Environment/VSE services, you must provide:

- Language Environment/VSE runtime options
- Language Environment/VSE CSD definitions

For Language Environment/VSE DCT definitions, refer to 1.5.5, “Destination Control Table (DCT) definitions” on page 19.

Language Environment/VSE runtime options

For flexibility reasons, it is possible to change runtime characteristics by using options tables. Because batch and CICS environments are very different, runtime options are also different. Language Environment/VSE comes with IBM-supplied runtime values that you may want to change.

The source of the options tables may be found in sublibrary PRD2.SCEEBASE:

- CEEDOPT.A for batch environments
- CEECOPT.A for CICS
- CEEUOPT.A for user-specific applications

Sample jobs to generate these options tables are also provided:

- CEEWDOPT.Z for batch environments
- CEEWCOPT.Z for CICS
- CEEWUOPT.Z for user-specific applications.

We recommend that you do *not* change the default values of CEEWCOPT provided by Language Environment/VSE unless requested by IBM maintenance people to fix APARs. However, if for any reason you have to do so, try to avoid changing the following keyword values:

```
ABTERMENC=((ABEND),OVR)
ALL31=((ON),OVR)
ANYHEAP=((4K,4K,ANYWHERE,FREE),OVR)
BELOWHEAP=((4K,4K,FREE),OVR)
HEAP=((4K,4K,ANYWHERE,KEEP,4K,4K),OVR)
LIBSTACK=((4K,4K,FREE),OVR)
STACK=((4K,4K,ANYWHERE,KEEP),OVR)
STORAGE=((NONE,NONE,NONE,OK),OVR)
TERMTHDACT=((MSG),OVR)
```

Language Environment/VSE CSD definitions

Defining Language Environment/VSE programs in the CICS System Definition (CSD) file is a required installation step. MQSeries needs the phases specified in the following members defined in the CSD:

- CEECCSD.Z (Language Environment/VSE base)
- IGZCCSD.Z (for COBOL)
- EDCCCSD.Z (for C)

You can find these members in the PRD2.SCEEBASE sublibrary. The definitions can be added by submitting a JCL that executes the DFHCSDUP program. However, a better and faster way is to do the following:

1. Copy the ICCF member SKLE370 from library 59 to your personal library.
2. Uncomment lines starting with * $$ SLI.
3. Change the default list name (VSELIST) in which the Language Environment/VSE group (CEE) is added (if yours is different).
4. Submit the job for execution.

All phases are defined in the group CEE. Do not forget to start a CEDA transaction to install it online; otherwise, restart CICS in COLD mode. This is only relevant the first time you install the CSD entries.

1.5.2 Systems Network Architecture (SNA) definitions

Since V2.1, you have a choice to interconnect MQSeries queue managers. You may use either TCP/IP or SNA LU 6.2 (or both). You specify which protocol to use when you define MQSeries channels.

With TCP/IP, you do not need any additional definitions for CICS. For LU 6.2, you *must* specify sessions and connections through the CEDA transaction.

Important: Keep in mind that when VSE acts as a server, it is in a client/server mode; in such a case, only TCP/IP connections are possible.

The peer-to-peer mode is described in this section. It assumes two MQ managers connected.

Defining CICS connections

A connection is the definition of a remote system that your CICS communicates with using SNA protocols. When you define a connection with CEDA**,** you give enough information to identify the system and specify its basic attributes. You provide details about sessions you will use to communicate with the system in session definitions. CICS uses the connection name to identify the other system when the definition has been installed.

An example of a CICS CONNECTION definition is shown in Figure 1-3.

```
OBJECT CHARACTERISTICS
 CEDA  View
  Connection     : WARP
  Group          : MQSERIES
 CONNECTION IDENTIFIERS
  Netname        : DEP3900
  INDsys         :
 REMOTE ATTRIBUTES
  REMOTESystem   :
  REMOTEName     :
 CONNECTION PROPERTIES
  ACcessmethod   : Vtam            Vtam | IRc | INdirect
  Protocol       : Appc            Appc | Lu61
  SInglesess     : No              No | Yes
  Datastream     : User            User | 3270 | SCs | STrfield | Lms
  RECordformat   : U               U | Vb
 OPERATIONAL PROPERTIES
  AUtoconnect    : Yes             No | Yes | All
  INService      : Yes             Yes | No
 SECURITY
  SEcurityname   :
  ATtachsec      : Local           Local | Identify | Verify
  Bindpassword   :                 PASSWORD NOT SPECIFIED
```

Figure 1-3 CICS CONNECTION definition

This example shows a definition for connecting a VSE MQ Manager (MQM) to an OS/2® Warp MQM in LU 6.2 mode. In this example, WARP is the connection name and DEP3900 is the name of the Partner LU on the workstation. Communication manager definitions for OS/2 may be found in “CM/2 configuration file MQSERIES.NDF” on page 306.

Defining CICS sessions

Before two systems communicate, they must be logically linked through one or more sessions. You specify the logical link in a session definition that determines how the two systems can communicate. CICS does not use the session name as soon as the definition has been installed. It is only used to identify the session in the CSD.

An example of a CICS SESSION definition is shown in Figure 1-4 on page 17.

```
OBJECT CHARACTERISTICS
  CEDA  VIew
  Sessions        : WARP1
   Group          : MQSERIES
  SESSION IDENTIFIERS
   Connection     : WARP
   SESSName       :
   NETnameq       :
   MOdename       : NORMAL
  SESSION PROPERTIES
   Protocol       : Appc               Appc | Lu61
   MAximum        : 00050 , 00020      0-32767
   RECEIVEPfx     :
   RECEIVECount   : No                 No | 1-999
   SENDPfx        :
   SENDCount      : No                 No | 1-999
   SENDSize       : 04096              1-30720
   RECEIVESize    : 04096              1-30720
  OPERATOR DEFAULTS
   OPERId         :
   OPERPriority   : 000                0-255
   OPERRsl        :                0 0-24,...
   OPERSecurity   :                1 1-64,...
   USERId         :
  SESSION USAGES
   Transaction    :
   SESSPriority   : 000                0-255
  OPERATIONAL PROPERTIES
   Autoconnect    : Yes                No | Yes | All
   INservice      :                    No | Yes
   Buildchain     : Yes             Yes | No
   USERArealen    : 000                0-255
   IOarealen      : 00000 , 00000      0-32767
   RELreq         : No                 No | Yes
   Discreq        : No                 No | Yes
   NEPclass       : 000                0-255
RECOVERY
   RECOvoption    : Sysdefault         Sysdefault | None
```

Figure 1-4 CICS SESSION definition

In this example, WARP1 is the session name for the Warp connection. Care should be taken to ensure the same modename is also defined at the remote system. Giving more details on other VTAM or SNA parameters is beyond the scope of this document; therefore, please refer to the related documentation.

CICS SIT definitions

As far as SNA communications are concerned, the only thing you must specify in the DFHSIT is `ISC=YES`.

1.5.3 File Control Table (FCT) definitions

Message queues are logical entities. Physically, they reside on KSDS VSAM clusters. A unique VSAM cluster may host several queues (not recommended for a production system). As previously mentioned, MQSeries is just a normal CICS application, and therefore, all general considerations for CICS files also apply to MQSeries. That is, you have to:

- Specify the VSAM clusters in the DFHFCT.
- Specify the number of index and data buffers you want to use.
- Decide whether or not they will be part of a LSRPOOL and specify the pool number.

A typical entry looks like the following:

```
MYQUEUE DFHFCT TYPE=DATASET,DATASET=MYQUEUE,                         *
               ACCMETH=VSAM,                                         *
               SERVREQ=(READ,UPDATE,ADD,BROWSE,DELETE),              *
               LOG=YES,                                              *
               RSL=PUBLIC,                                           *
               BUFND=16,BUFNI=16,STRNO=16,                           *
               RECFORM=(VARIABLE,BLOCKED),                           *
               LSRPOOL=5
```

As you can see, the number of strings is set to 16. Depending on your environment, this value may be too large or too small. We recommend you start with 16, then check the CICS statistics at the end of a busy day to see whether there were Wait on String conditions and the maximum number of strings used. This information helps you tune your MQSeries Message queues.

Also notice that we have specified LOG=YES. This is because, in a production environment, we cannot imagine running CICS without Recovery/Restart capabilities. Indeed, a DFHJCT must be defined.

For CICS TS, all definitions may be done online with the CEDA transaction. Use the definitions from the MQSeries sublibrary member MQCSD24.Z as a template for your own datasets.

1.5.4 Journal Control Table (JCT) definitions

To make sure no message is lost, it is strongly recommended that you use CICS journal logging. If you do not have a CICS system journal, you should define one. You should check with your CICS system programmer before creating or changing your JCT phase.

A parameter of the DFHJCT macro is the SUFFIX, which is a two-character value. To make this table operational, specify the two-character suffix as a SIT parameter, JCT=XX, either by recompiling your DFHSIT or with override parameters in your CICS startup JCL.

When specifying LOG=YES in DFHFCT entries, you are asking for an automatic logging to the system journal. It is used in case of emergency restart to restore file records of in-flight transactions at abend time to the value they had when the transaction abended. Automatic logging should not be confused with automatic journaling exploited by user applications. If you do not have a DFHJCT entry for the system log (also called system journal), and if you specified LOG=YES for DFHFCT entries, then you receive a AFCL abend condition when MQSeries starts.

The following is an example of a JCT definition. Buffer values may change according to the maximum VSAM record size for the MQSeries datasets.

```
         PUNCH ' CATALOG DFHJCTGG.OBJ REP=YES'
         DFHJCT TYPE=INITIAL,SUFFIX=XX
*-------------------------------------------------------------------*
*     SYSTEM JOURNALS: CICS.SYSTEM.LOG.A AND CICS.SYSTEM.LOG.B       *
*-------------------------------------------------------------------*
         DFHJCT TYPE=ENTRY,                                          *
               JFILEID=SYSTEM,                                       *
               BUFSIZE=32000,                                        *
               BUFSUV=8000,                                        *
               JOUROPT=(CRUCIAL,INPUT),                            *
               JTYPE=DISK2,                                        *
               OPEN=INITIAL,                                       *
               DEVADDR=(SYS019,SYS019)
         DFHJCT TYPE=FINAL
         END
```

JTYPE=DISK2 means that logging will alternate between two files. Therefore, do not forget to add their labels into your CICS startup and to assign logical units. For example:

```
// DLBL DFHJ01A,'CICS.SYSTEM.LOG.A',0,SD
// EXTENT SYS019,DOSRES,1,0,297984,3008
// DLBL DFHJ01B,'CICS.SYSTEM.LOG.B',0,SD
// EXTENT SYS019,DOSRES,1,0,300992,3008
// ASSGN SYS019,DISK,VOL=DOSRES,SHR   assign CICS journal DISKS
```

There is an additional step for using journals. You must first format them in a batch partition. The following JCL is an example for formatting your journals:

```
* $$ JOB JNM=CICSJOUR,CLASS=0,DISP=D,NTFY=YES
* $$ LST DEST=(host,user)
// JOB CICSJOUR    FORMAT CICS JOURNAL DATA SETS
* ---------------------------------------------------------------------*
* FORMAT SYSTEM JOURNAL: DFHJ01A                                       *
* ---------------------------------------------------------------------*
// DLBL JOURNAL,'CICS.SYSTEM.LOG.A',0,SD
// EXTENT SYS019,DOSRES,1,0,297984,3008
// ASSGN SYS019,DISK,VOL=DOSRES,SHR
// EXEC DFHJCJFP,SIZE=AUTO
/*
* ---------------------------------------------------------------------*
* FORMAT SYSTEM JOURNAL: DFHJ01B                                       *
* ---------------------------------------------------------------------*
// DLBL JOURNAL,'CICS.SYSTEM.LOG.B',0,SD
// EXTENT SYS019,DOSRES,1,0,300992,3008
// ASSGN SYS019,DISK,VOL=DOSRES,SHR
// EXEC DFHJCJFP,SIZE=AUTO
/*
/&
* $$ EOJ
```

Once everything is set up, restart your CICS system in COLD mode.

1.5.5 Destination Control Table (DCT) definitions

Transient data entries need to be defined in the CICS Destination Control Table (DCT). Make sure that the book IESZDCT is included in your DFHDCT source code, because it specifies the required entries for Language Environment/VSE (CESO and CESE).

MQSeries for VSE has several transient data queues: MQER, an intra-partition destination for logging MQSeries error messages; MQXP for registering expired messages, and MQIE for registering instrumentation events. You may either include:

```
COPY  MQCICDCT
```

(which is provided in the MQSeries sublibrary), or add the contents of MQCICDCT.Z to your DFHDCT.

You must rebuild your DCT phase and make it available to your CICS system.

1.5.6 MQSeries definitions

MQSeries for VSE, like all CICS applications, requires that you define a few programs and transactions in the CICS System Definition (CSD) file. Run the sample job MQJCSD if you run CICS 2.3 or MQJCSD24 for CICS TS. The correct use of these files is described in 1.3, “Installing under CICS/VSE® and CICS TS” on page 2.

If you do not want to use the standard group name (MQM), you have to modify most lines of the MQJCSD or MQJCSD24 job to fit your own environment. For example, if your group is MYGROUP, the definitions would be:

```
DEFINE  PROGRAM(MQBICIRH) GROUP(MYGROUP) LANGUAGE(COBOL) RSL(PUBLIC)
```

At the end of this job, the group name is added to the standard CICS group list provided by VSE, VSELIST. Here again, you may need to modify this job content either to change the list name or to add this group to other lists. It may be needed if you are running multiple CICS partitions with only one CSD file. For example:

```
ADD GROUP(MYGROUP) LIST(VSELIST)
ADD GROUP(MYGROUP) LIST(MYLIST1)
ADD GROUP(MYGROUP) LIST(MYLIST2)
```

Note: For CICS TS, dataset (FCT) entries are also defined in the MQSeries sublibrary member MQJCSD24.Z. Once again, you should refer to 1.3, “Installing under CICS/VSE® and CICS TS” on page 2 for details on the use of this file.

1.5.7 CICS system startup JCL

There is a trap that you must avoid in your CICS startup job if you wish to use TCP/IP communications with MQSeries channels. Be sure to specify the right order in the LIBDEF chain. The TCP/IP sublibrary (normally PRD1.BASE) must be specified before PRD2.SCEEBASE. This is because Language Environment/VSE is provided with a dummy phase ($EDCTCPV) needed for TCP/IP. The real phase comes with the TCP/IP product. If you do not specify the TCP/IP sublibrary first, no TCP/IP applications work under CICS. Do not forget to verify that this phase is correctly specified in the CSD. Otherwise, add it with the DFHCSDUP program (or online with CEDA), as follows:

```
DEFINE PROGRAM($EDCTCPV) GROUP(CEE) LANGUAGE(C) RSL(PUBLIC)
```

We recommend that you define your permanent LIBDEF chain as follows:

```
// LIBDEF *,SEARCH=(PRD2.CONFIG,                          X
               PRD2.COMM,                                 X
               PRD2.MQSERIES,                             X
               PRD1.BASED,                                X
               PRD1.BASE,                                 X
               PRD2.SCEEBASE,                             X
               PRD2.PROD,                                 X
               PRD2.CICSR,                                X
               PRD2.DBASE,                                X
               PRD1.MACLIBD,                              X
               PRD1.MACLIB),PERM
```

Use the temporary LIBDEF chain for your own application’s sublibraries. Note that you should replace PRD2.MQSERIES with the name of your MQSeries sublibrary if you changed this during installation.

MQSeries needs VSAM clusters for messages queues or system files (configuration, log, trace, and so forth). You must also add their labels to your CICS startup JCL (if they are not part of a catalogued procedure already specified in your current startup).

1.6 TCP/IP environment for MQSeries

MQSeries is, perhaps, your first experience with TCP/IP for VSE. Describing its installation process is beyond the scope of this document. However, a short overview may be helpful.

TCP/IP is normally installed in sublibrary PRD1.BASE. To finish the installation, you must:

- Choose a partition for running TCP/IP.

  The size of this partition depends on which applications you want to run and the number of potential clients to be connected to your servers.
- Customize a table that describes your own needs and environment. For example:
- Determine the VSE host IP address.
- Determine the connections to the outside world and the hardware used. For example, token ring, Ethernet, CTC, OSA adapters, and so forth.
- Determine which servers you want to use (FTP, Telnet, Web, and so forth).
- Determine which files (sublibraries, VSAM clusters, POWER) may be used by TCP/IP and seen by clients of these different servers.
- Determine the LU names for TN3270 clients.
- Have security parameters.
- Have tuning parameters.

  The “TCP/IP configuration table” on page 311 shows an example of a customized TCP/IP configuration.
- Add TCP/IP definitions to the CICS system definition file.

  “VTAM Telnet configuration” on page 313 shows the list of programs and transactions that should be added to the CSD file. You may notice that the program $EDCTCPV is also defined in this job step. If you have the last PTF of Language Environment/VSE applied, this line may give a return code 4, because it may have been previously defined in the CEE group. The other programs are needed for using client functions from 3270 terminals or the PING transaction for testing if a remote host is accessible.

1.7 Migrating from MQSeries 1.4 to MQSeries 2.1

MQSeries for VSE/ESA V1.4 has been out of service for several years. If you are still using V1.4 and want to migrate to V2.1.2, you probably want to save your work and use your current MQSeries object definitions and your VSAM cluster definitions. Unfortunately, the internal structure of the MQSeries files has changed, and you cannot simply reuse your configuration without migration steps. To help you migrate, sample jobs or programs are provided in the MQSeries sublibrary.

If you do not have a test system, and if the current MQSeries production sublibrary name is PRD2.MQSERIES, do not install V2.1 in this sublibrary while V1.4 is still operational. If you are installing by using the VSE Interactive Interface dialogs, you may change the default sublibrary name on the panels.

1.7.1 Migrating message queues

You *must* redefine all message queues because the message key length has changed since MQSeries 1.4. There is no way provided to migrate message queue contents. Therefore, all messages of MQSeries 1.4 still in queues must be processed before switching to the new version.

Since you are an MQSeries user, you should already have a job stream to define your VSAM clusters. If not, you may use the sample job MQJQUEUE. If you use your own definitions, change the key length from 52 to 56. Do not forget to delete old clusters first; otherwise, your definitions will fail. For example:

```
DELETE (MY.MESSAGE.QUEUE.ONE) CL NOERASE PURGE         -
   CATALOG(MYCAT.USER.CATALOG)
DEFINE CLUSTER(NAME(MY.MESSAGE.QUEUE.ONE)              -
   FILE(MYQONE)                                        -
   VOL(MYVOL1)                                         -
   RECORDS (30000 5000)                                -
   RECORDSIZE (1000 4089)                              -
   INDEXED                                             -
   KEYS(56 0)                                          -
   SHR(2))                                             -
  DATA (NAME (MY.MESSAGE.QUEUE.ONE.DATA) CISZ(4096))   -
  INDEX (NAME (MY.MESSAGE.QUEUE.ONE.INDEX) CISZ(1024)) -
  CATALOG(MYCAT)
```

1.7.2 Migrating the configuration file

The configuration file is a VSAM cluster in which all MQSeries object definitions are saved, such as:

- Channels
- Queues
- Aliases
- Global system parameters
- MQSeries messages

Migrating your current configuration file to the new one should follow this scenario:

1. Back up your current configuration to tape or to another VSAM cluster by using an IDCAMS REPRO job. This step is optional but strongly recommended. For example:

   ```
   // JOB BACKUP MQSERIES CONFIGURATION FILE
   // DLBL MQFCNFG,'MQSERIES.MQFCNFG',,VSAM,CAT=?cat-name?
   // DLBL BKPCNFG,'MQSERIES.MQFCNFG.BACKUP',,VSAM,CAT=?cat-name?
   // EXEC IDCAMS,SIZE=AUTO
      REPRO INFILE(MQCNFG) OUTFILE(BKPCNFG) REPLACE
   /*
   /&
   ```

   Note that MQSERIES.MQFCNFG is the default file ID for the MQSeries configuration file. You may have changed this during your initial installation.

2. Run the sample job MQJMIGR1 found in the MQSeries sublibrary. This job does the following:

   – Defines a work VSAM file named MQSERIES.MQOCNFG:

   ```
   /*                                                            */
   /*     VERIFY VSAM FILE, CANCEL THE JOB IF IT IS IN USE       */
   /*                                                            */
             VERIFY FILE(OLDCNFG)
   ```

```
          IF MAXCC > 0 THEN CANCEL
/*                                                        */
/*       DELETE AND DEFINE THE WORK FILE                  */
/*                                                        */
          DELETE  (MQSERIES.MQOCNFG)                     -
                 CL ERASE PURGE CAT(?cat-name?)
          SET MAXCC = 0
          DEFINE CLUSTER                                 -
                   (NAME (MQSERIES.MQOCNFG)              -
                    RECORDS (50 10)                      -
                    RECORDSIZE (2048 2048)               -
                    VOLUMES (?vol-id?)                   -
                    KEYS    (100 0)                      -
                    SHR     (2)                          -
                    INDEXED)                             -
                 DATA                                    -
                   (NAME (MQSERIES.MQOCNFG.DATA) CISZ(4096)) -
                 INDEX                                   -
                   (NAME (MQSERIES.MQOCNFG.INDEX) CISZ(512)) -
                    CAT (?cat-name?)
```

– Executes the program MQPCONFG that reads the old configuration file records, formats them to the new structure, and writes them to the work file:

```
// IF $MRC > 0 THEN
// GOTO NOPROC
// LIBDEF PHASE,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPCONFG,SIZE=AUTO
/*
/. NOPROC
```

At this point, you should continue as though you were installing without migrating. That is:

a. Define all VSAM clusters for MQSeries, including the configuration file.
b. Start the MQSU CICS transaction to load system objects to the configuration file.

Please refer to 1.3, “Installing under CICS/VSE® and CICS TS” on page 2 for more details.

3. Now, the only thing left to do is to merge your own object definitions to the new configuration file. The sample job MQJMIGR2 does this for you. It reads the work file records created in step 2 and copies them to the configuration file (via an IDCAMS REPRO command). If merging was successful, then the work file is deleted:

```
// DLBL MQFCNFG,'MQSERIES.MQFCNFG',,VSAM,CAT=?cat-name?
// DLBL MQOCNFG,'MQSERIES.MQOCNFG',,VSAM,CAT=?cat-name?
// EXEC IDCAMS,SIZE=AUTO
 /*                                                       */
 /*    VERIFY VSAM FILES, CANCEL THE JOB IF THEY ARE IN USE    */
 /*                                                       */
          VERIFY FILE(MQFCNFG)
          VERIFY FILE(MQOCNFG)
          IF MAXCC > 0 THEN CANCEL
          REPRO INFILE(MQOCNFG) OUTFILE(MQFCNFG) REPLACE
 /*                                                       */
 /*             DON'T ERASE THE WORK FILE IF REPRO FAILED      */
 /*                                                       */
          IF MAXCC > 0 THEN CANCEL
          DELETE (MQSERIES.MQOCNFG) CL NOERASE PURGE  -
                  CATALOG(?cat-name?)
```

Note: Most sample jobs may need changes to fit your own environment before you submit them, especially volumes and VSAM catalog names.

1.7.3 Migrating CICS CSD definitions

There are more programs and transactions CSD entries with version 2.1 than version 1.4. Therefore, you should run the job MQJCSD (or MQJCSD24 for CICS TS installations) as indicated in 1.3, “Installing under CICS/VSE® and CICS TS” on page 2. If you have your CICS system running, you may use the CEDA INSTALL function to install your MQSeries group for new programs and transactions. However, because so many other things have changed (message queues, configuration files, and sublibrary content), we strongly recommend that you shut down CICS normally and restart in COLD mode.

1.7.4 Migrating MQSeries application programs

If you are migrating from MQSeries for VSE V2.1.1, there is no need to relink your applications. If you are migrating from V1.4 or 2.1.0, you must relink your applications. This is because various improvements in the MQSeries for VSE product have led to changes to the MQI objects that are called from MQ applications. Unless your applications are relinked, they will not include these improvements. In addition, MQ internal control blocks may have changed and the old MQI objects may have incorrect mappings to these internal data structures. A failure to relink may result in unpredictable results.

1.7.5 Migrating from SNA to TCP/IP

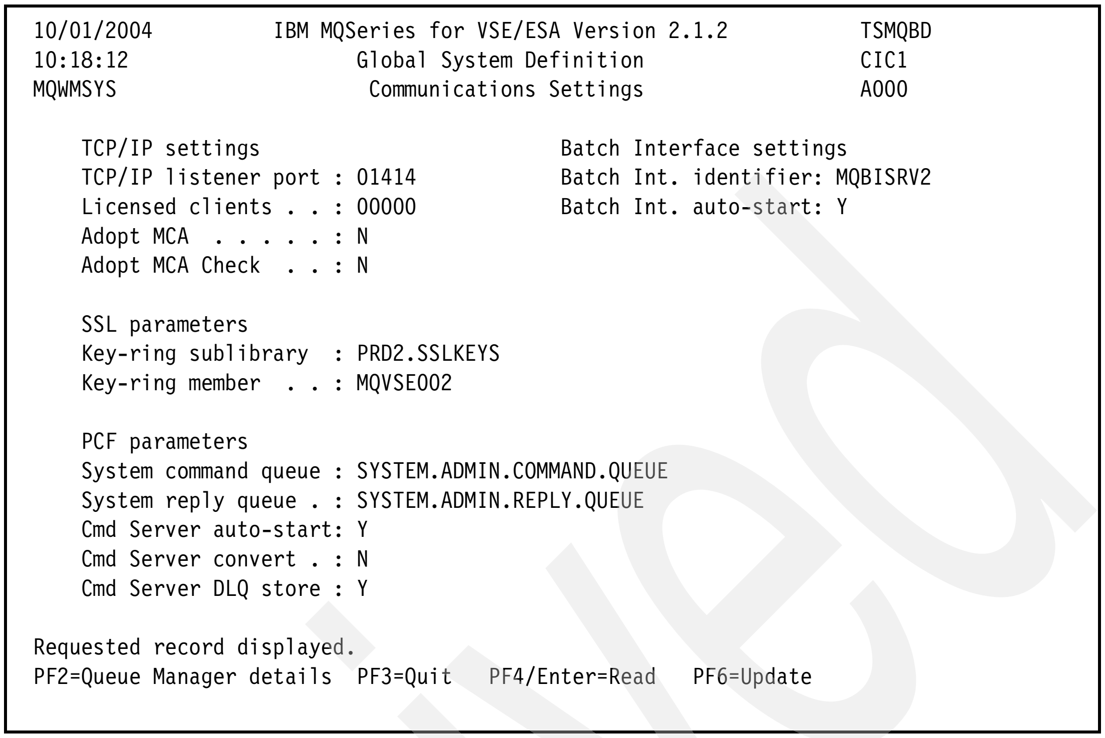
One of the most important improvements of version 2.1 is the capability to connect to a TCP/IP network. You may want to use TCP/IP instead of VTAM to exchange messages between MQSeries queue managers.

For LU 6.2 communications, you specified sessions and connections in the CICS system definition file. These are not needed with TCP/IP. It is only at the global system and channel definition levels that you specify relevant information.

Global system definitions

Communication settings for the queue manager are accessible using the MQMT transaction. Option 1.1 displays the queue manager, or global system definition. From here, you can use **PF9** to access the queue manager's communication settings.

The only setting of immediate importance at this point is the TCP/IP port value. The default is 1414. If you change this value, queue managers on other systems must be aware of the value you have chosen. However, if your VSE system hosts multiple CICS partitions each running an MQSeries queue manager, you *must* specify different values. This is because a port value represents a unique TCP/IP application.



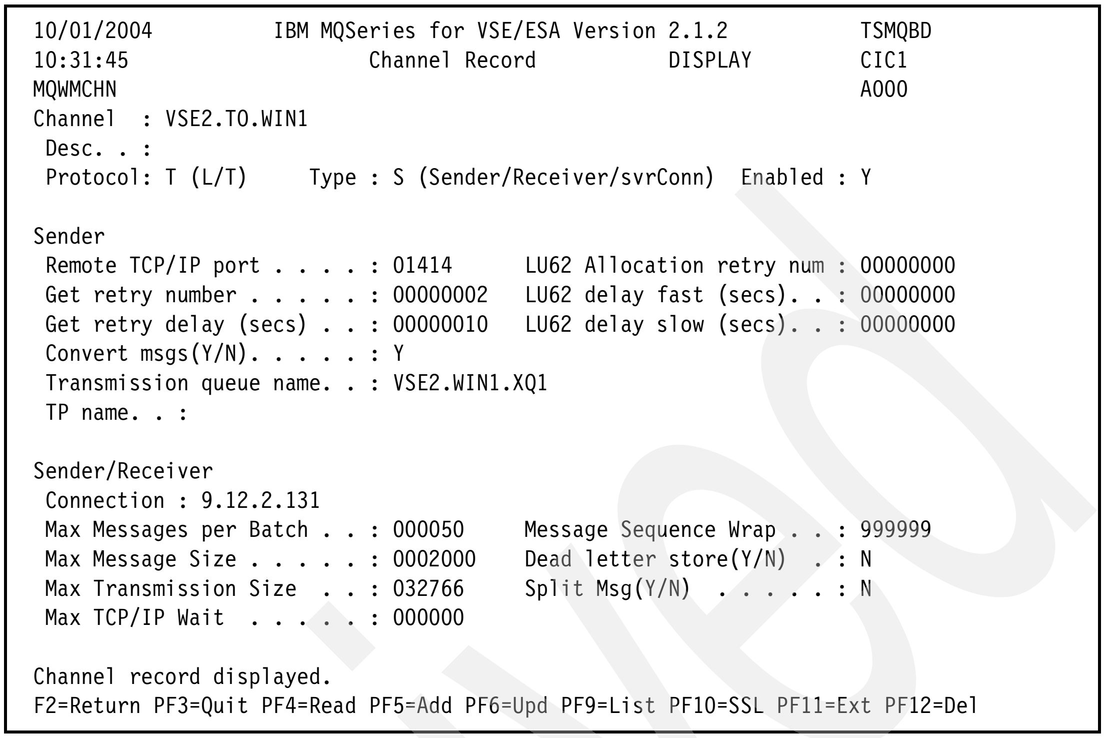
```
10/01/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:18:12                  Global System Definition                 CIC1
MQWMSYS                    Communications Settings                 A000

    TCP/IP settings                        Batch Interface settings
    TCP/IP listener port : 01414           Batch Int. identifier: MQBISRV2
    Licensed clients . . : 00000           Batch Int. auto-start: Y
    Adopt MCA  . . . . . : N
    Adopt MCA Check  . . : N

    SSL parameters
    Key-ring sublibrary  : PRD2.SSLKEYS
    Key-ring member  . . : MQVSE002

    PCF parameters
    System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
    System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
    Cmd Server auto-start: Y
    Cmd Server convert . : N
    Cmd Server DLQ store : Y

Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 1-5 Queue manager communication settings

Channel definitions

For TCP/IP connections, you need to fill in the following specific fields of the Channel definitions panel:

- Protocol
  Set this field to `T` (for TCP/IP); it was `L` (for LU 6.2).
- Remote TCP/IP Port
  Specify the TCP/IP port number of the remote queue manager (probably 1414). This value has meaning only for sender channels. For receiver or client channels, you may set the port number to `0`.
- Type
  Keep what you had before: `S` for a sender channel, `R` for a receiver channel, or `C` for client (SVRCONN).
- Connection
  Specify the IP address of the remote queue manager (Sender channels only).
- TP Name
  This field is a typical SNA concept and has no meaning in a TCP/IP environment; therefore, leave it blank.



```
10/01/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:31:45                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A000
Channel  : VSE2.TO.WIN1
 Desc. . :
 Protocol: T (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 01414      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000002   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : Y
 Transmission queue name. . : VSE2.WIN1.XQ1
 TP name. . :

Sender/Receiver
 Connection : 9.12.2.131
 Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0002000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 1-6 Channel definition screen

Configuration

Typically, an MQSeries environment involves multiple systems on different platforms. For example, you may want to transfer messages from your VSE system to UNIX or Windows XP systems in a peer-to-peer manner (that is, from queue manager to queue manager), or more simply, transfer messages from a workstation or from a VM/CMS user in client/server mode. In such cases, configuring MQSeries is a complex task.

Consequently, configuring your MQSeries system should follow a long-term design process that defines the architecture of your entire MQSeries environment. This implies the following:

- Naming conventions for:
  - Queue manager
  - Message queues
- Choices of communication protocols:
  - TCP/IP versus LU6.2
  - Client/server mode versus peer-to-peer
  - Communication parameters, for example, message sizes and retries
- Applications options:
  - Relationships between programs and queues
  - Triggering
  - Invalid messages handling
  - Code page translation

The goal of this chapter is not to describe these design tasks, but to provide you with additional information about how to configure MQSeries objects to cover different cases. However, it is not a replacement for the <cit>MQSeries for VSE/ESA V2 R1.0 System Management Guide, GC34-5364, which we recommend you read first.

MQSeries objects can be created and modified using the master terminal transaction (MQMT), Programmable Command Formats (PCF), or MQSeries Commands (MQSC). Both PCF and MQSC require certain queues to be defined to the queue manager before these facilities can be used. Consequently, it is necessary to begin defining your MQSeries configuration using the master terminal transaction.

MQMT provides a main menu and the selection of options. Option 1 allows you to configure MQSeries objects.

Configurable MQSeries objects include:

- Global System Definition
- Queues
- Channels

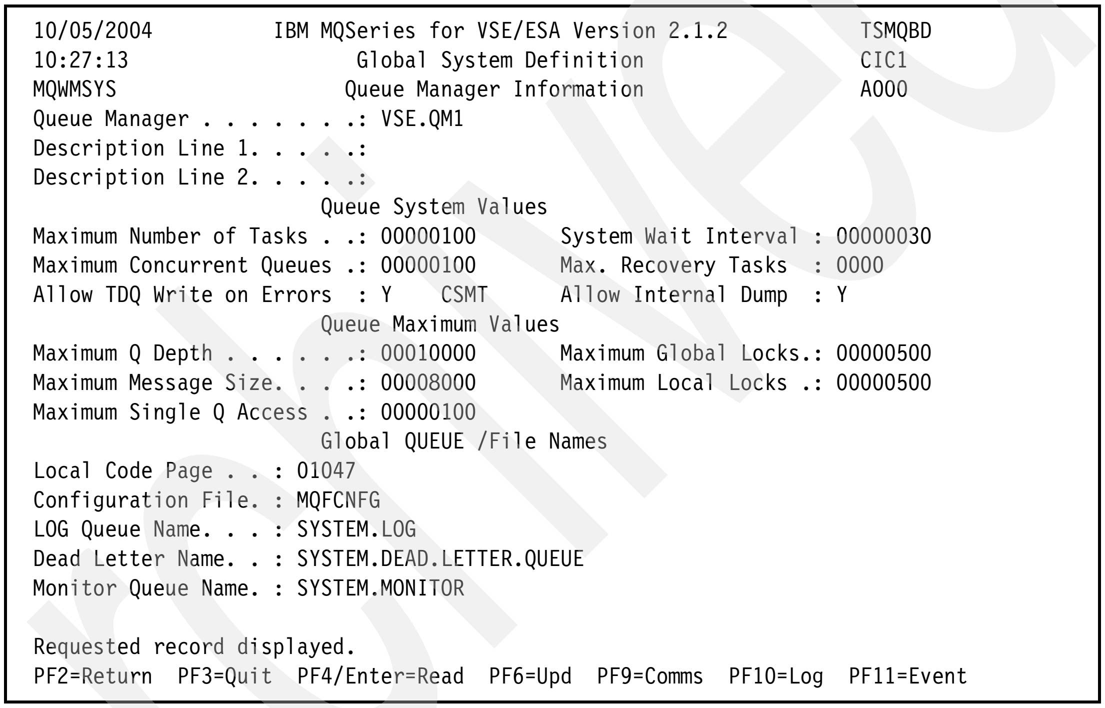
Each of these is described in the following sections.

2.1 Global System Definition (GSD)

The first thing you have to do when configuring is to specify global parameters for MQSeries. The term *global* does not apply to the entire VSE system even though you may have multiple MQSeries applications in different CICS partitions. It is related to only one CICS system (and, therefore, one queue manager).

You can maintain the Global System Definition (GSD) via MQMT option 1.1 (Maintenance Options: Global System Definition). This option opens the panel shown in Figure 2-1.

You will reuse many fields from this panel when you configure queues. However, there are differences. With the GSD, you generally specify limits or thresholds that you may not go over at the queue level. In other words, such parameters set in the GSD are intended to validate values set for queue definitions.



```
10/05/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:27:13                  Global System Definition                 CIC1
MQWMSYS                  Queue Manager Information                 A000
Queue Manager . . . . . . .: VSE.QM1
Description Line 1. . . . .:
Description Line 2. . . . .:
                        Queue System Values
Maximum Number of Tasks . .: 00000100       System Wait Interval : 00000030
Maximum Concurrent Queues .: 00000100       Max. Recovery Tasks  : 0000
Allow TDQ Write on Errors  : Y    CSMT      Allow Internal Dump  : Y
                        Queue Maximum Values
Maximum Q Depth . . . . . .: 00010000       Maximum Global Locks.: 00000500
Maximum Message Size. . . .: 00008000       Maximum Local Locks .: 00000500
Maximum Single Q Access . .: 00000100
                        Global QUEUE /File Names
Local Code Page . . : 01047
Configuration File. : MQFCNFG
LOG Queue Name. . . : SYSTEM.LOG
Dead Letter Name. . : SYSTEM.DEAD.LETTER.QUEUE
Monitor Queue Name. : SYSTEM.MONITOR

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF6=Upd  PF9=Comms  PF10=Log  PF11=Event
```

Figure 2-1 Global System Definition screen

The following points should be taken into consideration when choosing values for GSD parameters:

- Queue Manager

  This name should be unique across your MQSeries environment. In fact, every time an application issues an MQPUT or an MQGET, the queue name is logically concatenated with the queue manager name to form a unique identifier. For example, if a queue manager is MY.Q.MGR and a queue name is MY.QUEUE, the fully-qualified queue name is MY.Q.MGR.MY.QUEUE.

  The immediate implication of this is that you may have the same name for a queue if it is handled by different queue managers. Practically, this offers you the capability to have the same applications running under different queue managers, where the only difference is the queue manager name used as a variable (through an alias for example).

- Maximum Number of Tasks

  The maximum value you may specify is `1000`.

  When an MQSeries application starts, the first thing it has to do is identify itself to the queue manager. The MQI call MQCONN does this. This parameter, therefore, represents the maximum of simultaneous tasks being connected to the queue manager. It is not necessary to have a very high value. For example, when running CICS/VSE, with AMXT or CMXT = 50, setting this field equal to 100 is useless, because it is not possible to have so many tasks running at the same time.

- Maximum Concurrent Queues

  The maximum value you may specify is `1000`.

  This number is the maximum number of message queues you can access at the same time (all being opened) for your entire system. It is *not* the total queues that a single task may open.

- Max. Recovery Tasks

  This applies only to dual queues. If this parameter value is greater than zero, when a dual queue is out-of-sync with its primary queue, it is put in *Recovery status*. Then, a recovery transaction is initiated that tries to *repair* the dual queue (that is to re-synchronize it with the primary queue). If this parameter value is zero, the dual queue stays in RECOVERY state forever and messages are not replicated.

- Maximum Q Depth

  To avoid filling up VSAM clusters when messages are not processed, you specify the maximum number of messages that you might find in any queue in *written* state. Messages are in written state when they have not yet been read by an MQGET call. When an MQGET has been issued for a message, it becomes logically deleted. When a message is browsed, it remains in written state.

  When the maximum Q depth is reached, further MQPUTs are rejected with reason code MQRC-Q-FULL (2053).

  However, there is a trap here. This value is only used to validate the maximum Q depth you specify when you configure a queue. If, after setting the maximum Q depth at the queue level, you change the GSD maximum Q depth to a lower value, it will be accepted, but the queue itself uses its own maximum queue depth regardless. Consequently, this parameter is for validation purposes only.

- Maximum Message Size

  Once again, this parameter is used to validate queue level definitions. The maximum message size for a queue must be lower or equal to the value you set in the GSD.

- Maximum Single Q Access

  In an application program, to access a particular queue, an MQOPEN (or MQPUT1) call against this queue is required. This value represents the maximum number of open queues you can have concurrently for your entire system. Lower values may be specified at the queue definition level.

- Maximum Global Locks

  This is the upper limit for the equivalent parameter that you define for queues. Please refer to 2.2, “Configuring queues” on page 40.

- Maximum Local Locks

  This is the upper limit for the equivalent parameter that you define for queues. Please refer to the queue definition section.

- Local Code Page

  In this field, you specify the code page of the queue manager.

  A code page is a table of hexadecimal values representing alphabetic, numeric, or graphic characters. Usually, it depends on countries. Even in the same country, if two platforms are exchanging messages, MQSeries needs to know the sender code page and the receiver code page. If they are different, then a translation is necessary. This is the case, for example, when sending messages from a VSE 8-bit coded (EBCDIC) to a UNIX host 8-bit coded (ASCII).

  The local code page is used by MQSeries for VSE 2.1 in two different ways. It is used for queue manager-to-queue manager communication and client-to-server communication.

  In the queue manager to queue manager case, the local code page is immediately relevant to remote MQSeries systems. Remote queue managers need to know the local code page of other remote systems so that they can translate control information exchanged during message transmission.

  In addition, MQSeries channels can be configured to indicate whether or not message data should be translated before it is sent to a remote queue manager. In this case, the queue manager must know the local and remote code pages so that the message data can be translated correctly.

  In the client-to-server case, the queue manager uses the local (server) and remote (client) code pages to translate control information associated with client requests. Translation in this case, and in the case of message data conversion, MQSeries for VSE uses Language Environment/VSE code page translation services.

  Language Environment/VSE provides phases in the PRD2.SCEEBASE sublibrary for such translations. Most of the default translation phases provided with Language Environment/VSE allow conversion between EBCDIC code pages. While this document is being written, the only EBCDIC to ASCII conversion phases shipped with Language Environment/VSE are EBCDIC-1047 to ASCII-850 (or vice/versa). This clearly means that if, for example, you have a Windows XP server installed in the French language with code page 437, you *must* either change this value on your Windows XP system or compile translation phases on the VSE side. To change the value for a Windows XP client, you can set the environment variable MQCCSID. For example:

  ```
  SET MQCCSID=850
  ```

  For queue manager to queue manager, you can change the queue manager's CCSID using the **ALTER** command. For example:

  ```
  runmqsc YOUR.QM
  ALTER QMGR CCSID(850)
  END
  ```

  You only need to worry about code page conversion if you plan to run non-VSE clients, to use the Convert Msg option on your sender channels, or to use the MQGMO_CONVERT option with MQGET in your applications.

  Table 2-1 on page 32 shows the default Language Environment/VSE supported code pages. The last column is used to standardize VSE phase names, as explained in more detail in “Creating code page conversion tables” on page 120.

Table 2-1 Language Environment/VSE default-supported code pages

Codeset	Primary Country or Territory	2-byte CC
EBCDIC Codesets		
IBM-037	USA, Canada	EA
IBM-273	Germany, Austria	EB
IBM-274	Belgium	EC
IBM-275	Brazil	ED
IBM-277	Denmark, Norway	EE
IBM-278	Finland, Sweden	EF
IBM-280	Italy	EG
IBM-281	Japan (Latin-1)	EH
IBM-282	Portugal	EI
IBM-284	Spain, Latin America	EJ
IBM-285	United Kingdom	EK
IBM-290	Japan (Katakana)	EL
IBM-297	France	EM
IBM-300	Japanese DBCS	EN
IBM-500	International	EO
IBM-838	Thailand	EP
IBM-870	ROECE Latin and Yugoslav	EQ
IBM-871	Iceland	ER
IBM-875	Greece	ES
IBM-880	Cyrillic	ET
IBM-930	Japan Katakana Extended (combined with DBCS)	EU
IBM-939	Japan (Latin) Extended (combined with DBCS)	EV
IBM-1026	Turkey	EW
IBM-1027	Japan (Latin) Extended	EX
IBM-1047	Latin-1/Open Systems	EY
ASCII Codesets		
IBM-850	IBM PC - International	AA
IBM-932	IBM PC - Japanese	AB
IBM-eucJP	Japanese	AC
ISO8859 Codesets		
ISO8859-1	ISO Standard ASCII	I1
ISO8859-7	ISO ASCII - Greece	I7
ISO8859-9	ISO ASCII - Turkey	I9

– Dead Letter Name

When a queue manager receives a message for an unknown queue or unknown queue manager (because it can act as a router), it puts the message into a special queue called dead letter queue. It is your responsibility to handle the content of this queue, for example, by triggering a CICS transaction, or having an endless CICS program that scans this queue from time to time.

Note: A sample program to handle the Dead Letter Queue (DLQ) can be downloaded from the following Web site:

http://www.ibm.com/software/ts/mqseries/platforms/vseesa/

The sample program is available as a freeware "as-is" SupportPac™.

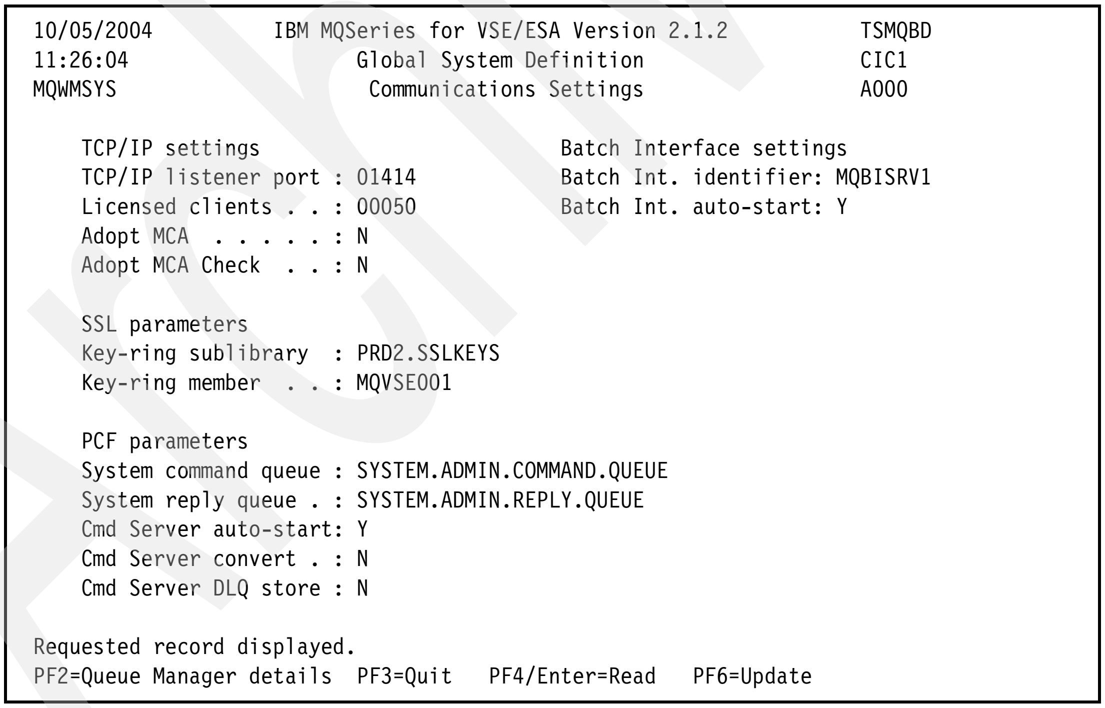
Specifying the dead letter queue name in this field does not exempt you from defining the queue with the configuration queue dialog. This remark applies also to the system log and monitor queues. In other words, the name you supply for the dead letter queue must also be defined as a queue.

For details on global system definition parameters not discussed here, refer to the *MQSeries for VSE System Management Guide*, GC34-5364.

The Global System Definition includes communications, log and trace, and event settings. These queue manager settings are accessible using PF keys from MQMT option 1.1, and are described in the following sections.

2.1.1 Communication settings

The queue manager's communications settings screen is accessible via **PF9** from MQMT option 1.1.



```
 10/05/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
 11:26:04                   Global System Definition                  CIC1
 MQWMSYS                     Communications Settings                  A000

     TCP/IP settings                         Batch Interface settings
     TCP/IP listener port : 01414            Batch Int. identifier: MQBISRV1
     Licensed clients . . : 00050            Batch Int. auto-start: Y
     Adopt MCA  . . . . . : N
     Adopt MCA Check  . . : N

     SSL parameters
     Key-ring sublibrary  : PRD2.SSLKEYS
     Key-ring member  . . : MQVSE001

     PCF parameters
     System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
     System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
     Cmd Server auto-start: Y
     Cmd Server convert . : N
     Cmd Server DLQ store : N

 Requested record displayed.
 PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 2-2 Queue manager communications settings

The queue manager's communications settings are logically grouped into four categories:

- TCP/IP settings
- Batch Interface settings
- SSL parameters
- PCF parameters

TCP/IP settings

The TCP/IP settings affect how the queue manager behaves under certain circumstances involving TCP/IP. If you don't plan to use TCP/IP, these settings are irrelevant.

- TCP/IP listener port

  The TCP/IP listener port indicates the port number the queue manager should use for incoming TCP/IP connection requests. These are requests from remote queue managers and clients. The default value is 1414, but any valid port number can be used, taking care not to use a port number that is reserved for another application.

  Remote queue managers and clients must specify the port number if they intend to establish a TCP/IP connection with your VSE queue manager. Each port number uniquely identifies a queue manager at an IP address for TCP/IP communication. If you are running multiple queue managers on a single VSE host, each must have its own unique port number. The same port number can be used on different hosts.

- Licensed clients

  The *Licensed clients* parameter was intended for special MQSeries license considerations where queue manager access by client applications would be restricted. At the time of publication, this restriction had not been implemented. Consequently, the *Licensed clients* parameter can be set to any numeric value without affecting MQSeries for VSE operation.

- Adopt MCA

  The *Adopt MCA* parameter provides a mechanism where by the queue manager can reactivate a TCP/IP channel that is deemed to have stalled. Normally, a request to start a channel that is already active results in the request being rejected. In some cases, however, a channel may have stalled (that is, the TCP/IP connection has been severed), leaving the channel in limbo.

  By setting this parameter to `Y`, the queue manager can terminate an active channel, when a subsequent start request is received, and start the channel anew. This service is not available for LU6.2 channels.

- Adopt MCA Check

  This parameter works in conjunction with the *Adopt MCA* parameter. If the Adopt MCA feature is not active, this parameter is ignored. If the Adopt MCA feature is active, and this parameter is set to `Y`, the queue manager checks that a channel start request for an active channel has been received from the same IP address as the active channel before restarting the channel. This is a precaution against terminating and restarting an active channel accidentally.

Batch Interface settings

The Batch Interface settings affect how the queue manager manages the batch interface service. The batch interface is a facility where by MQ applications running in batch can use the MQSeries resources controlled by CICS.

- Batch Int. Identifier

  The batch interface identifier is a unique name that identifies the queue manager for batch applications. The name can be 1-8 characters and defaults to MQBISERV. If you plan to run multiple queue managers in multiple CICS regions on your VSE system, each queue manager must have its own unique batch interface identifier. The same identifier can be used on different VSE systems.

  When a batch program runs, if you want to connect to a queue manager other than one with the default ID of MQBISERV then you require a // SETPARM card for MQBISRV variable in its JCL. For example:

  ```
  // SETPARM MQBISRV=bintid
  ```

where `bintid` is the batch interface identifier of the queue manager. The batch JCL must also have implicit or explicit LIBDEF SEARCH access to the Language Environment sublibrary PRD2.SCEEBASE for it to be able to extract the MQBISRV value.

MQI requests made by the batch program affect those resources (such as queues) owned by the identified queue manager.

- Batch int. auto-start

  The batch interface auto-start parameter indicates whether or not the queue manager should start the batch interface when MQ itself is started. The batch interface is a long-running transaction (MQBI).

  If this parameter is set to `Y`, the MQBI transaction is started automatically at MQSeries startup and stopped automatically at shutdown. Alternatively, the batch interface can be started manually at any time after MQSeries has started.

  The batch interface is described in greater detail in 8.2, “Batch application programs” on page 151.

SSL parameters

The SSL parameters affect how the queue manager handles SSL-enabled channels. A channel can be configured to use secure sockets layer (SSL) services. Such a channel is considered SSL-enabled.

- Key-ring sublibrary

  Before a channel can be configured to use SSL services, the SSL product must be installed on your VSE system. The installation process includes the definition of an SSL key-ring sublibrary, where private keys and certificates are stored. The key-ring sublibrary parameter should name this sublibrary.

- Key-ring member

  MQSeries for VSE uses a single PKI certificate when processing SSL-enabled channels. The key-ring member should identify the member name of a valid certificate residing in the key-ring sublibrary.

PCF parameters

The PCF parameters affect how the queue manager handles support for Programmable Command Format (PCF) messages. PCF messages are requests to the queue manager to create, delete, or manipulate objects owned by the queue manager.

- System command queue

  The system command queue is the target queue for PCF request messages. Application programs can put messages to this queue directly, or put messages to it via a remote queue definition on a remote system. The default name is SYSTEM.ADMIN.COMMAND.QUEUE.

  Whatever name is defined here, the actual queue must also be created using the queue definition dialog (MQMT option 1.2).

- System reply queue

  The system reply queue is a special queue used by the MQSeries Command (MQSC) facility. The MQSC utility program, MQPMQSC, is a batch program that communicates with the queue manager using PCF Escape messages. PCF Escape messages are verb-based text commands, for example, DEFINE QLOCAL.

  Because the MQSC utility uses PCF to send verb-based requests to the queue manager, it requires a reply queue for the results of its requests. The system reply queue identifies the queue used for this purpose. The default is SYSTEM.ADMIN.REPLY.QUEUE, but any valid queue name can be used.

Once again, the queue identified by the *System reply queue* parameter must also be defined to the queue manager.

- Cmd Server auto-start

  The system command queue is monitored by a long running transaction called the Command Server (transaction MQCS). If this transaction is not running, the PCF facility is disabled.

  By setting the command server auto-start parameter to `Y`, the queue manager will automatically start the command server when MQSeries is started and stop it at shutdown. Alternatively, the command server can be started manually by running the MQSC transaction.

- Cmd Server convert

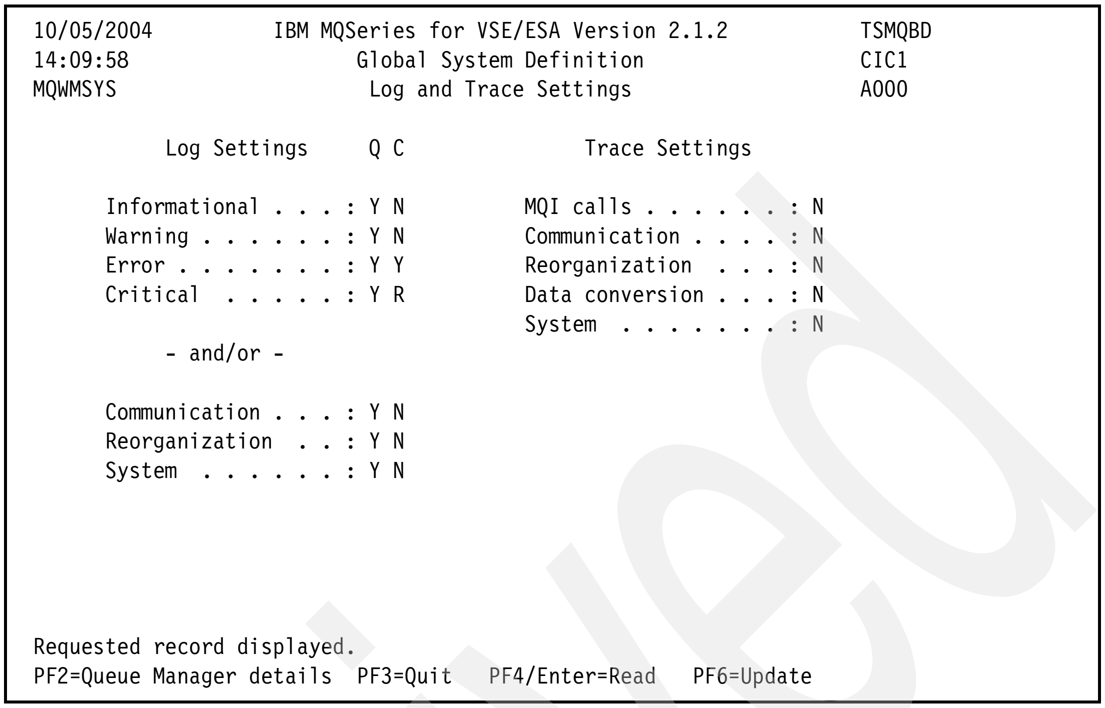
  Messages placed on the system command queue may have been placed there by a remote system, potentially running with a different code page (for example, an ASCII code page). By setting the command server convert parameter to `Y`, the command server will automatically convert messages that arrive on the system command queue before attempting to process them.

- Cmd Server DLQ store

  When a command request arrives on the system command queue that cannot be processed, or its reply message cannot be delivered, the relevant message is discarded. However, setting the command server DLQ store parameter to `Y` will cause the command server to place the relevant message on the dead letter queue.

2.1.2 Log and Trace settings

The queue manager's log and trace settings screen is accessible via **PF10** from MQMT option 1.1.



```
10/05/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
14:09:58                  Global System Definition                 CIC1
MQWMSYS                    Log and Trace Settings                  A000

           Log Settings     Q C             Trace Settings

      Informational . . . : Y N         MQI calls . . . . . . : N
      Warning . . . . . . : Y N         Communication . . . . : N
      Error . . . . . . . : Y Y         Reorganization  . . . : N
      Critical  . . . . . : Y R         Data conversion . . . : N
                                        System  . . . . . . . : N
           - and/or -

      Communication . . . : Y N
      Reorganization  . . : Y N
      System  . . . . . . : Y N




Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 2-3 Queue manager log and trace settings

The queue manager's log and trace settings are logically grouped into two categories:

- Log settings
- Trace settings

Log settings

The queue manager's log settings affect which messages are written to the system log, and optionally, also written to the VSE console.

Messages can be isolated by message severity, or optionally, by MQSeries facility. All messages have a severity of informational, warning, error or critical. Each message is also generated by one of the queue manager's general facilities.

By setting the relevant message severity or facility parameter to `Y` under the column labelled "Q", you specify that those messages should be written to the system log queue. If you set the corresponding parameter under the column labelled "C," the message will also be written to the VSE console.

Messages with a severity of “error” or “critical” can be sent to the console requiring an operator's reply. This is done by setting the relevant message severity or facility to "R" under the console column.

When configuring your queue manager log settings, you should limit the number of messages sent to the VSE console, and perhaps only request an operator's reply to critical messages (if at all). For this reason, the following configuration is recommended:

```
Log Settings         Q C

Informational . . . : Y N
Warning . . . . . . : Y N
Error . . . . . . . : Y Y
Critical  . . . . . : Y R
```

This configuration will send all messages to the system log queue, but only error and critical messages to the console. Additionally, only critical messages will require an operator's reply.

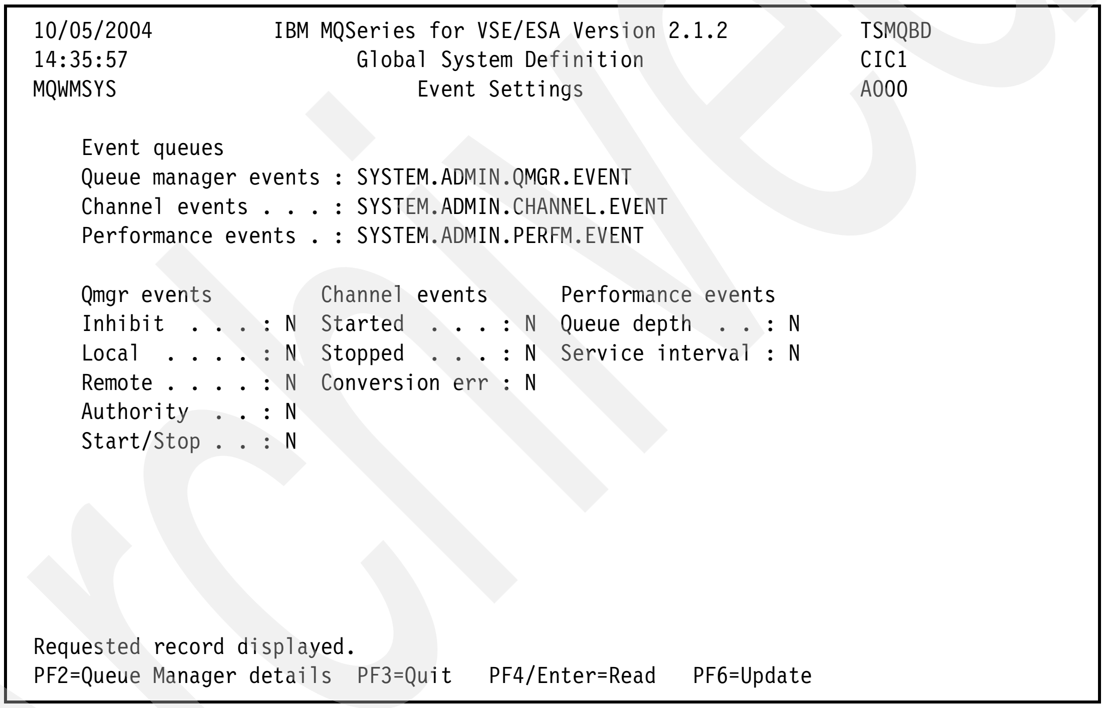
Trace settings

The queue manager's trace settings affect the generation of diagnostic trace entries in the CICS auxiliary trace. These diagnostic entries are for problem determination, and do not affect MQSeries operation.

The diagnostic entries are not documented, and are meaningful only to the IBM service group. They also generate considerable overhead, so your trace settings should always be set to `N` unless you are working with the IBM service group to resolve a problem.

2.1.3 Event settings

The queue manager's event settings screen is accessible via **PF11** from MQMT option 1.1.



```
10/05/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
14:35:57                  Global System Definition                CIC1
MQWMSYS                        Event Settings                     A000

    Event queues
    Queue manager events : SYSTEM.ADMIN.QMGR.EVENT
    Channel events . . . : SYSTEM.ADMIN.CHANNEL.EVENT
    Performance events . : SYSTEM.ADMIN.PERFM.EVENT

    Qmgr events         Channel events      Performance events
    Inhibit  . . . : N  Started  . . . : N  Queue depth  . . : N
    Local  . . . . : N  Stopped  . . . : N  Service interval : N
    Remote . . . . : N  Conversion err : N
    Authority  . . : N
    Start/Stop . . : N




Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 2-4 Queue manager event settings

The queue manager's event settings are logically grouped into four categories:

```
Event queues
Queue manager events
Channel events
Performance events
```

Event generation introduces processing overhead during MQSeries operation. Events should not be enabled unless you have a specific need to take action, or generate a report, when a specific event occurs.

Event queues

There are three event queues. Each queue is the target of a certain type of event messages. Event types include queue manager, channel and performance events.

If event generation is enabled for a specific event, and that event occurs, an event message is written to the appropriate queue.

Default event queue names are as follows:

SYSTEM.ADMIN.QMGR.EVENT– Queue manager events
SYSTEM.ADMIN.CHANNEL.EVENT– Channel events
SYSTEM.ADMIN.PERFM.EVENT– Performance events

If event generation is enabled for any of these categories, then the corresponding queue must be defined using MQMT option 1.2.

Queue manager events

Specific queue manager events can be enabled by setting the appropriate queue manager event parameter to Y. Queue manager events include:

- Inhibit

  An inhibit event occurs when an attempt to put or get a message to or from a queue is rejected because the relevant operation is inhibited.
- Local

  A local event occurs when access to a local MQSeries object fails. For example: an attempt is made to open a queue that is not defined to the local queue manager.
- Remote

  A remote event occurs when access to an MQSeries object involved with remote queue fails. For example: a transmission queue is not properly defined.
- Authority

  An authority event occurs when an attempt is made to access an MQSeries object, but the user making the attempt is not authorized.
- Start/Stop

  A start/stop event occurs when the queue manager is started or stopped.

Channel events

Specific channel events can be enabled by setting the appropriate channel event parameter to Y. Channel events include:

- Started

  This event occurs when a channel starts.
- Stopped

  This event occurs when a channel stops.
- Conversion error

  This event occurs when a channel encounters a data conversion error when taking messages from a local transmissions queue.

Performance events

There are two types of performance event. These can be enabled by setting the appropriate performance event parameter to Y. Performance events include:

- Queue depth

  A queue depth event occurs when a queue is full, or its depth hits a high or low water mark defined for the queue. High and low water marks are specified as a percentage of maximum queue depth as part of the queue definition. For example, you can request a queue depth event is generated when the queue is 90% full.

- Service interval

  Service interval events occur when a queue is deemed to have been inactive for a longer than acceptable period. The acceptable period is configurable as part of the queue definition.

Performance events are associated with individual queues, and as such, can be switched on or off for each individual queue. Whether performance events are generated for a particular queue is determined by the queue's definitions. However, no performance events are generated for a queue if performance events are disabled at the queue manager level.

2.2 Configuring queues

Basically there are only two types of queues:

- Local queues

  From/to which application programs read/write messages. The physical support of these queues is VSAM clusters defined in your CICS DFHFCT or CSD.

- Remote queues

  To which applications may only logically write messages. This queue is a local queue on a remote queue manager. The physical organization of the file that hosts this queue depends on the remote queue manager platform.

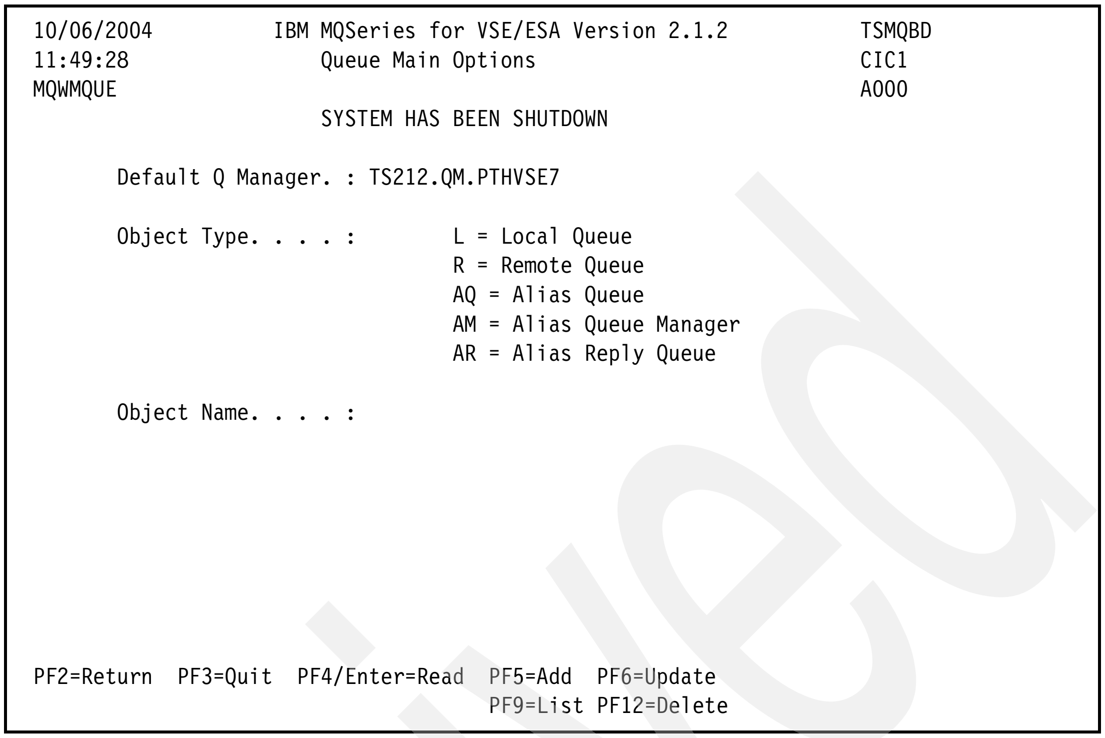
MQSeries works in a *store and forward* way to make sure messages are never lost. Therefore, before sending messages to a remote queue, it first writes them to a special local queue called a *transmission queue*.

Hard-coding queue names in programs makes them very dependent on systems or the queue managers they are running under. A better way is to use logical names in the program that are configured to identify physical queue names in MQSeries. In the MQSeries terminology, these logical names are called *aliases*.

When you select Queue Definitions from the Configuration menu panel, you may specify characteristics for these four types of objects.

2.2.1 Defining local queues

The screen shown in Figure 2-5 on page 41 is an example of the initial panel for defining a queue. It is available using MQMT option 1.2.

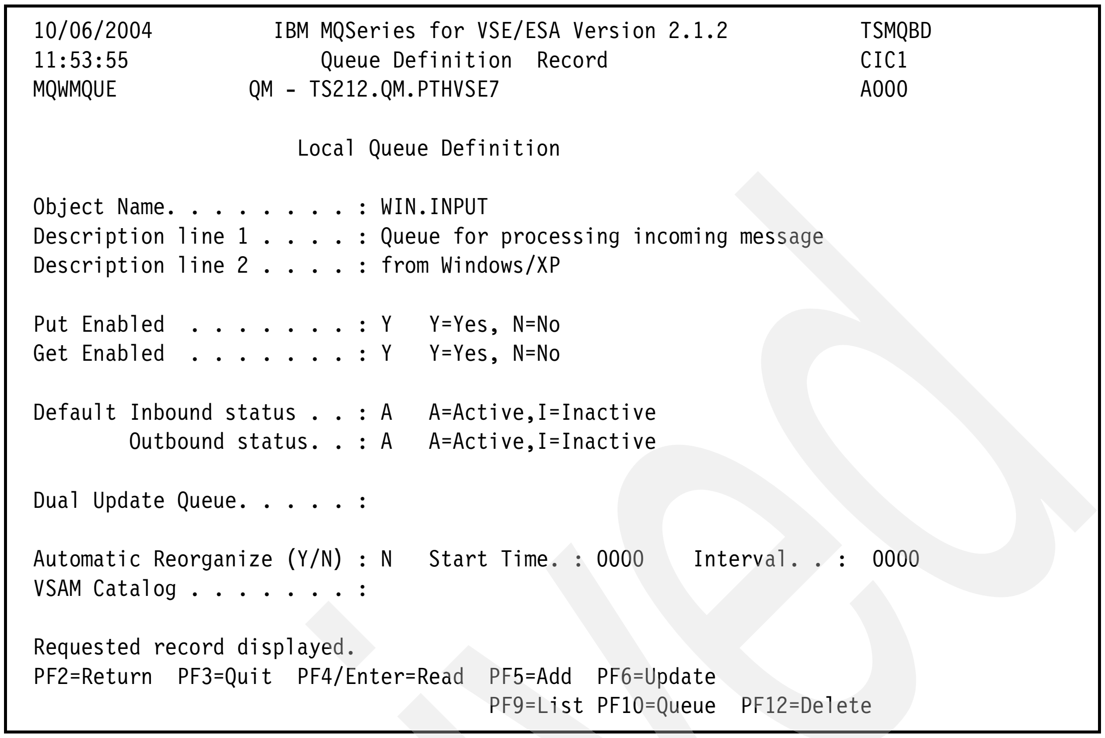


```
10/06/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:49:28               Queue Main Options                          CIC1
MQWMQUE                                                            A000
                       SYSTEM HAS BEEN SHUTDOWN

       Default Q Manager. : TS212.QM.PTHVSE7

       Object Type. . . . :        L = Local Queue
                                   R = Remote Queue
                                   AQ = Alias Queue
                                   AM = Alias Queue Manager
                                   AR = Alias Reply Queue

       Object Name. . . . :







PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF12=Delete
```

Figure 2-5 Initial queue definition screen

From this panel, you can define local, transmission, alias, and remote queues. You can also use this panel to display existing queues.

When defining a local or transmission queue, you must specify an object type of L (for local) and an object name, which is the name of the queue you intend to define. An example screen for defining a local queue is shown in Figure 2-6 on page 42.



```
10/06/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:53:55               Queue Definition  Record                    CIC1
MQWMQUE          QM - TS212.QM.PTHVSE7                             A000

                     Local Queue Definition

Object Name. . . . . . . . : WIN.INPUT
Description line 1 . . . . : Queue for processing incoming message
Description line 2 . . . . : from Windows/XP

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure 2-6   Local Queue Definition screen

The following points should be taken into consideration when choosing values for local queue parameters:

- Object Name

  The name of a queue should be unique to a particular queue manager, but not necessarily for your entire MQSeries environment. As we previously explained, a queue name is fully qualified under the format:

  `Queue_Manager_name.Queue_name`

  This fully qualified name must be unique across your entire network.

- Put/Get Enabled

  These parameters indicate whether or not messages can be put to or retrieved from the queue using MQI calls MQPUT, MQPUT1 and MQGET. Generally, these are set to `Y` to enable getting and putting messages.

- Default Inbound/Outbound Status

  When MQSeries initiates, it may allow applications to immediately write/read messages to/from a queue. This capability depends on what you have specified in these two fields. Unless you have good reasons, we recommend you keep the default value A (Active). Specifying I (Inactive) would lead to manual interventions to activate the queue.

- Dual *xxxxx* Queue

  Where *xxxxx* may be Source or Update.

  Generally, you will not define dual queues, and can leave this field blank. If you choose to specify a dual queue, the queue identified by this parameter must exist prior to the queue being defined; otherwise, the queue definition is rejected with the message “Dual QID not found”. Dual queues and primary queues should have the same characteristics.

Note that this panel defaults to Dual Update Queue, unless you are displaying an existing dual queue that is referenced by a local queue. In this case, the panel displays Dual Source Queue.

The idea behind the concept of a dual queue is to automatically duplicate messages written to a queue for integrity reasons. In such cases, there is a primary queue and a secondary queue (called a dual queue). Whenever a message is written to the primary queue, it is also written to the dual queue. It is obvious that these queues should not be defined in the same VSAM cluster, or to the same disk. It is even recommended (still for integrity reasons) to have these disk volumes attached to different control units.

A dual queue cannot be defined for a queue that is already a dual queue.

You should be aware that, in case of failures, takeovers are not automatic. Applications are indeed notified (via a return/reason code) that the primary queue has become unavailable. However, switching to the dual queue requires user intervention. You must execute the MQPUTIL program in a batch partition with the **`DUALQ TAKEOVER`** command (refer to the *MQSeries for VSE System Management Guide*, GC34-5364*,* for more information about MQPUTIL). To prevent the system operator from doing this manually, you may consider submitting the batch job from a CICS application program to the VSE/POWER Reader queue. This is easily accomplished with EXEC CICS SPOOLOPEN and EXEC CICS SPOOLWRITE REPORT and the JCL option.

- Automatic Reorganize

  The *Automatic Reorganize*, *Start Time*, *Interval*, and *VSAM Catalog* parameters all relate to the automatic VSAM reorganization feature of MQSeries for VSE. If the *Automatic Reorganize* parameter is set to `N`, the other reorganization parameters are ignored.

  If *Automatic Reorganize* is set to `Y`, a VSAM reorganization process will be activated by the queue manager at Start Time, and repeated every interval. The interval is expressed in minutes, so an interval of 1440 means daily.

  The automatic reorganization process is effectively a delete and redefine of the VSAM file that hosts the queue. The reorganization process, therefore, needs to know the VSAM catalog name in which the associated VSAM file will be redefined.

  An automatic reorganization cannot take place while the hosted queue is open to an application, or while a reorganization is already running. Similarly, a queue cannot be opened if its hosting VSAM file is currently being reorganized. Applications that attempt to open a queue that is being reorganized wait until the reorganization is complete.

  You should schedule reorganizations for your VSAM files whether you use the automatic reorganization feature or not. If you use the automatic feature, you should schedule the reorganization to occur at a time when the queue will not be in use, and at a frequency that ensures the VSAM file does not create too many extents.

  For more information about automatic reorganization, and reorganizing your MQSeries VSAM data sets in general, refer to 5.4, “VSAM reorganization” on page 97.

For details on local queue parameters not discussed here, refer to the *MQSeries for VSE System Management Guide*, GC34-5364.

Once you add your initial local queue definition using **PF5**, you are then required to complete the extended local queue definition shown in Figure 2-7 on page 44.

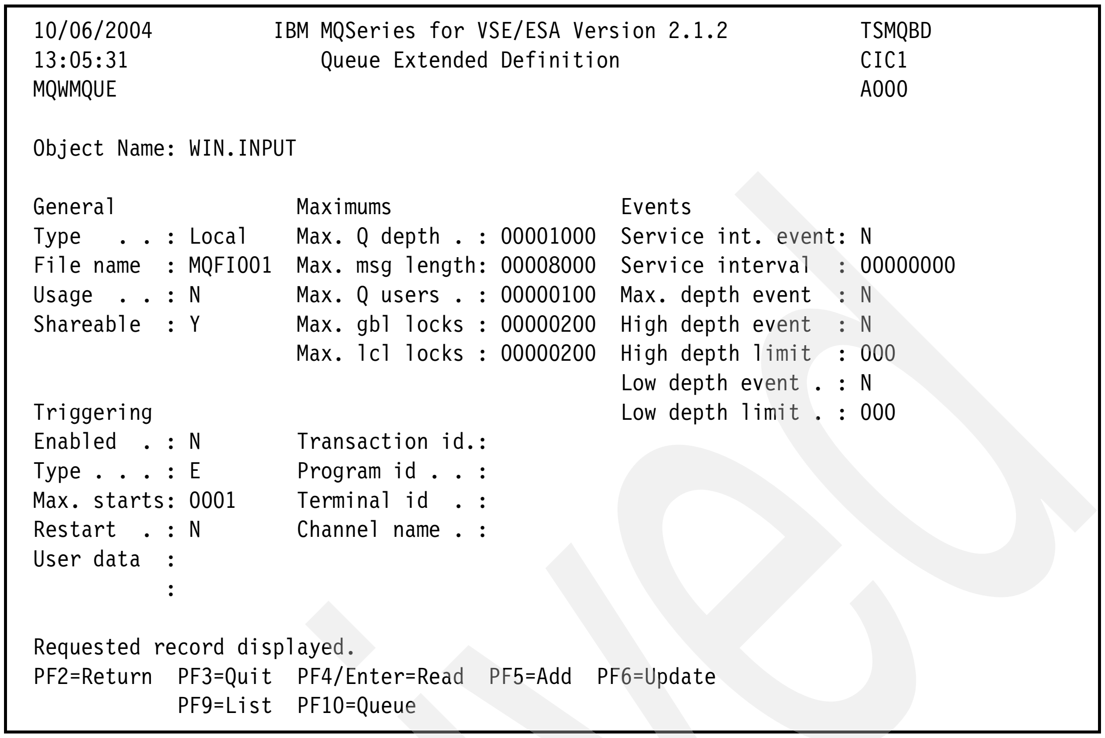

```
10/06/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
13:05:31               Queue Extended Definition                   CIC1
MQWMQUE                                                            A000

Object Name: WIN.INPUT

General              Maximums                   Events
Type   . . : Local    Max. Q depth . : 00001000  Service int. event: N
File name  : MQFI001  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000200  High depth event  : N
                      Max. lcl locks : 00000200  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : N        Transaction id.:
Type . . . : E        Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure 2-7 Local Queue Extended Definition screen

The following points should be taken into consideration when choosing values for extended local queue parameters:

- Shareable

  If you answer `Yes`, then multiple CICS application programs are allowed to simultaneously issue MQGET calls against this queue. This may be a problem for some applications that need exclusive control of the queue (for example, to make sure consecutive messages are processed by the same task). In such a case, specify `No`. Generally, however, you will specify `Y` for this parameter.

- File name

  This represents the file name of the KSDS VSAM cluster that should host this message queue. However, three conditions must be fulfilled before choosing the physical file name:

  a. The file must be defined to VSAM via an IDCAMS job step.

  b. The related entry must be added to your DFHFCT (or equivalent for CICS TS). See 1.5.3, "File Control Table (FCT) definitions" on page 17.

  c. A file label (DLBL) must be added to the Disk Label Area (DLA) permanently (STDLABEL or PARSTD) or temporarily.

- Maximum Values

  The five fields below this heading have the same meaning as they do for the global system definition (refer to 2.1, "Global System Definition (GSD)" on page 29, and the *MQSeries for VSE System Management Guide*, GC34-5364).

  – Maximum Q Depth

    This represents the maximum of *not read* messages you might have in the queue at one time. This value may be lower than or equal to the equivalent parameter you specified in the global system definition.

- Maximum msg length

  This represents the size of the largest message you can have in this queue. Since V2.1, this is only the data part of the message. You do not have to worry about the header size anymore.

  In addition, this value may now be greater than the VSAM Control Interval (CI) size. MQSeries takes care of spanning messages to multiple CIs (you must *not* specify `SPANNED` when you define the cluster to VSAM). This value must be lower than or equal to the equivalent parameter you specified in the global system definition.

- Maximum Q users

  This represents the maximum number of concurrent MQOPEN calls made against this queue. This value must be lower than or equal to the equivalent parameter you specified in the global system definition.

- Maximum gbl locks

  The maximum global lock value is used to manage concurrent access to queues. Ultimately, it is an amount of storage allocated by the queue manager when a queue is first opened. The queue manager uses the storage to keep track of message ranges that have been retrieved by applications. Consequently, if a queue is accessed sequentially, the maximum global locks value does not need to be very high. If queue messages are accessed randomly, the queue manager needs to keep track of a larger number of ranges, and the global locks value needs to be higher.

  Generally, queues are accessed sequentially, so a value as low as 200 should be more than enough. If you plan to access the queue randomly, use a value of 1000.

- Maximum lcl locks

  The maximum local locks value is used for the same purpose as the global locks value described above. Where the global value is allocated once per open queue, the local lock storage is allocated for each active open. So for a single queue, with two applications each with an active open against the queue, the queue manager would allocate one global lock table and two local lock tables.

  Once again, if the queue is accessed sequentially, a value of 200 should be sufficient. If you plan to access the queue randomly, use a value of 1000.

▸ Events

The parameters pertaining to events are briefly explained here. For a full description, refer to the *MQSeries for VSE System Management Guide*, GC34-5364.

- Service int. event

  You can request service interval event messages to be generated by setting this parameter to `Y`. You must also request the generation of performance events in the global system definition. Service interval event messages are written to the performance events queue.

- Service interval

  The service interval is an value expressed in milliseconds. It is the frequency at which you would expect the queue to be accessed during normal operation. If accesses to the queue take longer than the service interval a service interval event message is generated (if performance events are enabled).

- Max. depth event

  Set this parameter to `Y` if you want an event message generated when a queue is full. You must also request the generation of performance events in the global system definition.

– High depth event

  Set this parameter to `Y` if you want an event message generated when a queue reaches a certain depth. The depth is determined by the *High depth limit* parameter, which expresses a percentage of the maximum queue depth. For example, if high depth events are enabled and the high depth limit is set to `90`, an event message is generated when the queue reaches 90% capacity on an MQPUT or MQPUT1 call.

– High depth limit

  As described above, the high depth limit is a percentage relative to the maximum queue depth. For example, if the maximum queue depth is 5000, and the high depth limit is 80%, a high depth event would occur when the 4000th message was placed on the queue. Note that queue depths in this respect pertain to unread messages.

– Low depth event

  Set this parameter to `Y` if you want an event message generated when a queue decreases to a certain depth. The depth is determined by the Low depth limit parameter, which expresses a percentage of the maximum queue depth. For example, if low depth events are enabled and the low depth limit is set to `10`, an event message is generated when the queue drops to 10% capacity on an MQGET call.

– Low depth limit

  As described above, the low depth limit is a percentage relative to the maximum queue depth. For example, if the maximum queue depth is 5000, and the low depth limit is 20%, a low depth event would occur when the 1000th message was removed from the queue. Note that queue depths in this respect pertain to unread messages.

► Triggering

Triggering provides the capability of automatically starting a CICS transaction or program when a message is written to the queue. For example, if you receive a message from a remote queue manager, the message may be processed immediately.

As already mentioned, a transmission queue is a special type of local queue. A transmission queue is a temporary storage queue for messages destined for a remote system. Messages on a transmission queue are sent to the remote system by a special MQSeries program called a Message Channel Agent (MCA). MCAs work in pairs—a Sender MCA and a Receiver MCA.

MQSeries for VSE requires that transmission queues provide a trigger for the Sender MCA program MQPSEND. Consequently, a local queue definition for a transmission queue will always have triggering enabled, and will always identify program MQPSEND as the trigger program. Transmission queues also require the name of the channel that is used to establish a connection with the remote system.

A trigger, whether for a transaction or a program, is enabled by setting the Enabled parameter to `Y`.

► Trigger Type

You may specify `F` or `E`.

– **F** (First). When a message is written into an empty queue, the associated transaction is started or the associated program is linked. If a second message arrives before the first one was read, nothing special occurs. This means that only one transaction or program may be started on a trigger for this queue if you have chosen this option.

– **E** (Every). This option allows a transaction or program to be started every time a message is written to the queue. The maximum number of transactions or programs that may be triggered is specified in the field *Maximum Starts*. This option may typically be used for high message traffic on the queue to obtain a better throughput.

More information about triggering is provided in 8.3, “Trigger application programs” on page 158.

- Restart

  Normally, a trigger is started when a message is written to a queue. However, it may happen that starting a trigger is necessary at other times.

  If this field is set to `Y` (Yes), there are two additional situations when a trigger may also be started:

  – You manually open an non-empty queue.
  – The System Monitor task (MQSM) finds the queue is not empty.

  The System Monitor Task (MQSM) is a long running transaction that is started automatically when MQSeries is started. It performs certain housekeeping tasks such as cleaning up abandoned connections and open queue handles, and monitoring queues for the need to activate trigger instances. In this latter case, if the Restart parameter is set to `Y`, the system monitor starts a trigger instance if one is not already running.

  The idea behind a trigger type of F (first) is to activate a trigger instance to process a block of messages as they arrive on a queue. When the first message arrives the trigger is started, and it runs until all the messages are processed. To do this, the trigger program should include logic to wait a short period of time to be sure no more messages are in transit. It can then terminate, and be reactivated by MQSeries when the next block of messages begins to arrive.

  A problem that can occur is that a stray message may arrive while the trigger program is terminating. The trigger terminates, but because a single message remains on the queue, a new instance is not started even if more messages arrive on the queue. The Restart parameter gets around this problem. If you set Restart to `Y`, the system monitor task will start a trigger instance if there are any messages on the queue and an instance is not already running.

  Although the Restart parameter is useful in this way, it can also cause problems. If you set Restart to `Y` and for some reason the trigger instance fails to remove messages from the queue, or abends, the instance will be repeatedly restarted by the system monitor indefinitely.

  An alternative way around both problems is to use a trigger type of E (every). Strictly speaking, a trigger type of E means a trigger instance is started for every message that arrives on the queue. The number of concurrent instances, however, can be throttled by the Max. Starts parameter. In fact, Max. Starts can be set to `1`, which means only one trigger instance can run at a time, regardless of the number of messages that arrive on the queue.

  With a trigger type of E, when the trigger instance completes, a check is made to see if any messages remain on the queue. If so, a new instance is started until all the messages are processed.

  This approach can also cause problems if the trigger instance fails to remove messages from the queue. If so, when it terminates, MQSeries will detect that there are still messages on the queue and start a new trigger instance. This then causes a processing loop.

  To avoid this situation, the trigger instance can set a special flag in the commarea it is passed on invocation. MQSeries checks this flag when the trigger instance completes. If it is set, MQSeries does not start a new instance. Because the flag resides in the commarea, this solution is only available for program trigger instances and not transaction trigger instances. This method is described in more detail in section 8.3, “Trigger application programs” on page 158.

- Transaction id

  You can trigger either a transaction or a program to run in CICS. If a transaction is triggered, MQSeries issues a CICS START command, and passes a trigger data structure (MQTM) for the transaction. For a program, MQSeries issues a CICS LINK, and passes the same data in the commarea.

- Program id

  If you specify both Transaction id and Program id, the transaction is started, and the program name is passed as a parameter to the transaction. The transaction can retrieve this value as discussed in 8.3, “Trigger application programs” on page 158. You can take advantage of this feature if you plan to trigger the same transaction for many queues. You may consider writing a general purpose trigger program that links to specialized programs depending on trigger start parameters, for example, the message queue name.

- Terminal id

  When a transaction is triggered, an EXEC CICS START TRANSID(*xxxx*) is issued against this transaction. You may also want to associate this transaction with a terminal (this would be valid for a trigger type of F). In such a case, the command would be:

  ```
  EXEC CICS START TRANSID(xxxx) TERMID(yyyy)
  ```

  A good application may be to monitor, in real time on a terminal, the vital information of your enterprise based on messages received from branch offices (MQSeries queue manager connections) or entered locally by operators.

- Channel name

  The Channel name trigger parameter is only meaningful for transmission queue definitions. If you are defining a transmission queue, the channel name should identify a sender channel definition. The sender MCA program, MQPSEND, which is the required trigger program for transmission queues, uses the channel name to establish a remote connection and then sends message from the transmission queue over the connection to a remote receiver MCA. The remote receiver places the received messages on a local queue, or redirects them to another remote host.

- User data

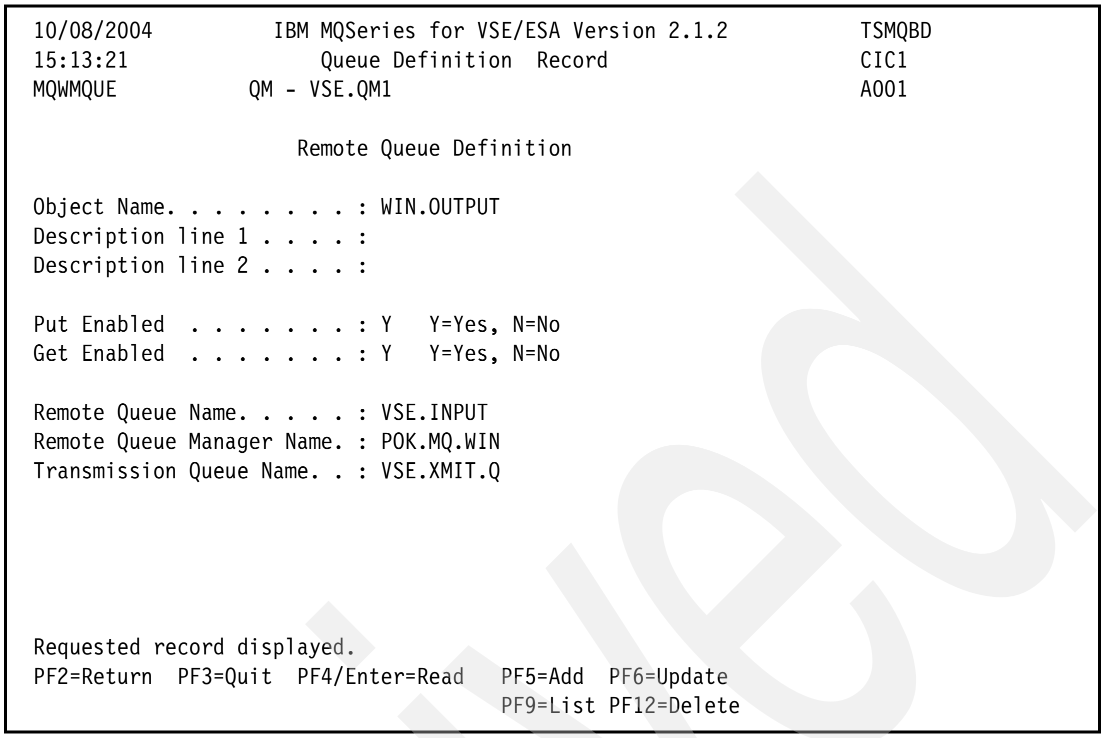
  You can pass an additional 128 characters of data to the trigger transaction or program using the User data parameter. Whatever data you define for this parameter is passed to the trigger instance in the MQTM data structure.

When all fields are filled in, press **PF5** (add). If you are modifying an existing local queue definition, use **PF6** to update. You should receive the message “Record added OK”.

2.2.2 Defining remote queues

Remote queues are also defined via the initial Queue Definitions panel (MQMT option 1.2). When defining a remote queue, you must specify an object type of `R` (for remote) and an object name, which is the name of the queue you intend to define.

An example screen for defining a remote queue is shown in Figure 2-8 on page 49.



```
10/08/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:13:21               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A001

                     Remote Queue Definition

Object Name. . . . . . . . : WIN.OUTPUT
Description line 1 . . . . :
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Remote Queue Name. . . . . : VSE.INPUT
Remote Queue Manager Name. : POK.MQ.WIN
Transmission Queue Name. . : VSE.XMIT.Q





Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read   PF5=Add  PF6=Update
                                       PF9=List PF12=Delete
```

Figure 2-8 Remote Queue Definition screen

Remember, a remote queue is simply a local queue belonging to another MQSeries queue manager. MQSeries uses a *store and forward* method. When putting messages to a remote queue, MQSeries writes the messages to a local queue. The local queue is, in fact, a transmission queue associated with the remote queue definition. MQSeries then tries to transfer these messages to the remote queue manager (if connected). Therefore, to define a remote queue, you need four parameters:

1. The queue name used by application programs (known as the *remote queue name*)
2. The local transmission queue name
3. The remote queue manager name
4. The name of the local queue defined on the remote queue manager

The Put Enabled and Get Enabled parameters are somewhat meaningless for remote queues. You cannot get from a remote queue anyway, and if you cannot put either, the queue itself is useless. It is possible to define an alias for a remote queue, in which case, you could set Put Enabled to `N` for the remote queue and set Put Enabled to `Y` for the alias.

2.2.3 Defining queue aliases

Instead of hard coding queue names in programs, you may use symbolic names called aliases. This way programs may be portable between platforms.

Alias queues are also defined via the initial Queue Definitions panel (MQMT option 1.2). When defining an alias queue, you must specify an object type of AQ (for Alias Queue) and an object name, which is the name of the alias you intend to define.

An example of an alias queue definition is shown in Figure 2-9 on page 50.

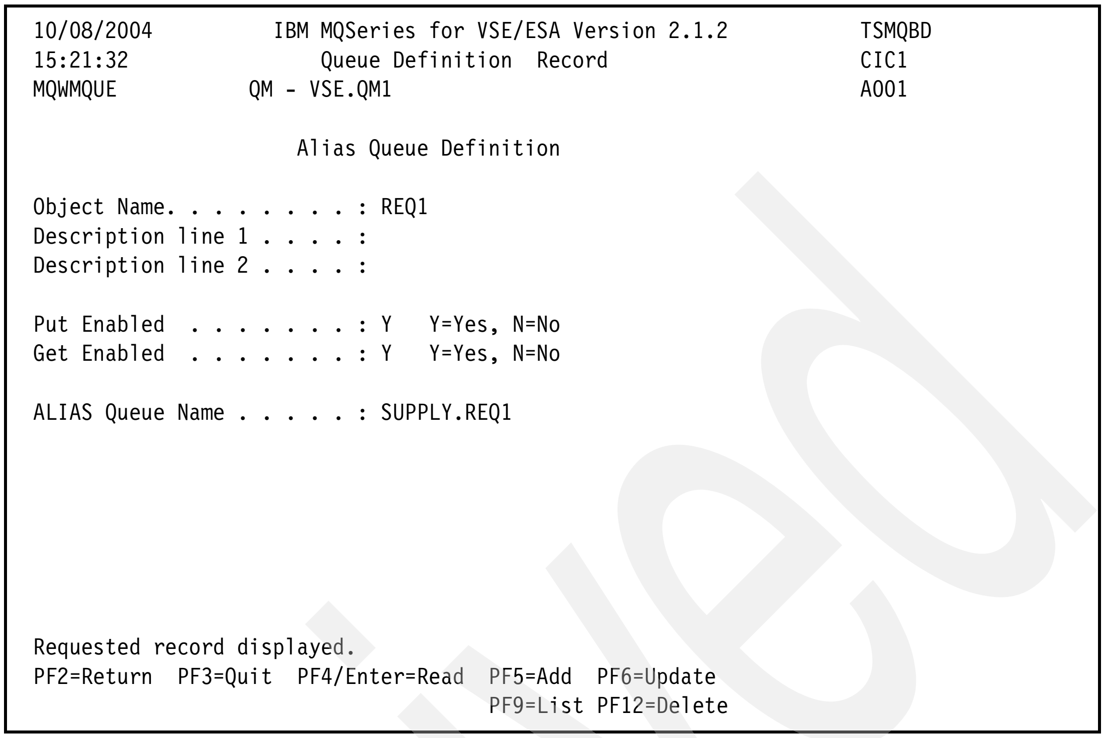

```
10/08/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:21:32               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A001

                     Alias Queue Definition

Object Name. . . . . . . . : REQ1
Description line 1 . . . . :
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

ALIAS Queue Name . . . . . : SUPPLY.REQ1








Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF12=Delete
```

Figure 2-9 Alias Queue Definition screen

The alias queue specifies a relationship between the alias and a queue. You may define multiple aliases for the same queue (although it is not recommended).

When you put or get a message to or from an alias queue, the message is actually stored in or retrieved from the associated or actual queue.

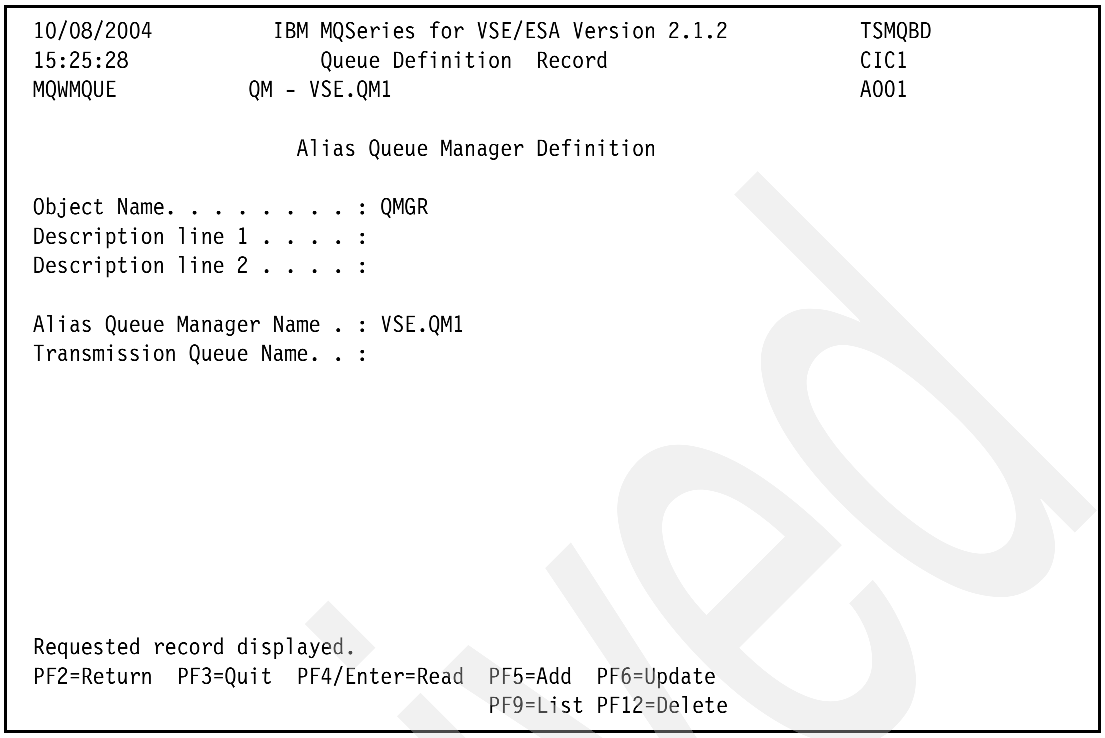
Specifying Put Enabled and Get Enabled settings does not override the equivalent settings at the queue level. That is, if you set these values to N for the alias, but Y for the associated queue, a program that refers to the alias will not be able to read or write messages. However, another program that accesses the queue by its actual name, in this case, is able to read and write messages to the queue.

2.2.4 Defining queue manager aliases

The reason for defining an alias for a queue manager is the same as for queues. That is, so that you can use a symbolic name in programs.

Alias queue managers are also defined via the initial Queue Definitions panel (MQMT option 1.2). When defining an alias queue manager, you must specify an object type of AM (for Alias Queue Manager) and an object name, which is the name of the alias you intend to define.

An example of an alias queue manager definition is shown in Figure 2-10 on page 51.



```
10/08/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:25:28               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A001

                     Alias Queue Manager Definition

Object Name. . . . . . . . : QMGR
Description line 1 . . . . :
Description line 2 . . . . :

Alias Queue Manager Name . : VSE.QM1
Transmission Queue Name. . :









Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF12=Delete
```

Figure 2-10 Alias Manager Definition screen

Unlike the queue manager name, the alias name does not need to be unique. In fact, for portability of programs, the same alias queue manager name may appear on multiple platforms.

The Alias QM Name parameter specifies the queue manager name for which you are creating the alias. Do not be confused, it is not the alias name. The alias name is specified by the Object Name parameter. For remote queue managers, you may leave this field blank and specify a Transmission Queue Name instead. This is because MQSeries is able to retrieve a remote queue manager name from the transmission queue definition.

2.2.5 Defining reply queue aliases

When an application program issues an MQPUT or MQPUT1 call to write a message into a queue, it may specify as a parameter the name of one of its local queues (called reply-to queue) to which the partner, usually a remote site, may send back an acknowledgment of message arrival or delivery.

An example of an alias reply queue definition is shown in Figure 2-11 on page 52.

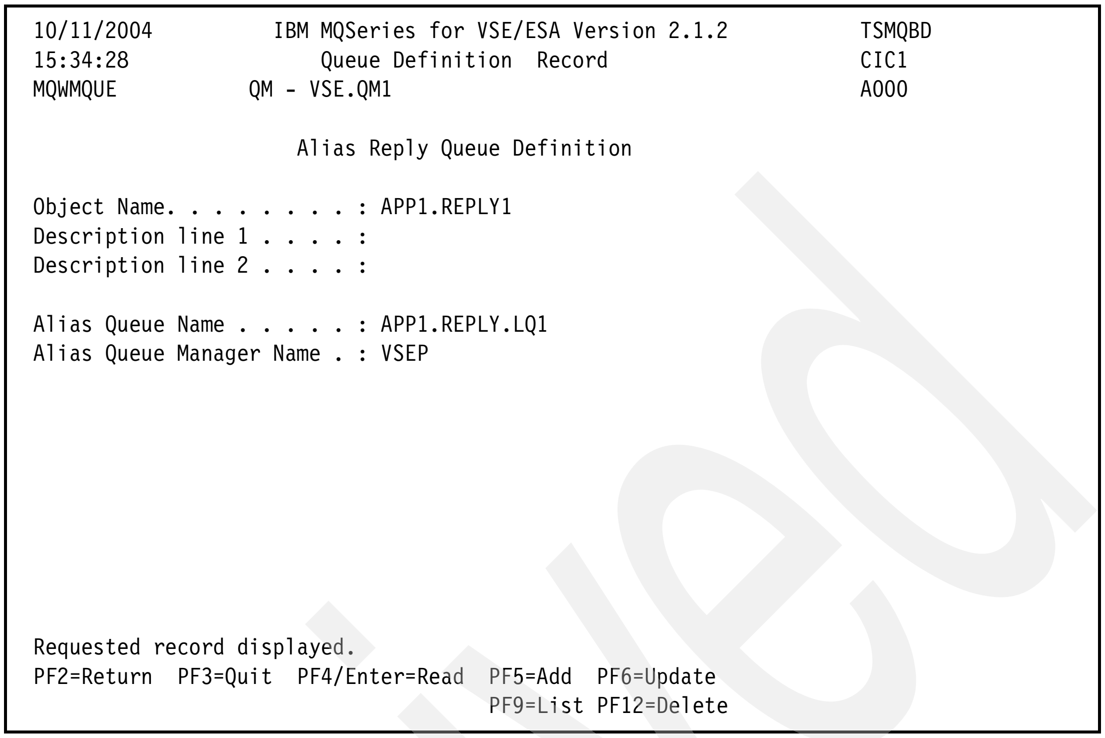

```
10/11/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:34:28               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A000

                     Alias Reply Queue Definition

Object Name. . . . . . . . . : APP1.REPLY1
Description line 1 . . . . . :
Description line 2 . . . . . :

Alias Queue Name . . . . . . : APP1.REPLY.LQ1
Alias Queue Manager Name . . : VSEP










Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF12=Delete
```

Figure 2-11   Alias Reply Definition screen

Alias reply queues are useful to simplify MQSeries definitions for reply queue processing. Use of alias reply queues allows configuration to be local to the MQSeries system interested in reply information without requiring addition configuration on remote systems.

For alias reply queues, the *Alias Queue Name* represents the intended queue for reply acknowledgment messages, and *Alias QM Name* represents the relevant queue manager.

For a full explanation of reply queues, refer to 8.5, “Application design and reply-to queues” on page 162.

2.3 Configuring channels

A channel is a unidirectional logical path between two partners defining a pair. At one end, there is the sender, and at the other end, there is the receiver. In the case of clients, one end is a client and the other a server.

A Queue Manager (QM) may be both a sender and a receiver. So, if a QM, for example QM A, wants to exchange messages in either direction with a QM B, then two channels need to be defined:

- From A to B
- From B to A

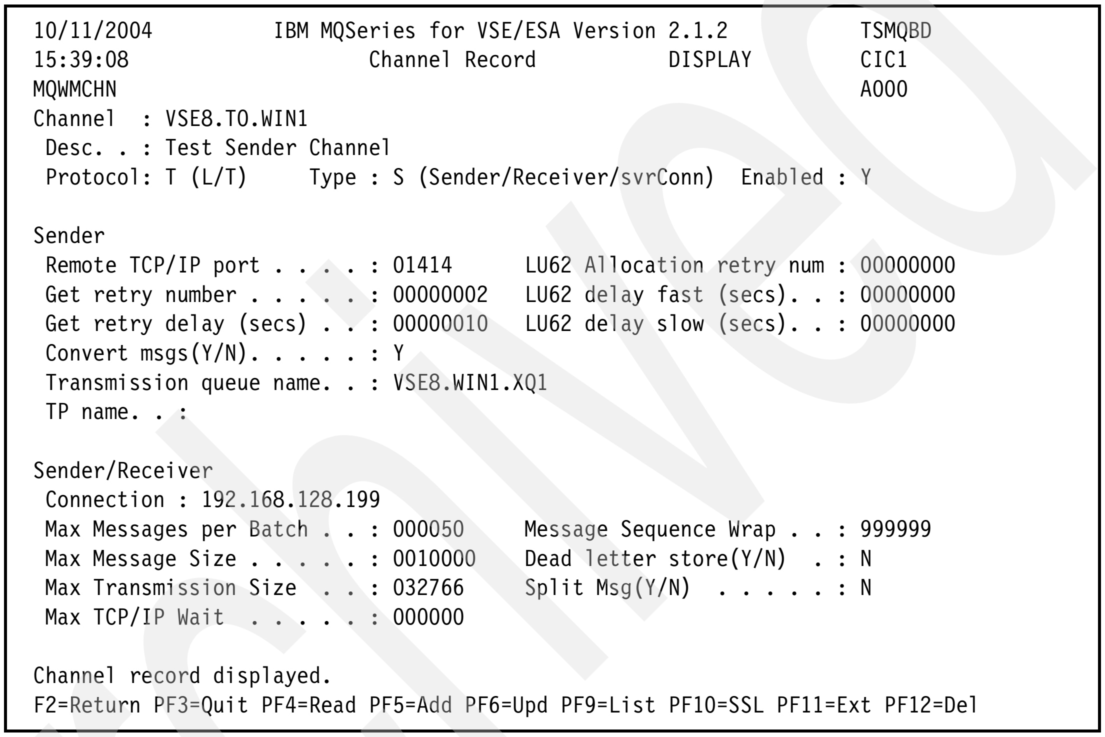
A channel must be defined at both sides and it must have the same name.

Important: On a few platforms, channel and queue names are case sensitive. We recommend that you always use uppercase for object definitions, not only under VSE, but also on other platforms.

Please refer to 6.3, “Channels” on page 106 and 6.4, “Message channel agents and communication programs” on page 115 for more detailed information. In this section, we provide detail on channel parameters that is not provided in the *MQSeries for VSE System Management Guide*, GC34-5364.

2.3.1 General configuration

Channels are defined via the Channel Definitions panel (MQMT option 1.3). Both sender and receiver channels are defined from this panel. An example of a channel definition is shown in Figure 2-12.



```
10/11/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:39:08                    Channel Record           DISPLAY         CIC1
MQWMCHN                                                              A000
Channel  : VSE8.TO.WIN1
 Desc. . : Test Sender Channel
 Protocol: T (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 01414      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000002   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : Y
 Transmission queue name. . : VSE8.WIN1.XQ1
 TP name. . :

Sender/Receiver
 Connection : 192.168.128.199
 Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0010000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 2-12 Channel definition screen

Channel parameters are logically grouped into three categories:

- General
- Sender
- Sender/Receiver

General parameters pertain to all channel definitions. These include the name of the channel, its type (Sender, Receiver or Client), the protocol, and an indicator for whether or not the channel is enabled. For client (SVRCONN) channels, only the general parameters are relevant. Sender and Sender/Receiver parameters do not affect client channel definitions.

Sender parameters pertain to sender channels only. Parameters prefixed `LU62` pertain exclusively to LU6.2 channel definitions. The TP Name parameter is also only relevant to LU6.2 channels, while the port number is only relevant to TCP/IP channels.

The Sender/Receiver parameters pertain to both sender and receiver channel definitions.

When defining channels, you should consider the following points:

- LU6.2 sender parameters

  When MQSeries for VSE is unable to allocate a session with a partner LU, it repeats the allocation process as many times as indicated by the *LU62 Allocation retry num* parameter. The time interval between each retry is specified by the *LU62 delay fast* parameter. If the allocation still fails, MQSeries retries at an interval of LU62 delay slow. Of course, doing too many retries too often may have a negative impact on overall CICS performance.

- Get retry number and delay

  These two parameters are relevant when the transmission queue is empty and before releasing resources held by a sender task. In other words, when the sender MCA empties a transmission queue, it retries the queue every *retry delay* seconds, *retry number* times. If no new messages have arrived on the transmission queue, the sender MCA terminates.

- Max Messages per Batch

  This parameter specifies the number of messages to be processed by sender or receiver tasks before an end of CICS Logical Unit of Work (LUW); that is, before the work is committed using a CICS syncpoint. Here again, it is typically a performance issue. The smaller the batch, the more frequently MQSeries issues a syncpoint. These are syncpoints issued by the sender and receiver MCAs.

  If the *Max Messages per Batch* parameter is set too high, MQSeries will demand more CICS resources between syncpoints. Consequently, you need to consider both performance and available resources when setting this parameter. A value of 100 may be sufficient.

- Message Sequence Wrap

  At each end of a channel, MQSeries keeps track of the number of messages sent or received. This is done by using a Message Sequence Number (MSN). When the MSN becomes greater than the value specified in this parameter, it restarts from zero at both ends.

  If both partners do not specify the same *Message Sequence Wrap* value, then initial negotiations before sending messages fail, and the channel is closed. This is one of the most common reasons for communication failures. So if you cannot establish communication between two queue managers, you may start your problem determination by verifying this value. Negotiation failures between sender and receiver MCAs are logged to the MQSeries system log.

- Convert msgs

  The Convert msgs parameter indicates whether or not messages should be translated as they are taken from the transmission queue and before they are sent to the remote system. The sender MCA uses the MQGET call to retrieve message from the transmission queue. If this parameter is set to `Y`, the sender MCA uses the MQGMO_CONVERT option when retrieving messages.

  Since this is the case, care should be taken to ensure that MQSeries can convert the messages on the transmission queue. Conversion is affected by the local and remote code pages, and the format specified in the message descriptor (MQMD). MQSeries provides conversion for certain built-in formats. In all other cases, you must provide a data conversion exit program, or set the Convert msgs parameter to `N`.

- Transmission Queue Name

  When you define a transmission queue, you specify a channel name. Conversely, when you define a sender channel you identify the associated transmission queue. This forms a one-to-one relationship.

- Connection

  For LU 6.2 channels, this parameter identifies a connection defined to CICS. For TCP/IP channels, this parameter identifies an IP address or host name. Note that you should not use leading zeros in an IP address (for example, 161.037.042.001) because MQSeries for VSE treats these as octal values.

  Connection definitions also involve session definitions in CICS. Refer to 1.5.2, "Systems Network Architecture (SNA) definitions" on page 15 for more details about these definitions.

- TP Name

  This parameter defines the Transaction Program (TP) name of the partner for APPC (LU6.2) communication. Therefore, it is ignored when using the TCP/IP protocol. Its meaning depends on the platform. For example:

  – For a remote QManager on a VSE system, it is just the transaction name MQ01 that starts program MQPRECV, the receiver MCA.

  – For a remote QManager on a OS/2 Warp system, you specify a TP name definition for Communication Manager (CM/2), usually AMQCRS6A. The definition for CM/2 would be as follows:

```
DEFINE_TP  TP_NAME(AMQCRS6A)
           PIP_ALLOWED(NO)
           FILESPEC(D:\MQM\BIN\AMQCRS6A.EXE)
           PARM_STRING(-n AMQCRS6A -m DCT.MQ.OS2)
           CONVERSATION_TYPE(ANY_TYPE)
           CONV_SECURITY_RQD(NO)
           SYNC_LEVEL(EITHER)
           TP_OPERATION(NONQUEUED_AM_STARTED)
           PROGRAM_TYPE(FULL_SCREEN)
           RECEIVE_ALLOCATE_TIMEOUT(INFINITE);
```

  When VSE establishes a connection with WARP, all information to start the partner program is found in the CM/2 definition file. The item to retrieve is TP_NAME (AMQCRS6A). From this definition, CM/2 is able to start the OS/2 session with:

```
D:\MQM\BIN\AMQCRS6A.EXE -n AMQCRS6A -m DCT.MQ.OS2
```

  The AMQCRS6A.EXE program is the receiver MCA for MQSeries on OS/2. Refer to "CM/2 configuration file MQSERIES.NDF" on page 306 to see the entire CM/2 configuration file.

Important: The *MQSeries for VSE System Management Guide*, GC34-5364, has an ambiguous sentence about the TP Name. It says:

Note: Although the TPNAME can be up to 64 bytes elsewhere, for MQSeries purposes it can have a maximum of 4 bytes.

This should read: *… for MQSeries for VSE purposes, it…*

This is because the TP Name for a target VSE system refers to a CICS transaction name (MQ01). The TP Name can be greater than 4 bytes for non-VSE target systems.

- Dead letter store

  When a message is sent to a remote Queue Manager (QM), and the remote QM cannot find or use the target queue, then:

- If the remote QM has defined a Dead Letter Queue (DLQ), the message goes to the DLQ. A feedback message is then sent to the sender MCA to be written into the system log file.
- If the remote queue manager could not put the undeliverable message to the DLQ, the sender MCA is notified, and both the sender and receiver log the error. The channel is terminated with messages remaining on the transmission queue.

> **Note:** A sample program to handle the Dead Letter Queue (DLQ) can be downloaded from the following Web site:
>
> `http://www.ibm.com/software/ts/mqseries/platforms/vseesa/`
>
> Follow the links to freeware SupportPacs for the MQSeries base product.

You can send messages to a remote queue on a remote queue manager. This in effect uses the remote queue manager as a router, because transmitted messages are placed on a remote queue and forwarded to yet another remote queue manager. If the remote queue definition is incorrect, and the remote receiver MCA cannot put messages to the remote queue, they are written to the DLQ and the sender is notified. If, however, they are successfully placed on a transmission queue for subsequent forwarding, the messages remain on the transmission queue until they are forward to the next remote queue manager. The rules for DLQ processing are then repeated.

When a message is sent to an unknown queue manager, and if this QM is defined as adjacent to the sender QM (they should be connected through a channel), the message stays in the transmission queue. This seems obvious since, in such a case, the channel could not be started.

- Max Transmission Size

  The maximum transmission size represents the maximum number of bytes that can be sent over an active connection at one time. The upper limit for this is 65535 for TCP/IP connections and 32000 for LU6.2 connections.

  Messages that exceed the maximum transmission length are segmented over a number of transmissions. Messages shorter than the maximum transmission length can be sent in a single transmission, however, MQSeries prepends a number of data structures ahead of the message that take up some of the transmission buffer area. The prepended data ranges from 48 to 476 bytes depending on whether the message is sent in a single transmission or is split over multiple transmissions.

  This parameter is not generally available on other MQSeries platforms. It is available with MQSeries for VSE to make the amount of storage allocated in CICS configurable. For example, if you were receiving messages smaller than 1024 bytes, and were using a batch size of 1, you could specify a maximum transmission size as small as `1500`.

- Max TCP/IP Wait

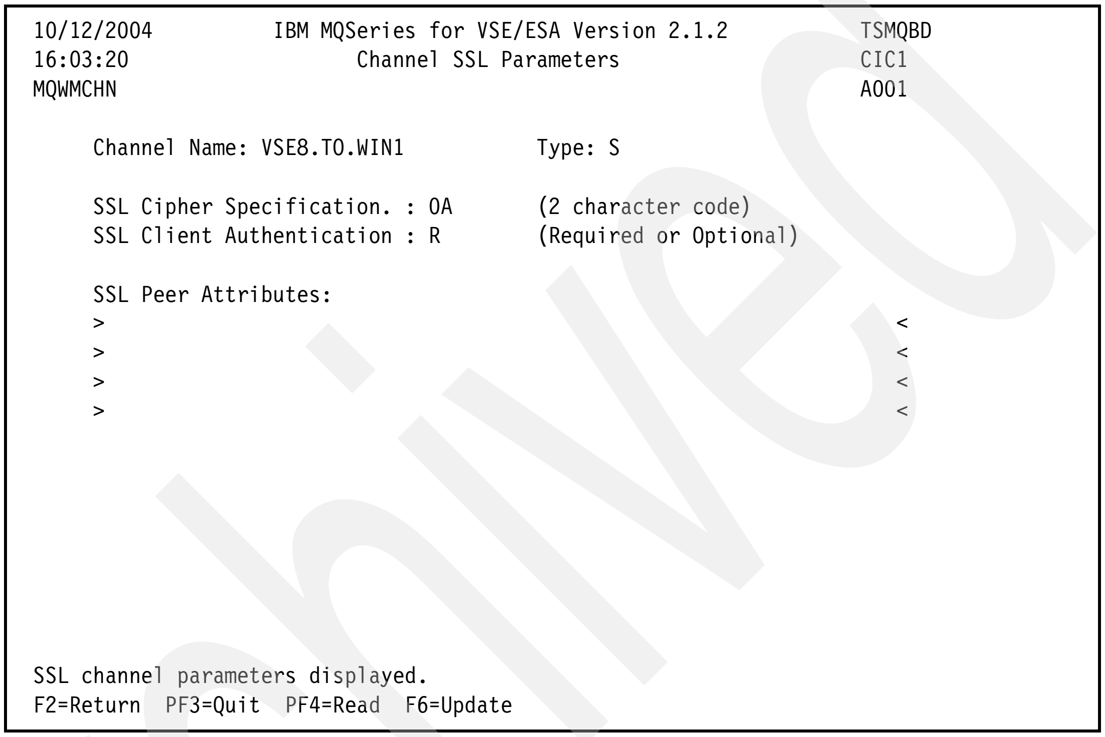
  The *Maximum TCP/IP Wait* parameter, as its name suggests, is only applicable to TCP/IP channels. It determines how long a sender or receiver MCA waits for data before “concluding” that the channel has failed. A value of 0 means the MCA will wait indefinitely.

  For receiver channels, if a value other than 0 is used, it is important to set the maximum TCP/IP wait to a value greater than the sender channel's disconnection interval. On MQSeries for VSE the disconnection interval is the channel's *get retries* setting multiplied by the *get delay*. This is the amount of time the sender MCA waits for messages to arrive on a transmission queue before “concluding” that all the messages have been sent and the channel can be stopped.

Because a receiver channel will always wait for the disconnection interval before terminating normally, the maximum TCP/IP wait, if used, should be greater to avoid continual failure.

2.3.2 SSL configuration

From the Channel definition screen (MQMT option 1.3), you can use **PF10** to access the channel's Secure Sockets Layer (SSL) configuration.



```
10/12/2004         IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
16:03:20                  Channel SSL Parameters                    CIC1
MQWMCHN                                                             A001

     Channel Name: VSE8.TO.WIN1          Type: S

     SSL Cipher Specification. : 0A      (2 character code)
     SSL Client Authentication : R       (Required or Optional)

     SSL Peer Attributes:
     >                                                                 <
     >                                                                 <
     >                                                                 <
     >                                                                 <









SSL channel parameters displayed.
F2=Return  PF3=Quit  PF4=Read  F6=Update
```

Figure 2-13 Channel SSL parameters

SSL configuration is optional. If you specify an SSL Cipher Specification, the channel will be deemed “SSL-enabled.” If a channel is SSL-enabled, its partner channel must also be SSL-enabled. It must also identify the same cipher specification.

Cipher specifications for MQSeries for VSE are 2-character codes. At the time of publication, the SSL for VSE product supported the following cipher specifications:

```
01 for RSA512_NULL_MD5
02 for RSA512_NULL_SHA
08 for RSA512_DES40CBC_SHA
09 for RSA1024_DESCBC_SHA
0A for RSA1024_3DESCBC_SHA
62 for RSA1024_EXPORT_DESCBC_SHA
```

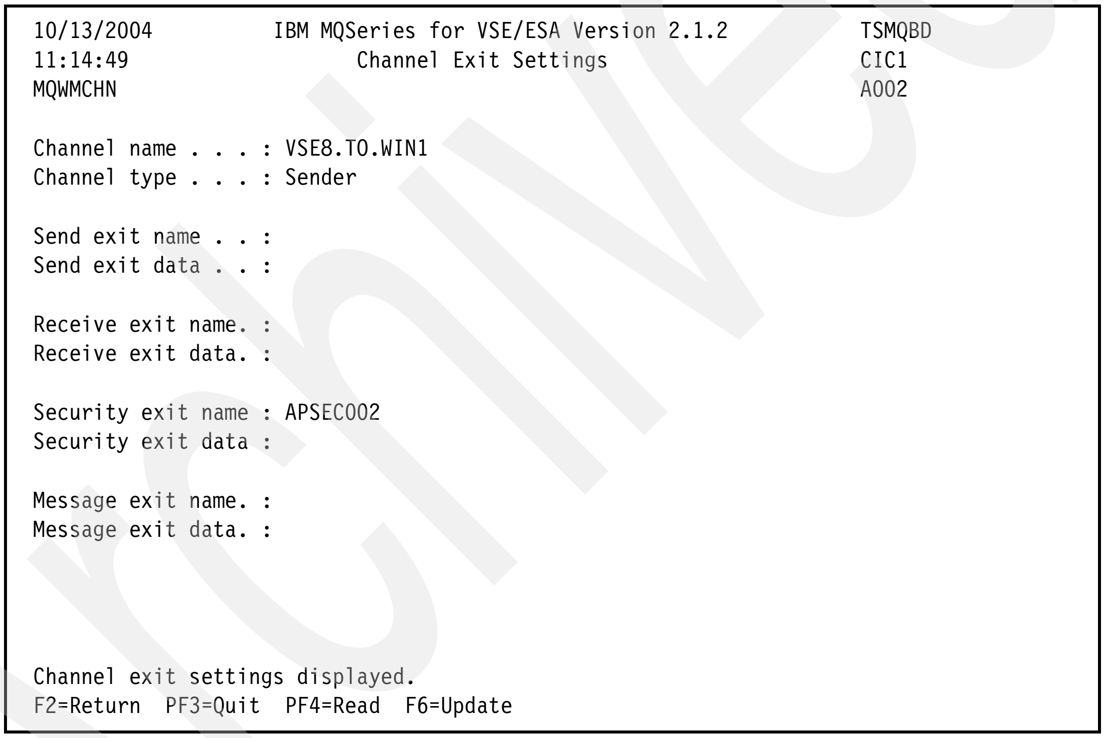
The Client Authentication parameter can be set to `R` (Required) or `O` (Optional). If the channel is a VSE receiver channel, it must be set to `R`. This is because VSE receiver channels require the remote system (or SSL client) to supply their certificate during SSL initialization processing.

The Peer Attributes parameter is used to determine whether or not the remote system's certificate contains specific and expected detail. The remote system's certificate is exchanged during SSL initialization. Its contents can then be compared with the requirements specified in the Peer Attributes parameter. For example, the Peer Attribute may stipulate that the remote system's certificate has an organization of "IBM". Keywords and syntax for the Peer Attributes parameter are described in the *MQSeries for VSE System Management Guide*, GC34-5364.

Most problems with SSL-enabled channels occur during SSL initialization. Initialization occurs when the channel is started. At this time, the two systems exchange SSL negotiation data. If the initialization fails, the channel is stopped with an error. You should ensure both ends of the channel identify the same cipher specification, and that the cipher specification is compatible with the certificate in use. The certificate used by the queue manager is the one identified in the queue manager's communication settings.

2.3.3 Exit configuration

From the Channel definition screen (MQMT option 1.3), you can use **PF11** to access the channel's Exit configuration.



```
10/13/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:14:49                  Channel Exit Settings                    CIC1
MQWMCHN                                                            A002

Channel name . . . : VSE8.TO.WIN1
Channel type . . . : Sender

Send exit name . . :
Send exit data . . :

Receive exit name. :
Receive exit data. :

Security exit name : APSEC002
Security exit data :

Message exit name. :
Message exit data. :




Channel exit settings displayed.
F2=Return  PF3=Quit  PF4=Read  F6=Update
```

Figure 2-14 Channel Exit Settings

Channel exits are customer-written programs that are run by MQSeries at predefined times during channel operation. For example, send and receive exits are run every time an MCA program sends or receives data over an active channel connection. Generally, you do not need to implement channel exits, and you can leave this part of the channel definition blank.

If you do plan to use channel exits, you need to keep certain rules in mind. Exits generally work in pairs. For example, a send exit that modifies data before it is transmitted to a remote system generally works in conjunction with a receive exit that applies a “reverse” modification when it receives the transmission. A security exit that passes authentication data works in conjunction with another security exit that is expecting the authentication data.

Operation

There are five MQSeries operation options available under MQMT option 2 (Operation):

- Start/Stop Queues
- Open/Close Channels
- Reset Channel Message Sequence
- Initialize/Shutdown of System
- Maintain Queue Message Records

These allow you to take action on the queue manager itself, or the objects managed by the queue manager. Again, our goal here is not to repeat what is explained in the *MQSeries for VSE System Management Guide*, GC34-5364, but to provide you with additional information.

3.1 Start/stop queues

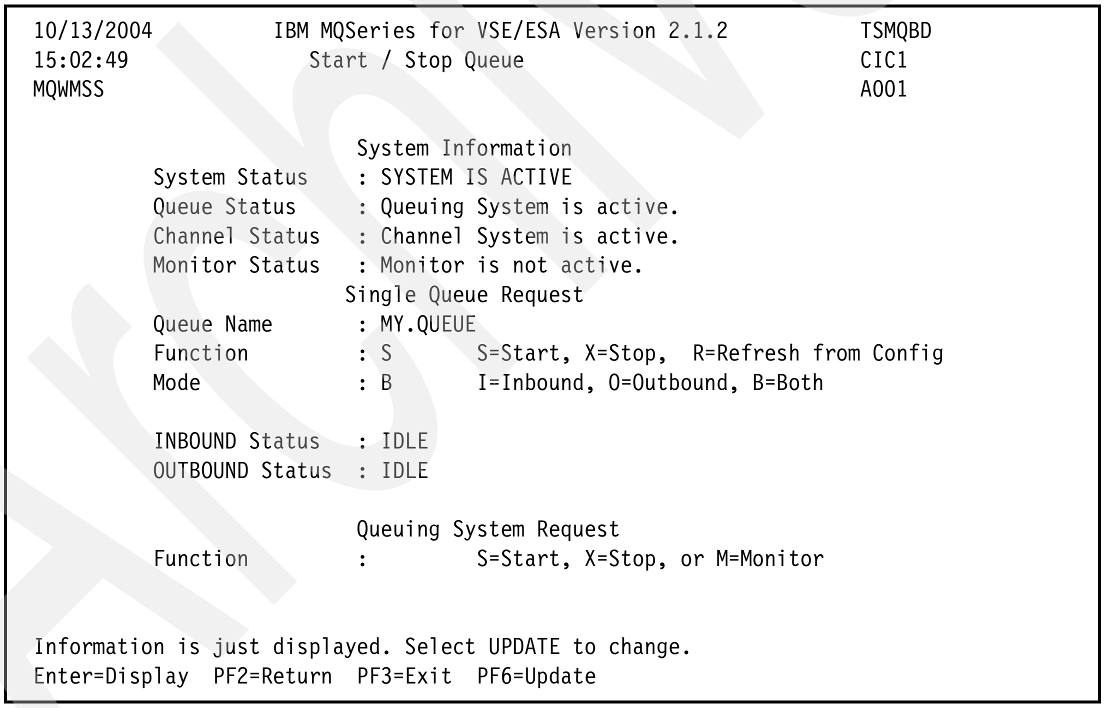
First, you should notice that this option is about *start* and *stop* for queues, and not *open* and *close*. Do not be confused. Opening and closing queues applies to MQI calls. From this option, you can control whether the queue manager accepts MQOPEN and MQCLOSE requests from application programs against a particular queue.

There is a trap you should be aware of when starting and stopping queues. When you define queues, there are four related fields:

- Default Inbound Status
- Default Outbound Status
- Get Enabled
- Put Enabled

When you start a queue from this dialog, you may only change the inbound and outbound statuses from disabled to idle (or the opposite). But if you specified Get/Put disabled when you defined the queue, they stay disabled. That is, if an application program tries to get or put messages from or to this queue, a reason code 2051 is returned. The only way of changing this status is by using an application program issuing an MQI MQSET call, or by changing the queue definition itself.

Figure 3-1 shows an example of the Start/Stop Queue screen.



```
10/13/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
15:02:49               Start / Stop Queue                            CIC1
MQWMSS                                                               A001

                           System Information
          System Status    : SYSTEM IS ACTIVE
          Queue Status     : Queuing System is active.
          Channel Status   : Channel System is active.
          Monitor Status   : Monitor is not active.
                          Single Queue Request
          Queue Name       : MY.QUEUE
          Function         : S         S=Start, X=Stop,  R=Refresh from Config
          Mode             : B         I=Inbound, O=Outbound, B=Both

          INBOUND Status   : IDLE
          OUTBOUND Status  : IDLE

                           Queuing System Request
          Function         :           S=Start, X=Stop, or M=Monitor


 Information is just displayed. Select UPDATE to change.
 Enter=Display  PF2=Return  PF3=Exit  PF6=Update
```

Figure 3-1 Start/Stop Queue screen

The use of this screen is well explained in the *MQSeries for VSE System Management Guide*, GC34-5364.

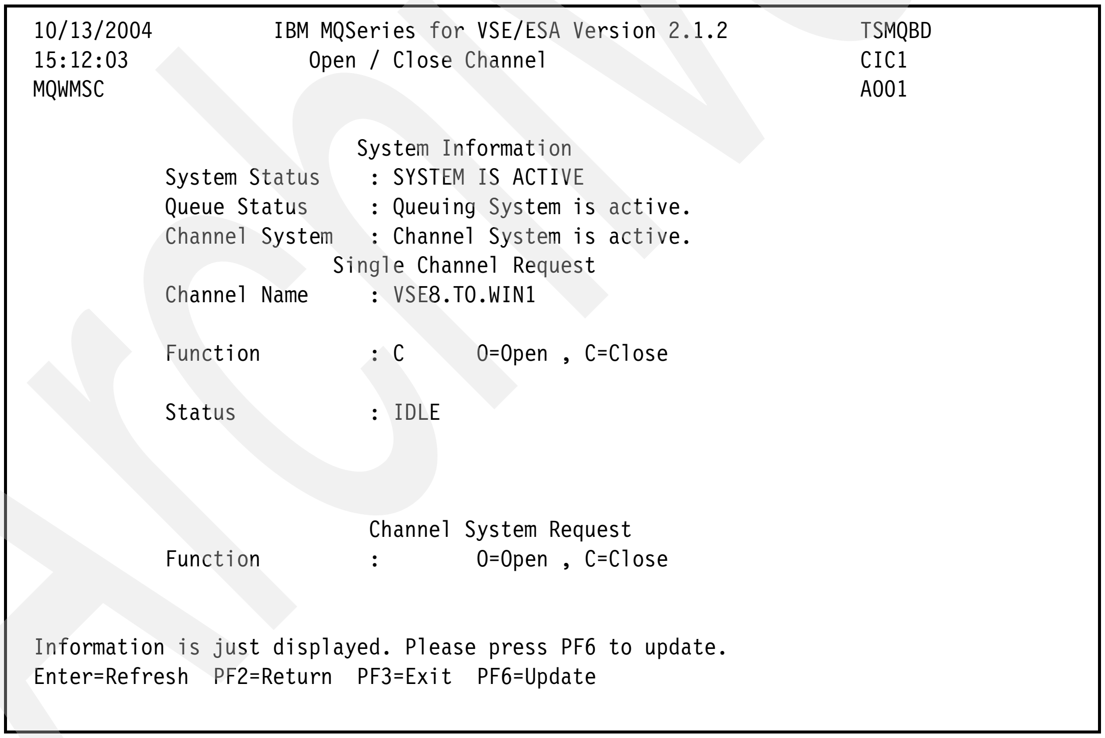
The last field of this screen, however, requires some elaboration. The Queuing System Request function can only be set if you have not specified values in other fields. It has a different meaning depending on what you are entering:

- It allows you to enable or disable all functions related to queues using S to Start or X to Stop. The *MQSeries for VSE System Management Guide*, GC34-5364 states that it is the system manager that is enabled or disabled. This is not the case. It is, in fact, queuing services that are started or stopped. The label of this field is misleading because queues keep the status they had before “stopping all queues” (for example, a queue is not started if it was in a stop state).
- It activates the trace facility of MQI calls (API Monitoring). Use the trace very carefully; otherwise, your system monitor queue may grow rapidly (with a lot of VSAM secondary allocations). You can also experience performance degradation. To activate the trace enter M, then press **PF6**. To deactivate the trace do the same. This selection acts as a toggle.

3.2 Open/close channels

When you define a channel, you may specify whether or not it should be enabled or disabled at MQSeries initialization. From this panel, you may change the initial status. This may be especially useful if you wish to stop sending (or receiving) messages to or from a remote queue manager or MQSeries clients.

An example of the Open/Close Channel screen is shown in Figure 3-2.



```
10/13/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:12:03              Open / Close Channel                         CIC1
MQWMSC                                                             A001

                           System Information
           System Status    : SYSTEM IS ACTIVE
           Queue Status     : Queuing System is active.
           Channel System   : Channel System is active.
                         Single Channel Request
           Channel Name     : VSE8.TO.WIN1

           Function         : C      O=Open , C=Close

           Status           : IDLE



                            Channel System Request
           Function         :        O=Open , C=Close


Information is just displayed. Please press PF6 to update.
Enter=Refresh  PF2=Return  PF3=Exit  PF6=Update
```

Figure 3-2 Open/Close Channel screen

Closing and opening a sender channel has no influence on the trigger specified for the associated transmission queue (MQPSEND). Just stopping then starting a queue which has `Triggering Restart = Y` or writing more messages to a queue may restart a trigger.

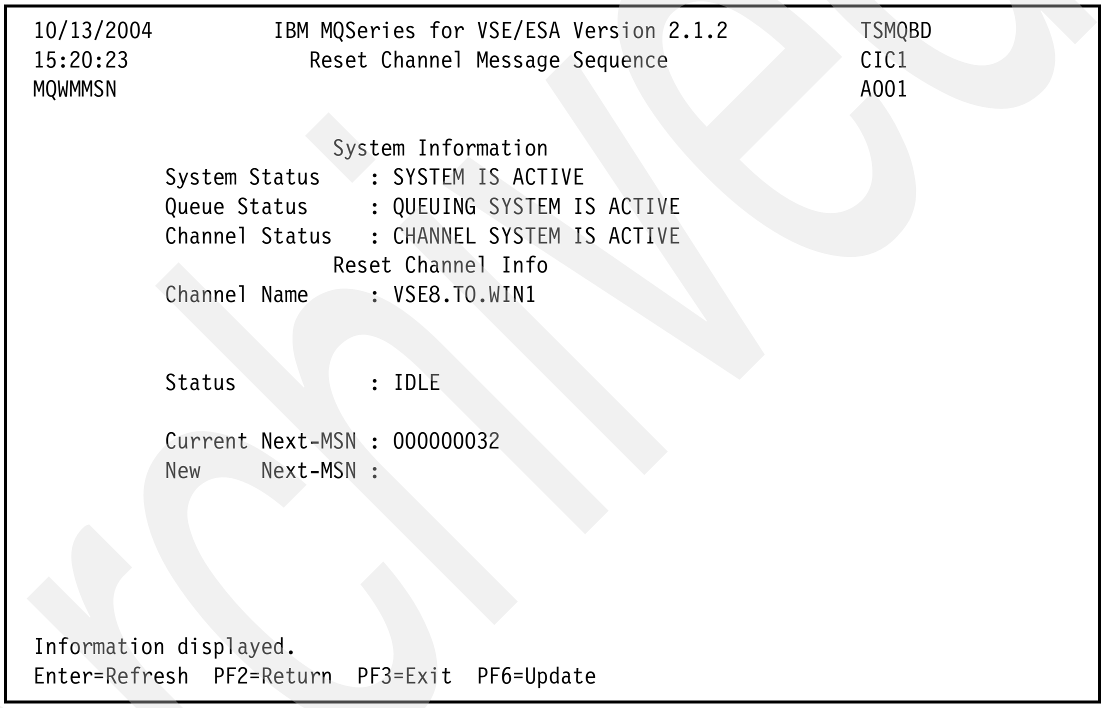
What we have said about queues also applies to channels. You do not open or close *all* channels, but you just enable or disable the *channel system*.

3.3 Reset channel message sequence

Channels assign a sequence number to each message sent or received. This sequence number is called a Message Sequence Number (MSN). The last MSN is saved within the channel record of the configuration file. If the MSNs at both ends of a channel are different, the initial negotiations for exchanging messages with the remote queue manager fail.

However, it may happen that this counter is out of sync with the partner counter for several reasons. For example, the partner has deleted and redefined the channel. Or a syncpoint was issued on one side but not on the other side (this may happen because VSE MQSeries does not have a two-phase commit process). In such an instance, you can manually change the MSN for a channel. The channel must be stopped before you can reset the message sequence number.

An example of the Reset Channel Message Sequence number screen is shown in Figure 3-3.



```
10/13/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:20:23              Reset Channel Message Sequence               CIC1
MQWMMSN                                                            A001

                          System Information
           System Status    : SYSTEM IS ACTIVE
           Queue Status     : QUEUING SYSTEM IS ACTIVE
           Channel Status   : CHANNEL SYSTEM IS ACTIVE
                          Reset Channel Info
           Channel Name     : VSE8.TO.WIN1


           Status           : IDLE

           Current Next-MSN : 000000032
           New     Next-MSN :





Information displayed.
Enter=Refresh  PF2=Return  PF3=Exit  PF6=Update
```

Figure 3-3 Reset Channel Message Sequence screen

You can also verify the MSN current value by entering the channel name and pressing the Enter key. Be careful, the value indicated here is the next MSN. If we check the MSN on the remote site, it could be either the MSN of the last message or the next MSN (depending on the operating system). In our example, the current value is 32, but for the related Windows XP channel it is 31.

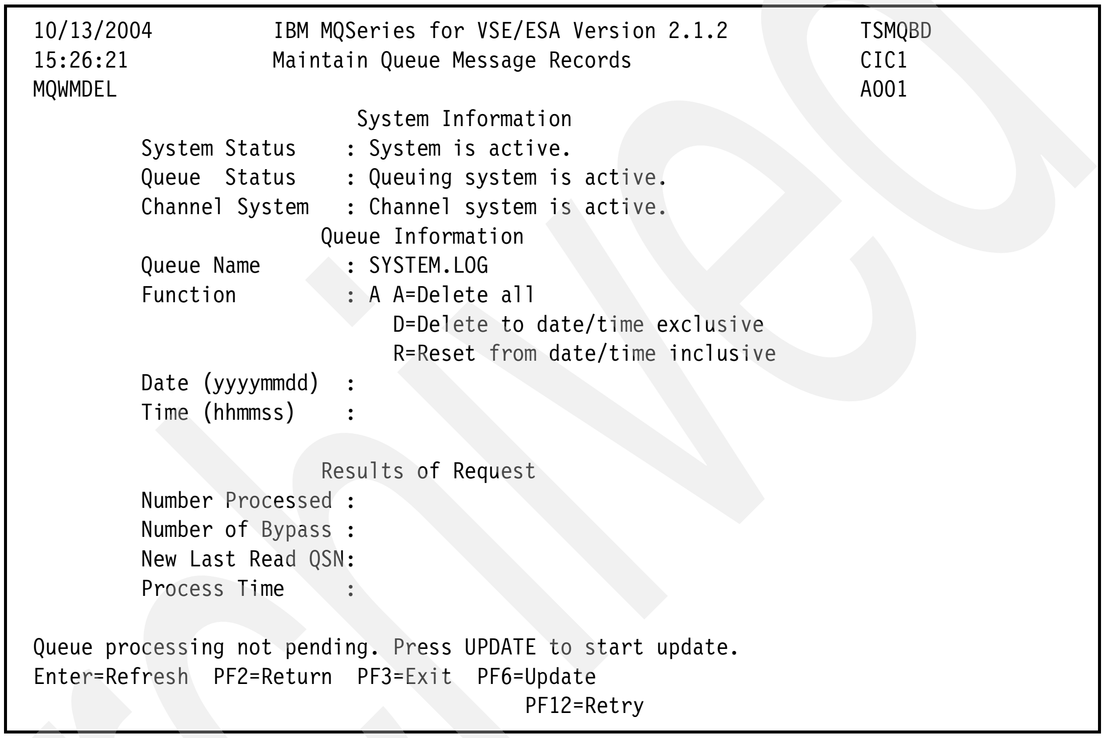
3.4 Initialization/shutdown of the system

The only thing we can add here is related to the batch interface.

In 8.2, “Batch application programs” on page 151, we explain that the MQSeries batch interface runs under a special transaction MQBI. This transaction may be added to the DFHPLTPI phase of CICS. If you stop MQSeries from the Initialize/Shutdown of System screen, the batch interface transaction does not terminate unless the batch interface auto start flag in the global definitions was Y. If this is not set to `Y` it is not necessarily a problem. When you restart MQSeries, the batch interface should still work without it needing to be stopped and restarted.

3.5 Maintain queue message records

As an administrative task, it may be necessary to clear out obsolete messages from queues or reset messages for reprocessing. These tasks can be achieved from the Maintain Queue Message Records screen as shown in Figure 3-4.



```
10/13/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:26:21           Maintain Queue Message Records                  CIC1
MQWMDEL                                                            A001
                          System Information
         System Status    : System is active.
         Queue  Status    : Queuing system is active.
         Channel System   : Channel system is active.
                       Queue Information
         Queue Name       : SYSTEM.LOG
         Function         : A A=Delete all
                              D=Delete to date/time exclusive
                              R=Reset from date/time inclusive
         Date (yyyymmdd)  :
         Time (hhmmss)    :

                       Results of Request
         Number Processed :
         Number of Bypass :
         New Last Read QSN:
         Process Time     :

Queue processing not pending. Press UPDATE to start update.
Enter=Refresh  PF2=Return  PF3=Exit  PF6=Update
                                         PF12=Retry
```

Figure 3-4 Maintain Queue Message Records screen

From this screen, you can do the following:

- Logically delete messages from queues
- Reset them to the written status so that they can be reprocessed by application programs issuing MQGET calls against the specified queue
- Physically delete all records for the queue

Remember that MQSeries for VSE logically deletes messages once they have been retrieved from a queue. Messages are not actually deleted from the VSAM file that hosts them until a VSAM reorganization occurs. A message has a `deleted` status when it is logically deleted, otherwise its status is `written`. You can view the status of messages using MQMT option 4.

MQSeries for VSE uses a Queue Sequence Number to individuate messages. The QSN is an incremental numeric value that forms part of the KSDS key when a message is stored in a VSAM file. When you specify `D` (Delete), the Queue Sequence Number (QSN) does not change. This is quite different from the MQPREORG program which reorganizes messages into sequential QSN order. In addition, from a VSAM point of view, the deleted records are not accessible anymore, but the space is not reclaimed. This means you cannot use this function to compress the KSDS data.

When you specify `A` (Delete all), it is not the same as starting a IDCAMS DELETE/DEFINE batch job. When you use this dialog to delete all messages from a queue, VSAM deletes all records, but it keeps the index structure and the secondary extents. That is, if you had 200 secondary allocations, using the A option keeps all these extents for your VSAM cluster. This could explain why you may have bad response times when opening and closing a VSAM cluster even when there are no records in it. Therefore, we recommend that you use the MQPREORG utility or the automatic reorganization feature to regularly delete and redefine the VSAM data sets that host your queues.

4

Monitoring

Monitoring MQSeries objects is an important and ongoing administrative task. By using online dialogs available with the MQSeries Master Terminal Transaction (MQMT), you can monitor MQSeries objects. For example:

- Application queues
- Channels
- Messages
- System queues

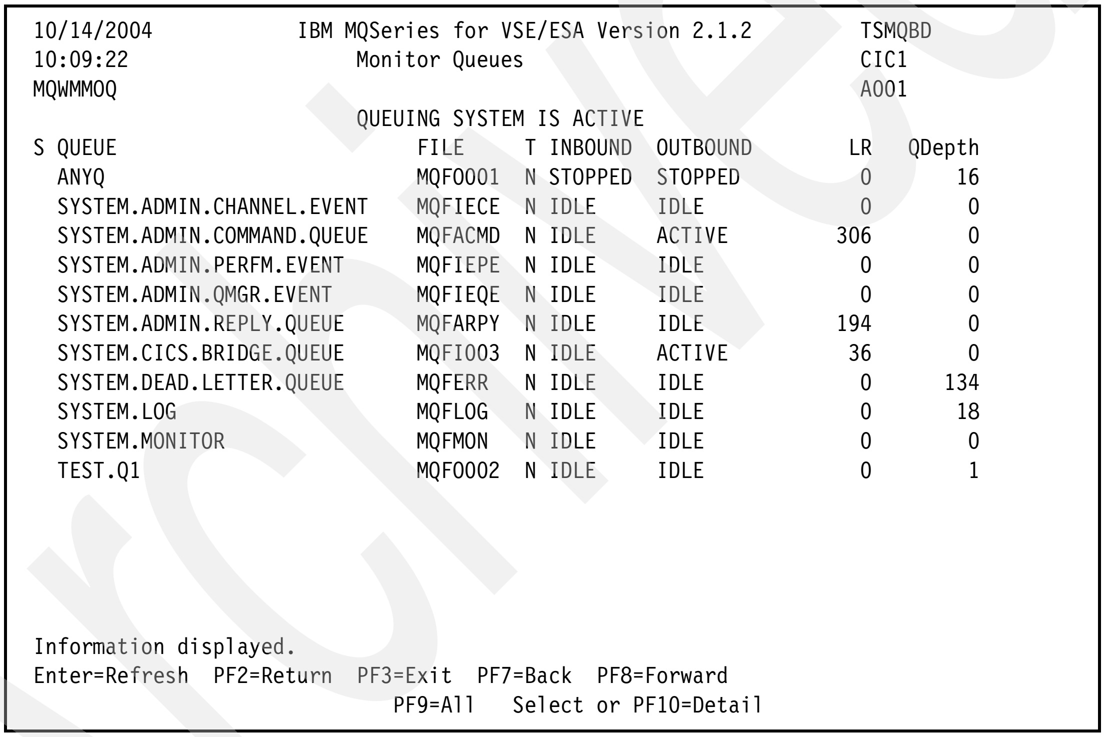
4.1 Monitoring application queues

You may want to monitor your application queues for many reasons, for example:

- To check whether messages sent by a remote system arrived correctly.
- To determine whether all messages you sent were received by a partner. That is, the count of a transmission queue is 0.
- To check that there are not too many logically deleted messages and to decide whether it is time to reorganize a queue.
- To examine the current status of a queue.

You can monitor the status of queues using MQMT option 3.1. MQSeries must be active to use this option.

An example of the Monitor Queues screen is shown in Figure 4-1.



```
10/14/2004           IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
10:09:22                 Monitor Queues                             CIC1
MQWMMOQ                                                             A001
                         QUEUING SYSTEM IS ACTIVE
S QUEUE                       FILE     T INBOUND  OUTBOUND        LR   QDepth
  ANYQ                        MQFO001  N STOPPED  STOPPED          0       16
  SYSTEM.ADMIN.CHANNEL.EVENT  MQFIECE  N IDLE     IDLE             0        0
  SYSTEM.ADMIN.COMMAND.QUEUE  MQFACMD  N IDLE     ACTIVE         306        0
  SYSTEM.ADMIN.PERFM.EVENT    MQFIEPE  N IDLE     IDLE             0        0
  SYSTEM.ADMIN.QMGR.EVENT     MQFIEQE  N IDLE     IDLE             0        0
  SYSTEM.ADMIN.REPLY.QUEUE    MQFARPY  N IDLE     IDLE           194        0
  SYSTEM.CICS.BRIDGE.QUEUE    MQFI003  N IDLE     ACTIVE          36        0
  SYSTEM.DEAD.LETTER.QUEUE    MQFERR   N IDLE     IDLE             0      134
  SYSTEM.LOG                  MQFLOG   N IDLE     IDLE             0       18
  SYSTEM.MONITOR              MQFMON   N IDLE     IDLE             0        0
  TEST.Q1                     MQFO002  N IDLE     IDLE             0        1





Information displayed.
Enter=Refresh  PF2=Return  PF3=Exit  PF7=Back  PF8=Forward
                              PF9=All   Select or PF10=Detail
```

Figure 4-1 Monitor Queues screen

The various fields of the Monitor Queues screen have the following meanings:

- QDepth

  The *QDepth* shows the current number of unread messages on a queue, including uncommitted messages being put to the queue. It does not show uncommitted messages currently being retrieved from the queue.

  This situation exists because MQSeries for VSE updates its internal counters when it detects that an application has syncpointed work. This is only possible on a subsequent MQI call following a SYNCPOINT call. Consequently, there is a brief window when work is actually committed but MQSeries internal counters are not yet updated.

  Remember also that a QDepth of 0 does not necessarily mean the VSAM file that hosts the queue is empty. Messages that have been retrieved from the queue using MQGET are actually retained in the queue with a logically deleted status. Such processed messages can only be removed with one of the various reorganization processes.

- LR

  As already mentioned, MQSeries for VSE uses an incremental numeric value to individuate messages. The incremental numeric is called a queue sequence number (QSN). The QSN is used as part of the key for the message when it is written to the KSDS VSAM file that hosts the queue.

  The *LR* value stand for "last read." This is misleading. It is really an internal value that identifies the starting point for finding the lowest QSN of messages yet to be retrieved from the queue. If a queue is processed sequentially, as most queues are, the LR value may represent the number of logically deleted messages on the queue.

  If you are monitoring transmission queues, and the queue has a non-0 queue depth, you can expect the LR value to steadily increase. If it does not increase, there is a problem which is probably caused because:

  – You explicitly closed the queue.

  – An application issued a MQSET to disable MQGET calls to the queue.

  – You explicitly closed the corresponding channel.

  – Your remote partner is not reachable. In most cases, this is the reason a transmission queue does not have an LR count of zero.

- Inbound/Outbound status

  The content of this field normally toggles between ACTIVE and IDLE. The ACTIVE status means at least one MQOPEN has been issued against the queue and is currently active. Conversely, the status is IDLE when no MQOPEN is currently active against the queue. The status may be STOPPED if you close the queue manually or if you defined your queue with the inbound or outbound status inactive.

  Take care that you do not confuse the status as displayed by this screen and your original queue definitions. Table 4-1 shows various combinations.

Table 4-1 Inbound and outbound queue definitions and statuses

Queue In/Out	**Monitor Status**	**Description**
ACTIVE	IDLE	Queue is available, but no MQOPEN is currently active.
ACTIVE	ACTIVE	Queue is available and at least one MQOPEN is active against the queue.
ACTIVE	STOPPED	Queue is unavailable due to manual request to stop the queue.
INACTIVE	STOPPED	Queue is unavailable by definition.

  However, even if a queue is IDLE, you may not be authorized to put to or get from the queue. This depends on what you have defined for the Put Enabled and Get Enabled respectively. You can verify this by pressing **PF10** (Detail) from the Monitor Queues screen.

  Other Inbound/Outbound values correspond to special situations or error conditions. Possibilities include:

  – MAX

    The queue has reached a queue depth matching the value you specified in the *Max QDepth* for the queue.

– FULL

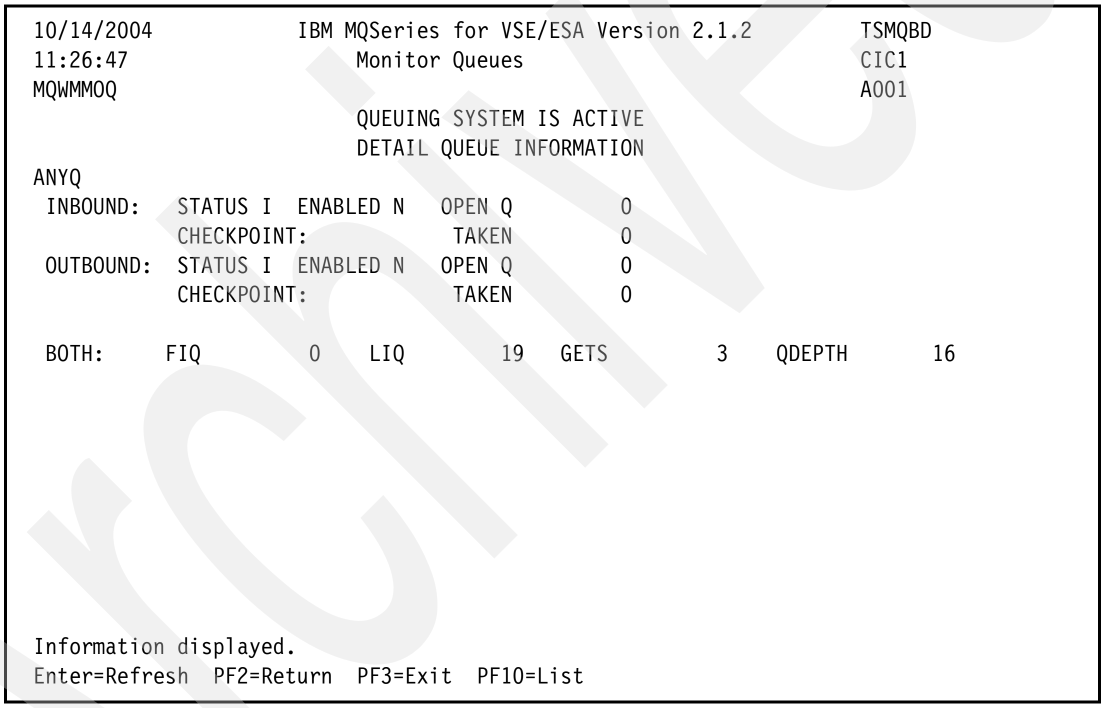
The VSAM cluster hosting this queue is full. This may happen even if all messages have a DELETED status. The best solution is to reorganize the queue using either the automatic reorganization feature or the batch program MQPREORG. For details on this program, refer to the *MQSeries for VSE System Management Guide*, GC34-5364.

– RECOVERY

If a local queue has been defined with a dual queue, the dual queue must be an exact replica of the primary queue to takeover in case of a primary queue failure. If this is not the case, the dual queue is flagged with the RECOVERY status. If your global system definition includes a nonzero *Max Recover Tasks* value, then a recovery transaction is started automatically for the dual queue.

From the Monitor Queues screen, you can move the cursor to a queue entry and press **PF10** for more detailed information relevant to the selected queue.

An example of the Monitor Queues: Detail Queue Information screen is shown in Figure 4-2.



```
10/14/2004          IBM MQSeries for VSE/ESA Version 2.1.2        TSMQBD
11:26:47                    Monitor Queues                        CIC1
MQWMMOQ                                                           A001
                            QUEUING SYSTEM IS ACTIVE
                            DETAIL QUEUE INFORMATION
ANYQ
 INBOUND:   STATUS I  ENABLED N   OPEN Q        0
            CHECKPOINT:            TAKEN        0
 OUTBOUND:  STATUS I  ENABLED N   OPEN Q        0
            CHECKPOINT:            TAKEN        0

 BOTH:     FIQ        0    LIQ       19   GETS         3    QDEPTH       16









Information displayed.
Enter=Refresh  PF2=Return  PF3=Exit  PF10=List
```

Figure 4-2 Monitor Queues: Detail Queue Information screen

The various fields of the Monitor Queues: Detail Queue Information screen have the following meaning:

- STATUS

  This field may be: I (Idle), B (Busy, same as Active), or S (Stopped).

- ENABLED

  Indicates whether I/O operations are permitted for the queue. If Y (yes) for INBOUND, then MQPUTs are possible. If Y (yes) for OUTBOUND, MQGETs are possible. An easy way to remember this relationship is to think *IN*BOUND in terms of PUT *IN*, and *OUT*BOUND in terms of GET *OUT*.

- OPEN Q

  This value shows the number of tasks currently working with this queue. There are two counters, one for INBOUND MQOPENs and one for OUTBOUND MQOPENs. An MQOPEN is considered INBOUND if the open is to put messages, and OUTBOUND if the open is to get messages.

- CHECKPOINT: TAKEN

  The *Checkpoint: Taken* value has nothing to do with syncpointing. It is, in fact, an internal counter used to keep track of the first available QSN for an MQGET call and the next available QSN for an MQPUT or MQPUT1 call. MQSeries updates a "checkpoint" record to keep track of these values.

  This value is not useful for system administration and should not be used. It will be removed from the monitor queues display at a later date.

- FIQ

  This value stands for First In Queue. This value normally should be 0 when the queue is empty. It corresponds to the Queue Sequence Number (QSN) of the first message in queue.

- LIQ

  LIQ stands for Last In Queue. It represents the QSN of the last message put to the queue. It can also be interpreted as the number of messages on the queue, that is, the total of unread, logically deleted, and rolled back messages. A high number indicates that there are a significant number of messages to process, or that the hosting VSAM file needs a reorganization.

- GETS

  This value represents the number of messages between the FIQ and LIQ that have already been retrieved or were written and then rolled back.

4.2 Monitoring channels

Monitoring channels also provides you with a lot of useful information about your MQSeries system behavior. Channels are monitored from MQMT option 3.2.

An example of the Monitor Channels screen is shown in Figure 4-3 on page 70.

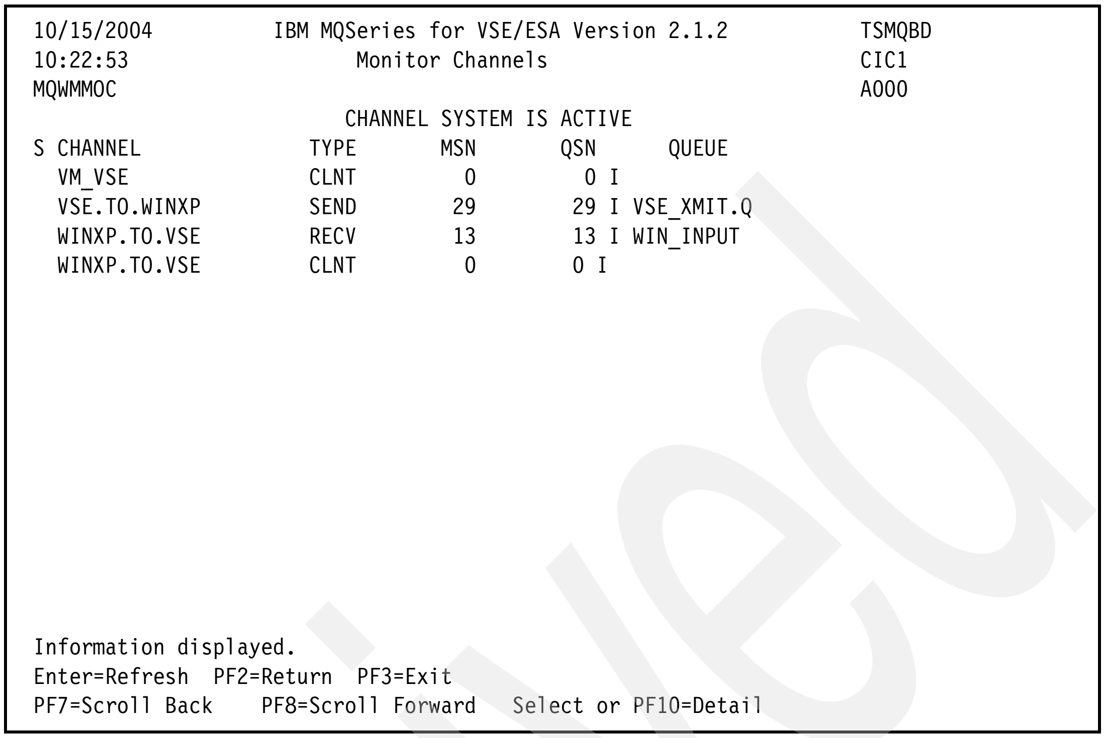

```
10/15/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
10:22:53                   Monitor Channels                          CIC1
MQWMMOC                                                              A000
                          CHANNEL SYSTEM IS ACTIVE
S CHANNEL               TYPE       MSN       QSN      QUEUE
  VM_VSE                CLNT         0         0 I
  VSE.TO.WINXP          SEND        29        29 I VSE_XMIT.Q
  WINXP.TO.VSE          RECV        13        13 I WIN_INPUT
  WINXP.TO.VSE          CLNT         0        0 I











Information displayed.
Enter=Refresh  PF2=Return  PF3=Exit
PF7=Scroll Back    PF8=Scroll Forward   Select or PF10=Detail
```

Figure 4-3   Monitor Channels screen

This screen shows the entire list of your channel definitions. At first glance, you can see whether a channel is a sender (SEND), a receiver (RECV), or a client (CLNT). In this example, we have defined channels to connect to a Windows XP MQSeries queue manager (peer-to-peer mode). As we previously said, a channel is a unidirectional path between two partners, and therefore, we need two channels to send and receive messages in both directions. We also defined two client channels (client/server mode), one connected to a Windows XP workstation and the other one to a VM/CMS virtual machine.

Things to remember:

- Since Release 2.3, the MQSeries client code is shipped as a standard feature of VM/ESA®.
- VSE client channels work only with TCP/IP.

With MQ clients, there is no queue associated with the channel. This is because a client application is mirrored by a local VSE server program. Clients do not transmit messages; they issue MQI calls as though they were running in the CICS partition. Channels for peer-to-peer connections have associated queues: a *transmission queue* for the sender channel, and a *local queue* for the receiver channel (which is the remote queue for the partner). For receiver channels, the associated queue is the last queue used by the receiver MCA. For example, if you have two remote queue definitions that identify two different local queues, but the same transmission queue, the Monitor Channels screen displays the local queue last used by the receiver MCA.

The following points describe the meaning of some of the fields on the Monitor Channels screen:

- MSN and QSN

  On this panel, the Message Sequence Number (MSN) and the Queue Sequence Number (QSN) have the same value. Usually, this is not the case.

  The QSN is concatenated with the queue name and a 4-byte descriptor field to create a unique message key for the VSAM cluster. When you reorganize a queue, the QSN is reset to 1. A reorganize involves a delete and redefine of the VSAM file. MQSeries keeps track of QSNs for local queues by recording the current QSN for each queue in a header record. The header record is written to the VSAM file of the queue. In the event of a reorganization, the header record is lost. Consequently, MQSeries rewrites queue header records, resetting the QSN to 1, when MQSeries is restarted and the queue is initially opened.

  QSNs are only relevant to the local queue manager for building unique keys. The QSN is not used for synchronization between MCAs.

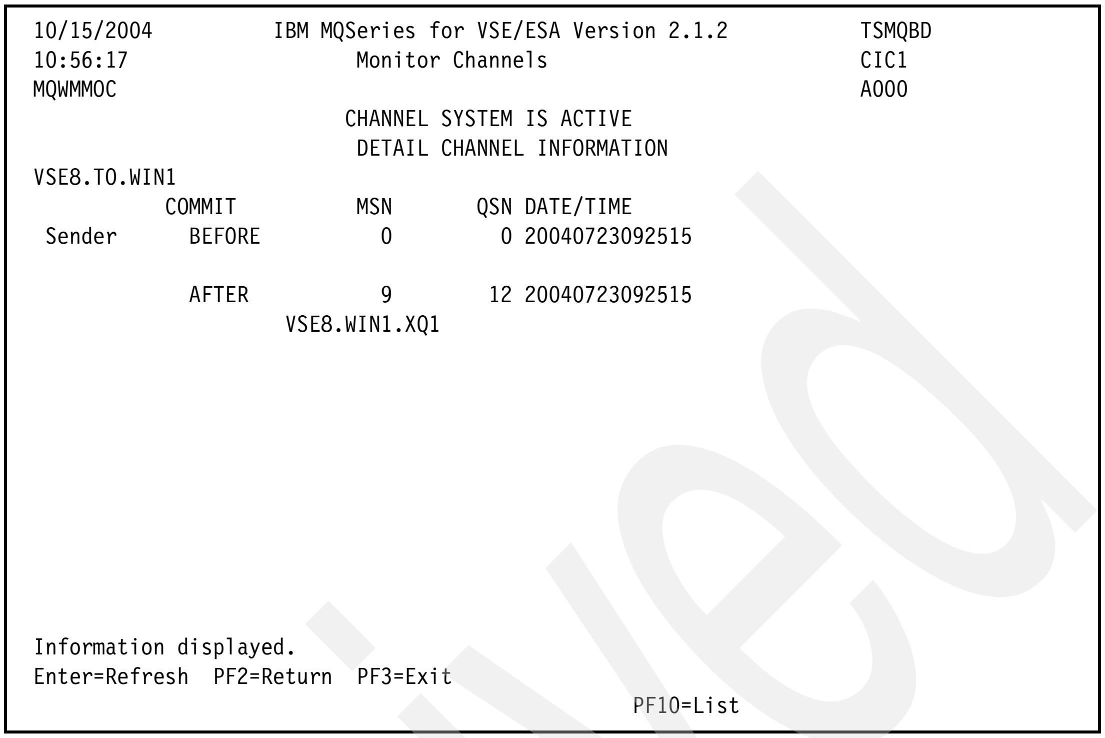
  The MSN, on the other hand, is a number that is used by both partners to make sure no messages are lost. MSNs are not associated with a queue. Instead, they are associated with a channel and are maintained in the MQSeries configuration file. When channels are opened, the last MSN value is retrieved. If, for any reason, you have to reset an MSN to 1 or any other value, for an MQ/VSE receiver channel the same operation must also be done at the partner side. Otherwise, the initial MCA negotiation would fail. When the MSN reaches the value you specified in the field *Message Sequence Wrap* at configuration time, both partners restart the MSN from 1. For an MQ/VSE sender channel, the new value is sent to the remote receiver following negotiation and before sending messages.

  There is a trap that you should be aware of. If you reset the MSN value on, for example, UNIX or Windows XP platforms, the new sequence number is set to 0. However, from the CICS side (VSE or OS/390 platforms), you must specify 1, because for S/390® systems it means *next* and on other platforms, it means *current*.

  The MQSeries use of MSNs is intended to ensure message integrity. Consider the following scenario:

  > You have two (or more) remote queue definitions on MQSeries system A that identify two different local queues on MQSeries system B. The two remote queue definitions may identify the same transmission queue and, consequently, the same channel for distributed queuing. An application program may write 100 messages into the first remote queue in batches of 10 messages. In a worst-case scenario, on the last batch, the receiver MCA may commit the final batch and abend before it acknowledged receipt of the batch. In this case, the sender has an MSN of 90 and the receiver 100. Now, a second application program begins to write 100 messages to the second remote queue. This is not a problem. The 10 messages from the last batch of the first 100 messages are still on the transmission queue and these are sent to the receiver MCA *before* the new 100 messages. The receiver MCA can ignore the first 10 messages because it knows they have already been processed. It then accepts the subsequent 100 messages and writes them to the second local queue. The final result is that each local queue has received 100 messages and the current MSN for the channel rests at 200.

From the Monitor Channels screen, you can select a channel with the cursor and press **PF10** (Detail) to display the Monitor Channels: Detail Channel Information screen. An example of this screen is shown in Figure 4-4 on page 72.



```
 10/15/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
 10:56:17                   Monitor Channels                          CIC1
 MQWMMOC                                                              A000
                           CHANNEL SYSTEM IS ACTIVE
                            DETAIL CHANNEL INFORMATION
 VSE8.TO.WIN1
            COMMIT          MSN       QSN DATE/TIME
  Sender      BEFORE          0         0 20040723092515

              AFTER           9        12 20040723092515
                        VSE8.WIN1.XQ1








 Information displayed.
 Enter=Refresh  PF2=Return  PF3=Exit
                                                  PF10=List
```

Figure 4-4 Monitor Channels: Detail Channel Information screen

The detailed channel information includes commit before and after values for MSNs. When two partners are exchanging messages, and when these messages are written to a queue, the VSE receiver MCA issues a CICS syncpoint (synonymous with commit) upon receiving confirm request from the remote sender. It then sends an acknowledgment to the partner (sender MCA) which subsequently deletes the last successful batch of messages from its transmission queue and issues a CICS syncpoint. In V1.4, a syncpoint was issued for each message sent or received. With V2.1, you may decide on the number of messages before a syncpoint is taken. You specify this number when configuring the channel with the Max Messages per Batch value. Syncpoints take place:

- When the last message from the transmission queue has been sent or received.
- When the number of messages sent or received reaches the Max Messages per Batch value.

The commit before value represents the MSN at the time of commit before the next commit. The commit after value represents the current MSN after the last commit. Perhaps a better way of describing these values is:

- Commit before

  MSN at last commit
- Commit after

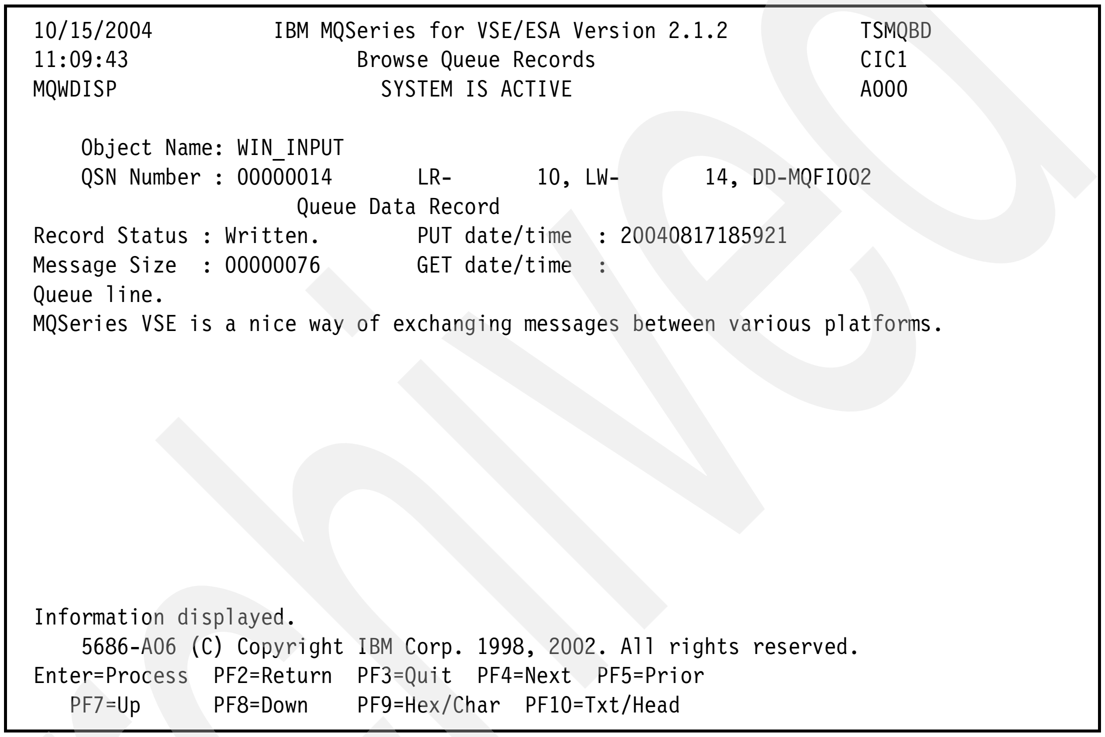
  Current MSN since last commit

The same applies for QSNs relative to commits.

4.3 Monitoring messages

Monitoring queues and channels does not help you know whether your application programs are working or whether the content of messages is what you were expecting. A good starting point, before using debug or trace tools, is to examine messages directly using MQMT option 4, Browse Queue Records.

In the following example, we received messages from a Windows XP system that were written to queue WIN_INPUT. An example of the Browse Queue Records screen for one of these messages is shown in Figure 4-5.



```
10/15/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:09:43                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: WIN_INPUT
    QSN Number : 00000014       LR-       10, LW-       14, DD-MQFI002
                       Queue Data Record
Record Status : Written.        PUT date/time  : 20040817185921
Message Size  : 00000076        GET date/time  :
Queue line.
MQSeries VSE is a nice way of exchanging messages between various platforms.









Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 4-5 Browse Queue Records screen

From this screen, you can browse backward and forward using **PF5** and **PF4** respectively. You can also overtype the QSN number to go directly to a particular message. You can also go directly to the LW message by press the **PF1** key.

The following points may be helpful when using the Browse Queue Records screen:

- LR

  LR stands for "last read," but, like the LR value on the monitor queues screen, this is misleading. The LR value actually represents the lowest QSN value that has been read or rolled back from the queue. In the above example, the LR is 10, which means the next message available is the message with a QSN of 11. If the queue is processed sequentially, you can assume that messages keyed with QSNs 1 to 10 have all been logically deleted or put and subsequently rolled back.

  This option, MQMT option 4, allows you to browse unread and logically deleted messages.

- LW

  LW stands for "last written." The LW is the QSN of the last message on the queue. The LW value does not necessarily represent the queue depth. The queue depth is calculated by subtracting the LR from the LW, and then subtracting the number of messages that may have been read randomly or rolled back. The QSNs of messages that have been read randomly or rolled back are stored in an MQSeries internal table called the global lock table. Each queue has its own global lock table and MQSeries keeps it up to date as application units of work are committed.

- Message Size

  The message size represents the length of the data portion of a message. It does not include the length of the MQSeries message header.

- PUT date/time and GET date/time

  These values are expressed in the format `yyyymmddhhmmss`. The difference between the *PUT date/time* and *GET date/time* values can provide an idea of the elapsed time your system needed to process a message. Of course, if messages are not yet processed, the *GET date/time* field is blank. This is the local system's time and not GMT. This PUT time stamp is used to determine message expiry if this has been set in the message.

- Queue line

  The detail below the Queue line heading is the content of the message itself. You can use the **PF9** key to toggle between character and hexadecimal displays of the data. You can page up and down through the data using **PF7** and **PF8**, but only the first 3352 bytes can be browsed.

  You will notice that the message data in our example is in EBCDIC even though it was sent from an ASCII queue manager. This is because the sender channel on the remote system is configured to convert message data before it is sent. If the channel was not configured to convert message data, the data would have been received in ASCII, and it would be up to the application that retrieves the message from the queue to convert the data.

  Applications can either do the conversion themselves, or use the MQSeries data conversion facility available with the MQGET call. Data conversion can be a complex issue if you are dealing with a mixture of integers, strings and binary data. The topic of data conversion is covered in more detail in section 6.6.1, “Client code page conversion” on page 119.

You can toggle between character and hexadecimal display by pressing **PF9** (Hex/Char). You can also toggle from the data to the header part by pressing **PF10** (Txt/Head).

An example of a message header is shown in Figure 4-6 on page 75.

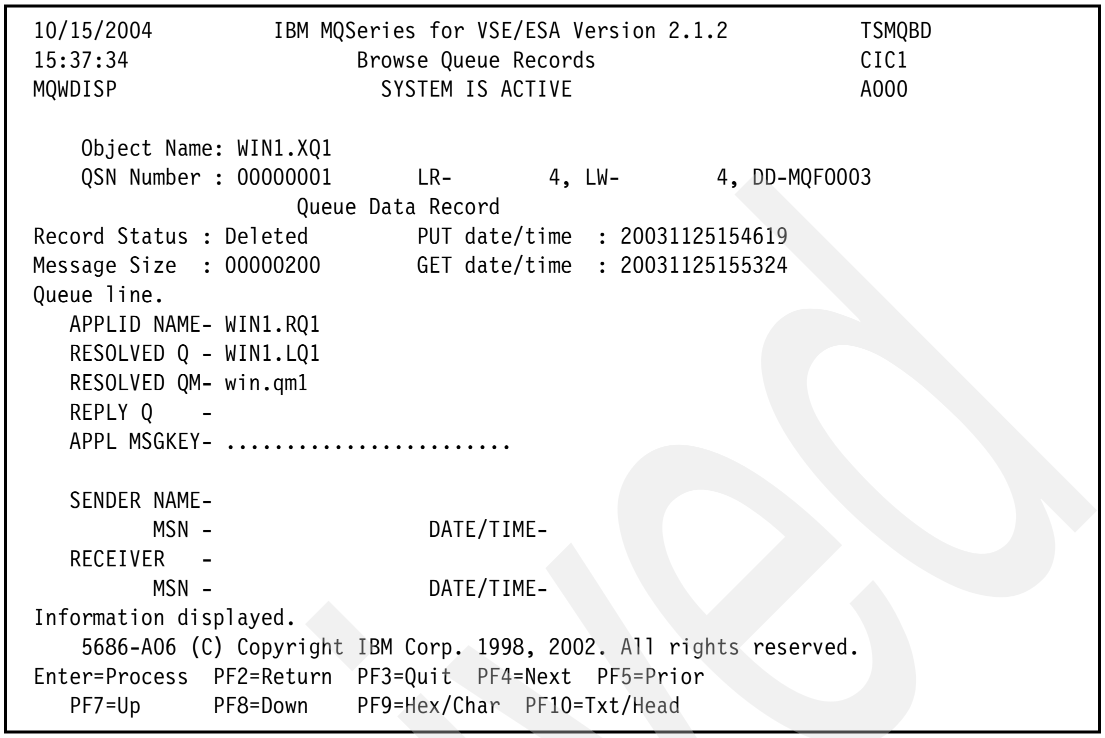

```
10/15/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:37:34                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: WIN1.XQ1
    QSN Number : 00000001       LR-        4, LW-       4, DD-MQF0003
                      Queue Data Record
Record Status : Deleted         PUT date/time  : 20031125154619
Message Size  : 00000200        GET date/time  : 20031125155324
Queue line.
   APPLID NAME- WIN1.RQ1
   RESOLVED Q - WIN1.LQ1
   RESOLVED QM- win.qm1
   REPLY Q    -
   APPL MSGKEY- ........................

   SENDER NAME-
          MSN -                  DATE/TIME-
   RECEIVER   -
          MSN -                  DATE/TIME-
Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 4-6 Browse Queue Records: Message header screen

Values on this screen have the following meaning:

- APPLID NAME

  The name of this field is quite misleading because it has nothing to do with the CICS applid (from DFHSIT) or a CICS application program. It simply refers to the queue name used in an MQOPEN or MQPUT1 call prior to putting the message to the queue. It could be the actual queue name or an alias queue name.

- RESOLVED Q and QM

  This refers to the actual name of the queue and queue manager. Consequently, if the APPLID NAME is an alias queue, the RESOLVED Q would be the actual queue name associated with the alias.

- REPLY Q

  This value refers to the reply queue name passed as a parameter of an MQPUT or MQPUT1 call (if any). Reply queues are used for MQSeries acknowledgment of the arrival and delivery of messages. Arrival occurs when a message is first put to a queue. Delivery occurs when a message is read from a queue.

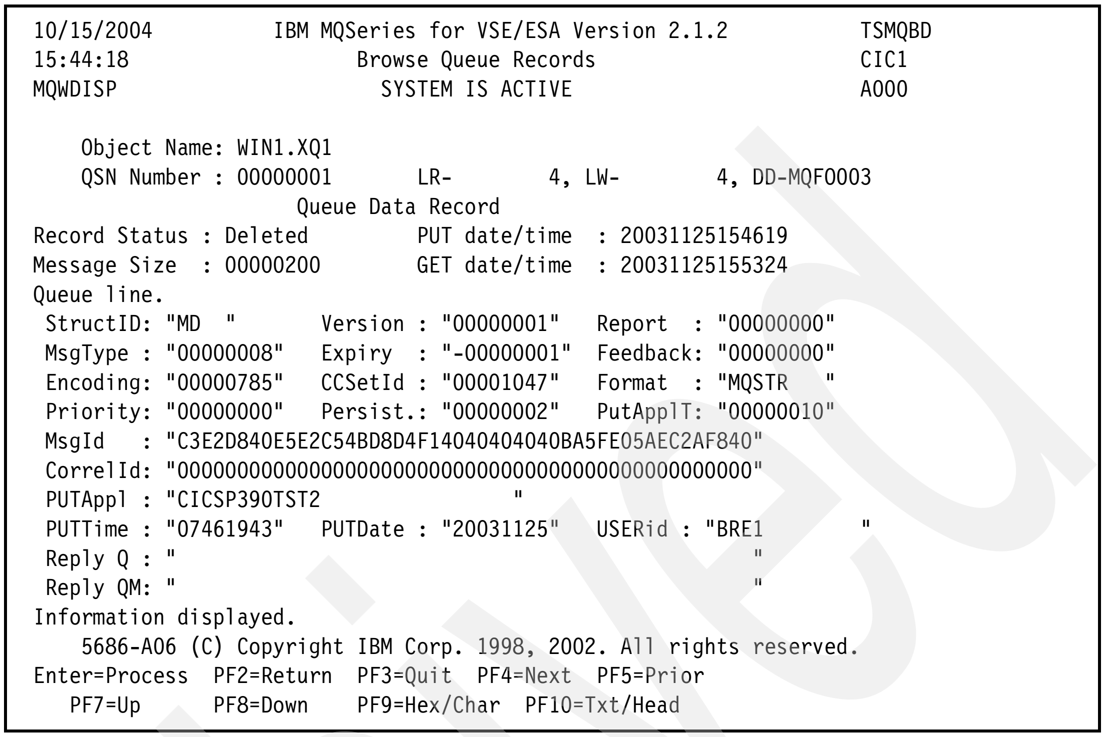
  For details about reply queues, refer to 8.5, “Application design and reply-to queues” on page 162.

- APPL MSGKEY

  The name of this field may also be confusing. It refers to the MsgId parameter passed of an MQPUT or MQPUT1 call.

Although it is not documented in the *MQSeries for VSE System Management Guide*, GC34-5364, you can get additional information by pressing **PF11**. Using **PF11** from the header display displays the Message Descriptor (MD). The MD is a structure passed as a parameter to MQPUT and MQPUT1 calls.

An example of the message descriptor display is shown in Figure 4-7.



```
10/15/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:44:18                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: WIN1.XQ1
    QSN Number : 00000001       LR-        4, LW-        4, DD-MQF0003
                      Queue Data Record
Record Status : Deleted         PUT date/time  : 20031125154619
Message Size  : 00000200        GET date/time  : 20031125155324
Queue line.
 StructID: "MD  "       Version : "00000001"   Report  : "00000000"
 MsgType : "00000008"   Expiry  : "-00000001"  Feedback: "00000000"
 Encoding: "00000785"   CCSetId : "00001047"   Format  : "MQSTR   "
 Priority: "00000000"   Persist.: "00000002"   PutApplT: "00000010"
 MsgId   : "C3E2D840E5E2C54BD8D4F14040404040BA5FE05AEC2AF840"
 CorrelId: "000000000000000000000000000000000000000000000000"
 PUTAppl : "CICSP390TST2                "
 PUTTime : "07461943"   PUTDate : "20031125"   USERid : "BRE1        "
 Reply Q : "                                                "
 Reply QM: "                                                "
Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 4-7 Browse Queue Records: Message descriptor screen

Most of the fields on this screen are useful only for IBM internal use. However, the PUTAppl may be of interest. PUTAppl describes the program that issued the MQPUT or MQPUT1 call. When issued from a VSE MQSeries system, it is the CICS applid (from DFHSIT) concatenated with the CICS application transaction. In our example, the CICS applid is CICSP390 and the transaction that issued the call was TST2.

If the MQPUT or MQPUT1 call was issued from a remote system such as Windows XP, this field would contain the path of the application program, for example, C:\MQM\BIN\AMQSPUT.EXE.

4.4 Monitoring system queues

MQSeries for VSE includes the following system queues:

- System log
- System monitor
- System dead letter queue
- System command queue
- System command reply queue
- Queue manager event queue
- Channel event queue
- Performance event queue

4.4.1 System log

The system log is used to record messages related to system activity. These include both diagnostic and error messages. Messages written to the system log queue have a unique number and severity code. Messages by number are described in the *MQSeries for VSE System Management Guide*, GC34-5364.

The types of message written to the system log can be controlled using the log and trace settings of the queue manager. These are accessible using MQMT option 1.1 and **PF10**. For example, you can suppress diagnostic or informational messages, and have only warning, error and critical messages written to the log.

If for any reason the behavior of your MQSeries system does not work as you expect it should (for example, if application messages are not sent or received as expected), you can browse the tail of the system log for possible error messages. If MQSeries has detected an error, messages in the system log should describe the reason for the failure, unless they are suppressed by your log and trace settings. To view the tail end of the system log, use MQMT option 4, clear the queue name, and press **PF1**. Using **PF1** when no queue name is specified defaults to SYSTEM.LOG and displays the last message in the queue (that is, the latest message).

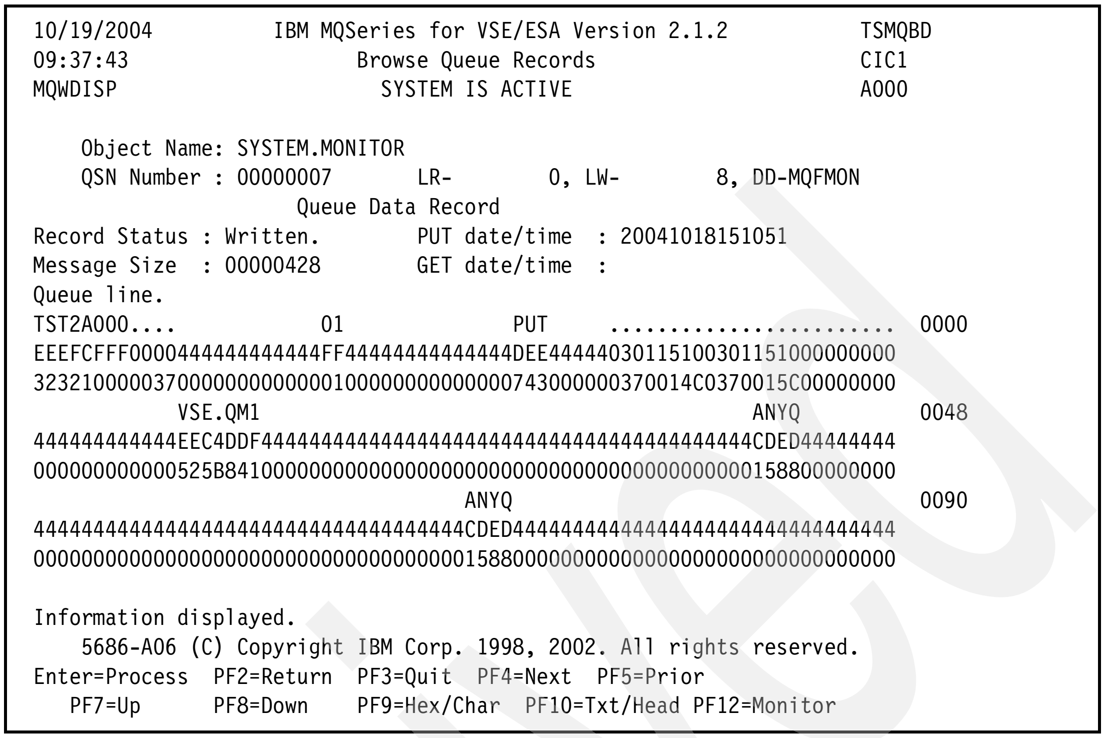
Unless messages are totally suppressed by your log and trace queue manager settings, your system log will slowly fill up with system activity messages. Therefore, you should set the maximum queue depth to an appropriate value, and ensure that the VSAM file that hosts the system log queue (usually MQFLOG) is periodically reorganized.

If messages cannot be written to the system log for any reason, messages are instead written to the CICS CSMT transient data queue. However, messages are only written to the CSMT queue if the queue manager parameter *Allow TDQ Write on Errors* is set to Y.

4.4.2 System monitor

The name of this queue can be misleading. It has nothing to do with the MQSeries System Monitor task, MQSM. It contains only trace records for MQSeries MQI calls issued by both MQSeries and application programs. A better name might have been the MQI Trace queue.

An example of the system monitor queue, as displayed from MQMT option 4, Browse Queue Records, is shown in Figure 4-8 on page 78.



```
10/19/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
09:37:43                   Browse Queue Records                      CIC1
MQWDISP                      SYSTEM IS ACTIVE                        A000

    Object Name: SYSTEM.MONITOR
    QSN Number : 00000007       LR-        0, LW-       8, DD-MQFMON
                      Queue Data Record
Record Status : Written.        PUT date/time  : 20041018151051
Message Size  : 00000428        GET date/time  :
Queue line.
TST2A000....            01              PUT     .......................  0000
EEEFCFFF0000444444444444FF44444444444444DEE444440301151003011510000000000
323210000037000000000000001000000000000000743000000370014C037001 5C00000000
            VSE.QM1                                         ANYQ         0048
444444444444EEC4DDF44444444444444444444444444444444444444444CDED44444444
000000000000525B84100000000000000000000000000000000000000000158800000000
                                    ANYQ                                 0090
444444444444444444444444444444444444CDED4444444444444444444444444444444444
000000000000000000000000000000000000158800000000000000000000000000000000

Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head PF12=Monitor
```

Figure 4-8 System monitor message

Unfortunately, there is no way to just trace at function, transaction, or program level. When the trace is activated, all MQSeries MQI calls are recorded. Therefore, we recommend you use this capability only in a test environment and for a very short period of time. Otherwise, the overall performance of your CICS partition could suffer considerably.

To enable the trace function, use MQMT option 2 (Operations) and suboption 1 (Start/Stop Queues). This displays the Start/Stop Queues screen. The last field on this screen, Function, allows you to select **M** (for Monitor) and press **PF6** (Update). This activates the MQI trace facility, and all subsequent MQI calls are logged to the system monitor queue. Tracing continues until you stop MQSeries or reset active tracing via MQMT option 2.1. You can browse the Monitor Queue normally using MQMT option 4, Browse Queue Records.

Keep in mind that records are written after the MQI calls have been executed. Therefore, you can easily check the values of condition or reason codes (offset x’40’ and x’44’, respectively). The structure of the trace records is described in Table 11-1 on page 203.

4.4.3 System dead letter queue

The dead letter queue is not really a system queue. It is a normal local queue. As previously explained, it is needed to store all messages that cannot be either sent to an unknown queue manager or written to an unknown local queue.

It is your responsibility to include a mechanism for checking the dead letter queue and dealing with undelivered messages in your overall application design. To this end, you may consider the use of queue triggers.

Note: A sample program to handle the Dead Letter Queue (DLQ) can be downloaded from the following Web site:

`http://www.ibm.com/software/ts/mqseries/platforms/vseesa/`

The sample program is available as a freeware "as-is" SupportPac.

4.4.4 System command queue

The system command queue exists to accept local or remote PCF command request messages. PCF stands for Programmable Command Format. A PCF message is a binary message that defines a command for the queue manager to execute. For example, a PCF message may be a request to change the value of a queue, define a queue, delete a channel, or modify the queue manager. The structure and scope of PCF messages is described in the *MQSeries for VSE System Management Guide*, GC34-5364.

The default name of the system command queue is SYSTEM.ADMIN.COMMAND.QUEUE. For MQSeries for VSE, this name is configurable as part of the communication settings of the queue manager. Use MQMT option 1.1, and **PF9** to view your communication settings.

Messages that arrive on the system command queue are processed by the MQSeries command server transaction (MQCS), if it is running. The result of processing the PCF command is sent to the reply-to queue in the original request.

For more information about the system command queue and system administration using PCF commands, refer to Chapter 5, "System administration" on page 81.

4.4.5 System command reply queue

The system command reply queue is unique to MQSeries for VSE. It exists to support MQSeries Commands (MQSC) issued from batch using the MQPMQSC command utility. MQSC commands are verb-based requests equivalent to the PCF binary requests. For example, from a batch program using the MQPMQSC utility you can use verb commands such as DEFINE, ALTER, DELETE, and DISPLAY to manipulate MQSeries objects controlled by the queue manager. The syntax and scope of MQSC commands is described in the *MQSeries for VSE System Management Guide*, GC34-5364.

The default name of the system command reply queue is SYSTEM.ADMIN.REPLY.QUEUE. This name is configurable as part of the communication settings of the queue manager. Use MQMT option 1.1, and **PF9** to view your communication settings.

The MQPMQSC utility uses the MQSeries Batch Interface to send MQSC commands to the system command queue (as PCF Escape messages), and waits for responses on the system command reply queue.

For more information about the system command reply queue and MQSeries Commands, refer to Chapter 5, "System administration" on page 81.

4.4.6 Event queues

The event queues include the queue manager, channel, and performance event queues. The default names for these queues, respectively, are:

```
SYSTEM.ADMIN.QMGR.EVENT
SYSTEM.ADMIN.CHANNEL.EVENT
SYSTEM.ADMIN.PERFM.EVENT
```

The event queue names are configurable as part of the queue manager's event settings, accessible using MQMT option 1.1, and **PF11**.

If you activate the generation of instrumentation event messages, when such events occur, they are written to the relevant event queue. Instrumentation events can be activated via the queue manager's event settings. For events relating to queues, the relevant events must also be activated as part of the queue definition. Such events are only relevant to local queues.

If you plan to activate instrumentation events and to use the event queues, the installation sample JCL files MQJQUEUE.Z and MQCICFCT.Z (or MQJCSD24.Z for CICS TS) contain default definitions for the necessary VSAM and CICS resources.

Events and event queues are described in greater detail in Chapter 5, “System administration” on page 81.

5

System administration

In this chapter, we describe system administration for MQSeries for VSE, and provide examples of how you might implement local or remote management of your VSE queue manager.

System administration involves controlling and monitoring your queue manager and its objects, such as channels and queues. Your queue manager can be administered locally or remotely by administrative applications that use programmable interfaces such as PCF, or MQSeries Commands (MQSC). Your queue manager can be monitored locally or remotely using Instrumentation Events.

Because MQSeries for VSE runs as a CICS subsystem, using VSAM KSDS files to store message data, system administration also involves maintenance of your VSAM files. You can maintain your VSAM files using VSAM utilities, or you can use MQSeries' automatic or batch reorganization features.

5.1 Programmable Command Formats (PCF)

Programmable Command Formats (PCFs) provide a means for administrative applications to manipulate the queue manager and its objects. You can use PCFs to create, delete, or dynamically change queue manager, queue, and channel attributes. You can also use PCFs to inquire on these objects. You cannot, however, modify a queue that is currently open, or a channel that is active.

A PCF is a special type of message written to the system command queue. The structure of PCF messages is documented in the *MQSeries for VSE System Management Guide*, GC34-5364. The system command queue is a system queue designated to receive PCF messages. The name of the system command queue is an attribute of the queue manager's communication settings, and can be configured through MQMT option 1.1 or **PF9**.

The system command queue is monitored by the MQSeries for VSE command server transaction (MQCS). The command server waits for PCF messages to arrive on the system command queue, validates them and then passes them on to the command processor transaction (MQCP) for processing.

If your system command queue is not defined, or if the command server is not running, PCF messages cannot be processed. You can automatically start the command server when MQSeries is started by setting the command server auto-start parameter to `Y` in the queue manager's communication settings, or by running transaction MQCS. The command server is stopped automatically when MQSeries is shut down if the command server auto-start parameter is set to `Y`, or by running transaction MQCS with the `X` option (that is, `MQCS X`).

Because PCF messages are processed from a queue, you can write administrative applications that run either locally or remotely. If your application runs locally, it can simply put messages directly to the system command queue for processing. If your application runs remotely, your application can put messages to a remote queue that identifies the system command queue as its target. The result of a PCF message, whether success, failure or requested information, is sent by the command processor transaction to the ReplyTo queue specified in the PCF message's MQMD. So if you plan to write a remote administrative application, you can use remote queues for replies, and send responses back to the queue manager where your application is running.

Regardless of whether your administrative program runs locally or remotely, the programming is the same. That is, the structure of PCF messages and what they do is the same.

5.1.1 PCF data structures

As already mentioned, PCF data structures are described in the *MQSeries for VSE System Management Guide*, GC34-5364. All PCF messages and replies begin with 36-byte PCF header structure, MQCFH. The PCF header has the following definition:

```
typedef struct tagMQCFH {
  MQLONG  Type;           /* Structure type */
  MQLONG  StrucLength;    /* Structure length */
  MQLONG  Version;        /* Structure version number */
  MQLONG  Command;        /* Command identifier */
  MQLONG  MsgSeqNumber;   /* Message sequence number */
  MQLONG  Control;        /* Control options */
  MQLONG  CompCode;       /* Completion code */
  MQLONG  Reason;         /* Reason code */
  MQLONG  ParameterCount; /* Count of parameters */
} MQCFH;
```

Note that the Command field identifies the command you want to execute. There are constants defined for each command, for example, MQCMD_CHANGE_Q. The constants are defined in language-specific header files and copybooks in the MQSeries installation sublibrary. See 8.1.1, “Copybooks and header files” on page 148 for more details.

Although these copybooks and header files contain constants for all PCF commands, not all are supported by MQSeries for VSE. They are included in case you plan to write an administrative application that administers non-VSE queue managers that do support the commands. The *MQSeries for VSE System Management Guide*, GC34-5364, has a definitive list of the PCF commands supported by MQSeries for VSE.

PCFs invariably involve additional parameters. For example, a parameter might be the name of the queue you are attempting to modify, or the new maximum message size you are trying to set for a channel. Such parameters are appended to the header and are usually integer or string parameters.

Integer parameters are appended as MQCFIN structures, one for each integer parameter. The MQCFIN structure has the following definition:

```
typedef struct tagMQCFIN {
  MQLONG  Type;         /* Structure type */
  MQLONG  StrucLength;  /* Structure length */
  MQLONG  Parameter;    /* Parameter identifier */
  MQLONG  Value;        /* Parameter value */
} MQCFIN;
```

Each MQSeries object attribute has an integer constant associated with it, for example MQIA_Q_TYPE. These are defined in the provided header files and copybooks.

String parameters are appended as MQCFST structures. Once again, you will append one MQCFST structure for each string parameter relevant to the command you are trying to execute. The MQCFST structure has the following definition:

```
typedef struct tagMQCFST {
  MQLONG  Type;            /* Structure type */
  MQLONG  StrucLength;     /* Structure length */
  MQLONG  Parameter;       /* Parameter identifier */
  MQLONG  CodedCharSetId;  /* Coded charset identifier */
  MQLONG  StringLength;    /* Length of string */
  MQCHAR  String[1];       /* String value */
} MQCFST;
```

One thing to remember with string parameters is that their total length, StrucLength, must be evenly divisible by 4. In other words, if you have a 2-character string parameter, the StringLength can be 2, but the overall structure must be padded with nulls or blanks to make the overall length of the MQCFST structure evenly divisible by 4. This is an idiosyncrasy that, if missed, can put subsequent parameters out of alignment.

PCF commands that inquire on MQSeries objects (such as MQCMD_INQUIRE_Q) use integer list type parameters to specify a list of attributes. Such commands typically return values for the requested attributes as MQCFIN and MQCFST parameters. The MQCFIL structure is defined as follows:

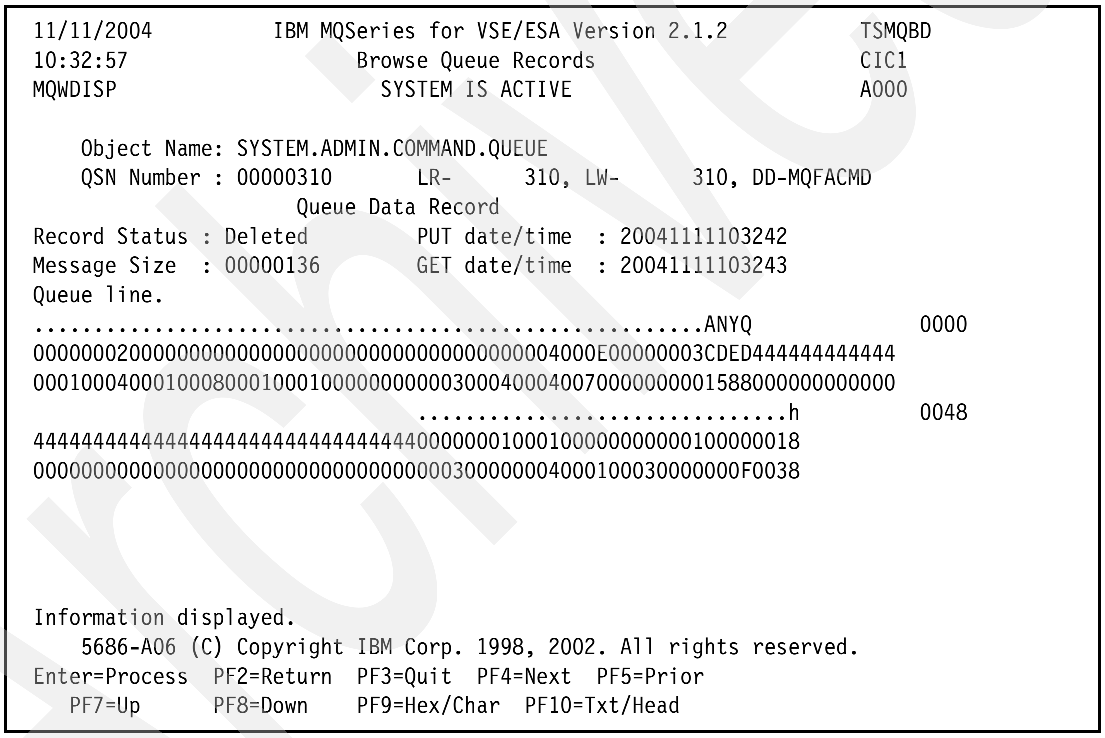
```
typedef struct tagMQCFIL {
  MQLONG  Type;         /* Structure type */
  MQLONG  StrucLength;  /* Structure length */
  MQLONG  Parameter;    /* Parameter identifier */
  MQLONG  Count;        /* Count of parameter values */
  MQLONG  Values[1];    /* Parameter values */
} MQCFIL;
```

Finally, in response PCF messages only, a PCF response can contain a list of strings in an MQCFSL structure. The MQCFSL structure has the following definition:

```
typedef struct tagMQCFSL {
  MQLONG  Type;            /* Structure type */
  MQLONG  StrucLength;     /* Structure length */
  MQLONG  Parameter;       /* Parameter identifier */
  MQLONG  CodedCharSetId;  /* Coded charset identifier */
  MQLONG  Count;           /* Count of parameter values */
  MQLONG  StringLength;    /* Length of one string */
  MQCHAR  Strings[1];      /* String values */
 } MQCFSL;
```

The MQCFSL structure is specifically used to return lists of queue or channel names from the **INQUIRE_Q_NAMES** and **INQUIRE_CHANNEL_NAMES** commands, respectively.

Figure 5-1 illustrates what a PCF command looks like in hexadecimal. This example is for a change queue command, **MQCMD_CHANGE_Q**, to change the maximum queue depth, MQIA_MAX_Q_DEPTH:



```
11/11/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
10:32:57                   Browse Queue Records                      CIC1
MQWDISP                      SYSTEM IS ACTIVE                        A000

    Object Name: SYSTEM.ADMIN.COMMAND.QUEUE
    QSN Number : 00000310       LR-      310, LW-      310, DD-MQFACMD
                      Queue Data Record
Record Status : Deleted         PUT date/time  : 20041111103242
Message Size  : 00000136        GET date/time  : 20041111103243
Queue line.
........................................................ANYQ              0000
000000020000000000000000000000000000000000000004000E00000003CDED444444444444
000100040001000800010001000000000003000400040070000000001588000000000000
                                ...............................h          0048
4444444444444444444444444444444400000001000100000000000100000018
0000000000000000000000000000000000030000000400010003000000000F0038




Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 5-1 Hexadecimal example of a PCF command

Figure 5-2 on page 85 illustrates what the PCF response to the change queue command might look like.

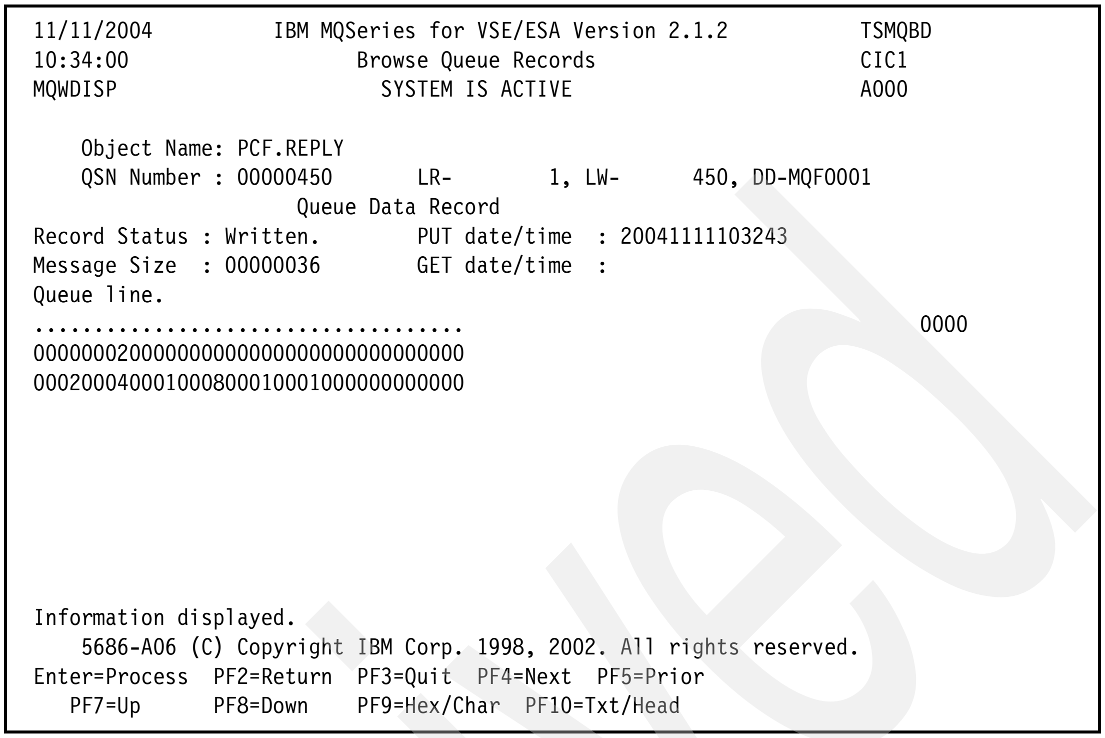

```
11/11/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:34:00                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: PCF.REPLY
    QSN Number : 00000450       LR-        1, LW-      450, DD-MQF0001
                      Queue Data Record
Record Status : Written.        PUT date/time  : 20041111103243
Message Size  : 00000036        GET date/time  :
Queue line.
....................................                                    0000
000000020000000000000000000000000000
000200040001000800010001000000000000









Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 5-2 Hexadecimal example of a PCF response

In the following section, we look at the programming behind generating PCF commands.

5.1.2 PCF programming

If you are writing your administrative application on VSE, you can use Language Environment C, COBOL, or PL/I. You can also use Assembler language, but the Assembler program must then be called by a Language Environment main program to ensure proper program initialization. This is needed for the MQI calls your program will issue. If you are writing your application on a non-VSE system, you can use the languages supported on that platform.

In this section, we provide an example of a VSE COBOL program that issues a change queue command to change the maximum queue depth of a target queue.

Whether your program is written in COBOL, C, or PL/I, you need to include the PCF-related copybooks. In a COBOL program you would do this in the program's Working-Storage Section, or the Linkage Section if you plan to dynamically create PCF messages.

Firstly, you need the copybooks that contain constants for your MQI calls, and for PCF messages in general, for example:

```
01 WS-CMQV.
   COPY CMQV.

01 WS-CMQCFV.
   COPY CMQCFV.
```

You also need to include the copybooks that contain PCF structure definitions. In this example, we use a fixed data structure for the change queue PCF message for simplicity. However, if your program will generate a range of PCF messages, you will probably want to dynamically build messages by allocating storage and using the program's Linkage Section to map diverse message structures.

The change queue command has two mandatory parameters: the queue name you want to change, and the queue type (for example, local). The command also needs parameters for those attributes you want to change. In our case, we only want to change the maximum queue depth, so we need three parameters altogether.

```
01 WS-PCF-MSG.
   05 WS-HEADER.
      COPY CMQCFHV.
   05 WS-QNAME-PARM.
      COPY CMQCFSTV.
      15 WS-TARGET-Q  PIC X(48).
   05 WS-QTYPE-PARM.
      COPY CMQCFINV.
   05 WS-QDEPTH-PARM.
      COPY CMQCFINV.
```

Notice that our PCF message contains a PCF header, followed by three parameters: a string parameter for the queue name, an integer parameter for the queue type, and an integer parameter for the maximum queue depth. The string parameter structure does not include a field for the string value itself, so we have inserted a 48-character field to contain the queue name.

Your program will need other variables, but for brevity, we have not included the entire Working-Storage Section here. Instead, this example is available in full in “PCF sample” on page 276.

In the Procedure Division, the first thing we need to do is connect to the queue manager. Our example will run as a CICS transaction, local to the queue manager. In this case, we can use a blank queue manager name, which defaults to the local queue manager.

```
SET WS-HCONN-VALUE TO NULL.
MOVE SPACES TO WS-QM-NAME.

CALL 'MQCONN' USING WS-QM-NAME
                    WS-HCONN
                    WS-CCODE
                    WS-RCODE.
```

When making MQI calls, your program should always check the completion code, and possibly also the reason code.

Next, we build the PCF message. As shown in the working storage definition, our PCF message is a header followed by three PCF parameters. The copybooks we have included in our PCF message declaration have default values for many of the fields, so we don't need to change these. You can look in the copybook file to see what fields have default values.

The constant for a change queue command is MQCMD-CHANGE-Q, and the header also needs to know how many parameters follow.

```
MOVE MQCMD-CHANGE-Q   TO MQCFH-COMMAND.
MOVE 3                TO MQCFH-PARAMETERCOUNT.
```

The first parameter, the queue name, is a string parameter. We need to set the overall length of the string parameter, and identify which parameter we are supplying, in this case MQCA-Q-NAME. We also need to provide the string length and its value. In this example we will change the maximum queue depth of the queue called ANYQ.

```
ADD  MQ-Q-NAME-LENGTH TO MQCFST-STRUCLENGTH.
MOVE MQ-Q-NAME-LENGTH TO MQCFST-STRINGLENGTH.
MOVE MQCA-Q-NAME      TO MQCFST-PARAMETER.
MOVE 'ANYQ'           TO WS-TARGET-Q.
```

The second parameter, the queue type, is an integer parameter. The queue type is identified by the constant MQIA-Q-TYPE. Since our target queue, ANYQ, is a local queue, the value for the queue type parameter is MQQT-LOCAL.

```
MOVE MQIA-Q-TYPE      TO MQCFIN-PARAMETER
                      OF WS-QTYPE-PARM.
MOVE MQQT-LOCAL       TO MQCFIN-VALUE
                      OF WS-QTYPE-PARM.
```

The last parameter, the maximum queue depth we want to set, is an integer parameter. Maximum queue depth is identified by the constant MQIA_MAX_Q_DEPTH, and we use a value of 5000.

```
MOVE MQIA-MAX-Q-DEPTH TO MQCFIN-PARAMETER
                      OF WS-QDEPTH-PARM.
MOVE 5000             TO MQCFIN-VALUE
                      OF WS-QDEPTH-PARM.
```

Now that the message is ready, we can put it to the system command queue using a combination of MQOPEN, MQPUT, and MQCLOSE, or more simply by using the MQPUT1 call. But first, we have to set mandatory values in the message's MQMD. Among these, we need to identify the target queue for our PCF message, SYSTEM.ADMIN.COMMAND.QUEUE, and a ReplyTo queue for a response to our PCF message, PCF.REPLY.

```
MOVE 'SYSTEM.ADMIN.COMMAND.QUEUE' TO MQOD-OBJECTNAME.
MOVE SPACES                       TO MQOD-OBJECTQMGRNAME.
MOVE MQMT-REQUEST                 TO MQMD-MSGTYPE.
MOVE MQFMT-ADMIN                  TO MQMD-FORMAT.
MOVE 'PCF.REPLY'                  TO MQMD-REPLYTOQ.
MOVE MQOO-OUTPUT                  TO MQPMO-OPTIONS.

CALL 'MQPUT1' USING WS-HCONN
                    WS-MQOD
                    WS-MQMD
                    WS-MQPMO
                    WS-PCF-LEN
                    WS-PCF-MSG
                    WS-CCODE
                    WS-RCODE.
```

As with all MQI programs, messages are not available to other programs until they are syncpointed. We need to syncpoint (or rollback) explicitly, and then use the MQCMIT (or MQBACK) MQI call to let MQSeries know that the message is available.

```
IF WS-CCODE-VALUE = MQCC-OK
    EXEC CICS SYNCPOINT END-EXEC
    CALL 'MQCMIT' USING WS-HCONN
                        WS-CCODE
                        WS-RCODE
    END-CALL
ELSE
    EXEC CICS SYNCPOINT ROLLBACK END-EXEC
    CALL 'MQBACK' USING WS-HCONN
                        WS-CCODE
                        WS-RCODE
    END-CALL
END-IF.
```

After the message is syncpointed, the MQSeries Command Server will retrieve, validate, and pass it on to the MQSeries command processor transaction for processing.

Normally, an administrative program would wait for the PCF response message, and process it. For simplicity, at this point we have created and sent a PCF message, which will generate a response to our ReplyTo queue, PCF.REPLY. If successful, ANYQ will have a maximum queue depth of 5000.

Finally, it is necessary to disconnect from the queue manager. If an MQI application fails to disconnect before ending, the MQSeries system monitor transaction (MQSM) will detect the abandoned connection and free the associated resources. This causes warning messages to be written to the system log, so it is better for the program to explicitly disconnect from the queue manager.

```
IF WS-HCONN-VALUE NOT = NULL
    CALL 'MQDISC' USING WS-HCONN
                        WS-CCODE
                        WS-RCODE
    END-CALL
END-IF.
```

This completes our program example. The same principles apply to the generation of all PCF messages. Once again, a full listing of this example is available in “PCF sample” on page 276.

5.2 MQSeries Commands (MQSC)

MQSeries Commands are verb-based requests that allow you to create, delete, and modify MQSeries objects. For example,

```
ALTER QLOCAL(ANYQ) MAXDEPTH(5000)
```

MQSeries Commands are processed by a batch utility program called MQPMQSC, which is provided in the MQSeries installation sublibrary. Commands are read as SYSIPT data by the MQPMQSC program.

The MQPMQSC utility uses the MQSeries batch interface to put messages to system command queue, and expects responses on the system command reply queue. The names of these two queues are configurable as part of the queue manager's communication settings. Once the MQPMQSC program connects to the VSE queue manager via the batch interface, it uses the MQINQ MQI call to look up the names of these queues.

Because the MQPMQSC program uses the batch interface, your JCL when running the utility must identify the queue manager's batch identifier using a // SETPARM card. Your JCL must also specify your Language Environment installation sublibrary in a LIBDEF SEARCH card. For example:

```
// JOB MQSCRUN
// SETPARM MQBISRV='MQBISRV1'
// LIBDEF *,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPMQSC,SIZE=MQPMQSC
ALTER QLOCAL(ANYQ) MAXDEPTH(5000)
/*
/&
```

Note that MQSeries Commands are verb-based requests, but the system command queue expects PCF messages. The MQPMQSC utility converts the verb-based requests to a special PCF message called a PCF Escape message.

A PCF Escape message has a PCF header followed by an integer EscapeType parameter and a string EscapeText parameter. The EscapeType parameter always identifies the PCF Escape message as an MQSeries Command, and the EscapeText contains the verb-based request. PCF Escape messages are described in the *MQSeries for VSE System Management Guide*, GC34-5364.

If you want to administer MQSeries for VSE remotely, but you want to use MQSeries Commands rather than general PCF messages, you can use PCF Escape messages. Although these are PCF messages, they have a simple header and parameters. The main part of a PCF Escape is the verb-based request.

As an example, we can change the COBOL program described in section 5.1.2, “PCF programming” on page 85 to generate a PCF Escape message.

Firstly, the declaration of our PCF message in working storage needs to change to map to the PCF Escape data structure as follows:

```
01 WS-PCF-MSG.
   05 WS-HEADER.
      COPY CMQCFHV.
   05 WS-ESC-TYPE.
      COPY CMQCFINV.
   05 WS-ESC-TEXT.
      COPY CMQCFSTV.
      15 WS-MQ-CMD  PIC X(100).
```

In this case, we have reserved 100 bytes for the verb-based request.

As in the original example, we still need to connect to the queue manager, and then build the PCF message. This time, the command is the MQCMD-ESCAPE, and there are two parameters.

```
MOVE MQCMD-ESCAPE        TO MQCFH-COMMAND.
MOVE 2                   TO MQCFH-PARAMETERCOUNT.
```

The first parameter is the *EscapeType*, which is an integer parameter with a parameter constant of MQIACF-ESCAPE-TYPE, and a value of MQET-MQSC to indicate that the PCF Escape is for an MQSeries Command.

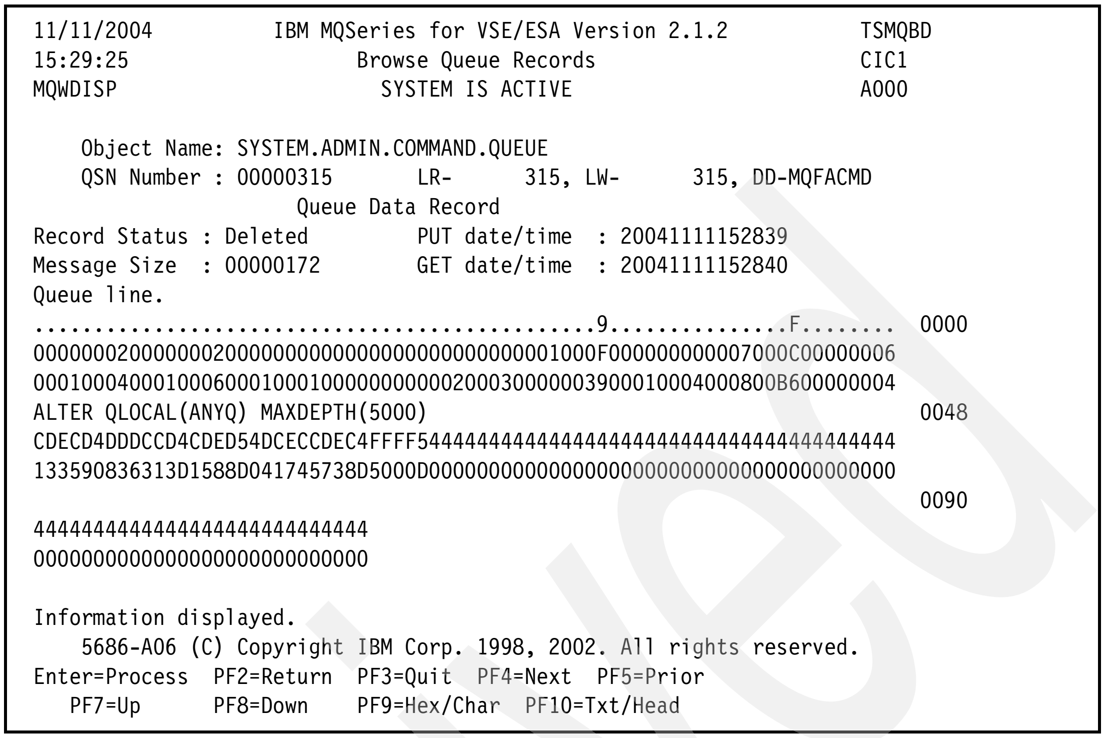
```
MOVE MQIACF-ESCAPE-TYPE  TO MQCFIN-PARAMETER.
MOVE MQET-MQSC           TO MQCFIN-VALUE.
```

The second parameter is the *EscapeText*, which is a string parameter with a parameter constant of MQCACF-ESCAPE-TEXT. The string is the MQSeries Command.

```
MOVE LENGTH OF WS-MQ-CMD TO MQCFST-STRINGLENGTH.
ADD  MQCFST-STRINGLENGTH TO MQCFST-STRUCLENGTH.
MOVE MQCACF-ESCAPE-TEXT  TO MQCFST-PARAMETER.
MOVE 'ALTER QLOCAL(ANYQ) MAXDEPTH(5000)'
                         TO WS-MQ-CMD.
```

The PCF Escape message is now built. We can now put the message to the system command queue using MQPUT1, syncpoint, and disconnect as we did in the original example.

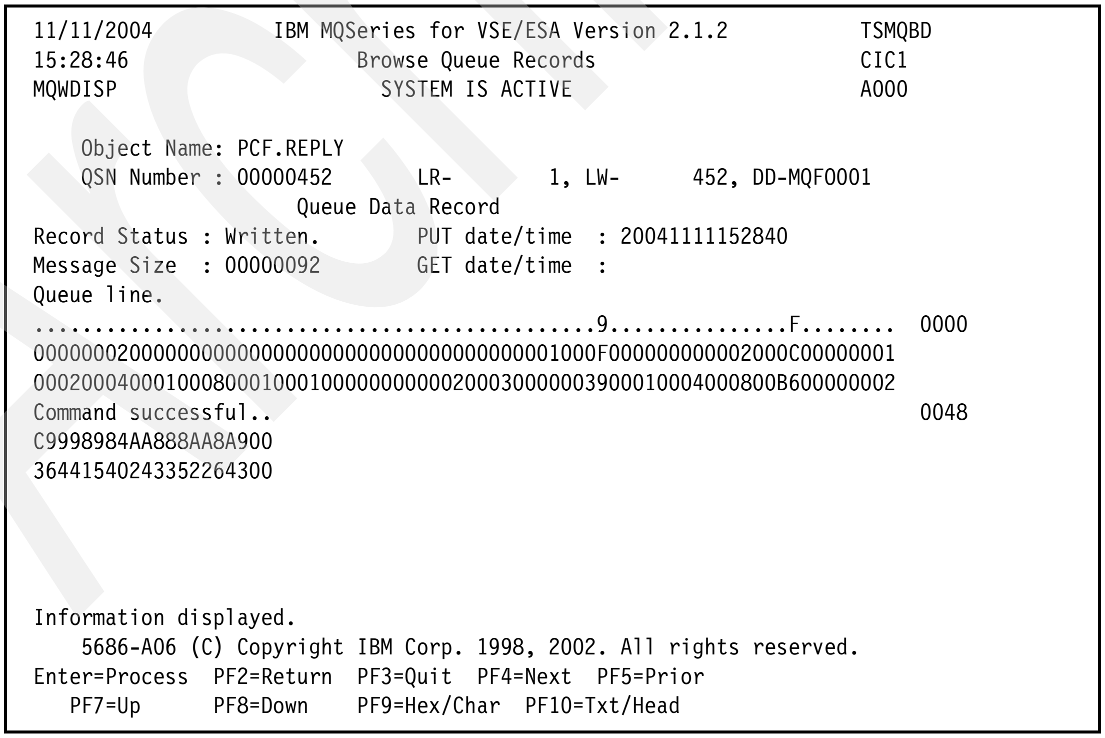
Figure 5-3 on page 90 shows the PCF Escape message we generated:



```
11/11/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:29:25                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: SYSTEM.ADMIN.COMMAND.QUEUE
    QSN Number : 00000315       LR-      315, LW-      315, DD-MQFACMD
                      Queue Data Record
Record Status : Deleted        PUT date/time  : 20041111152839
Message Size  : 00000172       GET date/time  : 20041111152840
Queue line.
..............................................9..............F........  0000
000000020000000200000000000000000000000000001000F000000000007000C00000006
000100040001000600010001000000000002000300000039000100040008 00B600000004
ALTER QLOCAL(ANYQ) MAXDEPTH(5000)                                        0048
CDECD4DDDCCD4CDED54DCECCDEC4FFFF5444444444444444444444444444444444444444
133590836313D1588D041745738D5000D000000000000000000000000000000000000000
                                                                         0090
4444444444444444444444444444
0000000000000000000000000000

Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 5-3 Hexadecimal example of a PCF Escape message

Figure 5-4 shows the PCF Escape response generated by the MQSeries command processor:



```
11/11/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:28:46                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: PCF.REPLY
    QSN Number : 00000452       LR-        1, LW-      452, DD-MQF0001
                      Queue Data Record
Record Status : Written.       PUT date/time  : 20041111152840
Message Size  : 00000092       GET date/time  :
Queue line.
..............................................9..............F........  0000
000000020000000000000000000000000000000000001000F000000000002000C00000001
000200040001000800010001000000000002000300000039000100040008 00B600000002
Command successful..                                                     0048
C9998984AA888AA8A900
36441540243352264300




Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 5-4 Hexadecimal example of a PCF Escape response

A full listing of this example is available in “PCF sample” on page 276.

5.3 Instrumentation events

Instrumentation events can be used to monitor your MQSeries queue manager with a local or remote administrative application. Instrumentation events are operational messages generated by the queue manager as predefined events occur, for example when the queue manager starts or stops, when a channel starts or stops, or when a queue reaches a certain depth.

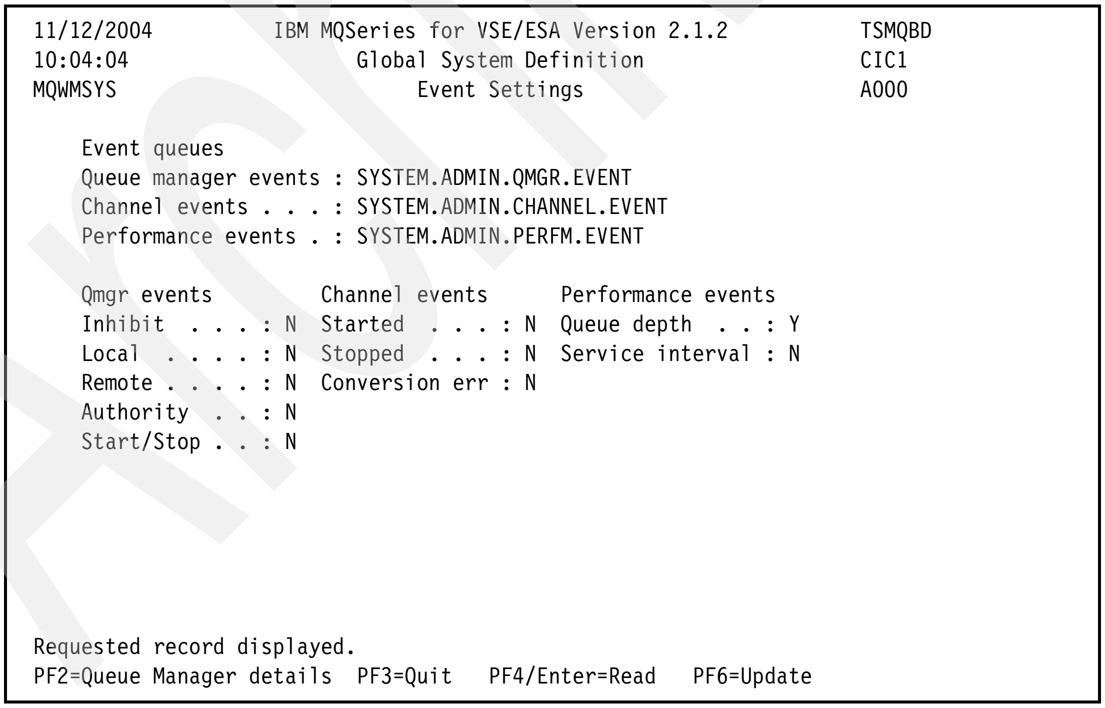
The generation of instrumentation events is configurable as part of the queue manager's event settings, and for events relevant to specific queues, as part of the queue's definition.

The range of Instrumentation Event messages that can be generated are documented in the *MQSeries for VSE System Management Guide*, GC34-5364. There is also a brief overview in section 2.1.3, “Event settings” on page 38.

In this chapter, we are interested in MQSeries system administration. Because monitoring your queue manager's operation is a system administration task, in the following sections, we provide an example of how you might write an administrative application that monitors your queue manager.

5.3.1 Activating event messages

Before event messages can be processed by an application, they must be activated. If they are not activated, they will not be generated, and your application will have nothing to process. As already mentioned, instrumentation events are a configurable part of the queue manager definition. You can access event configuration through MQMT option 1.1 or **PF11**.

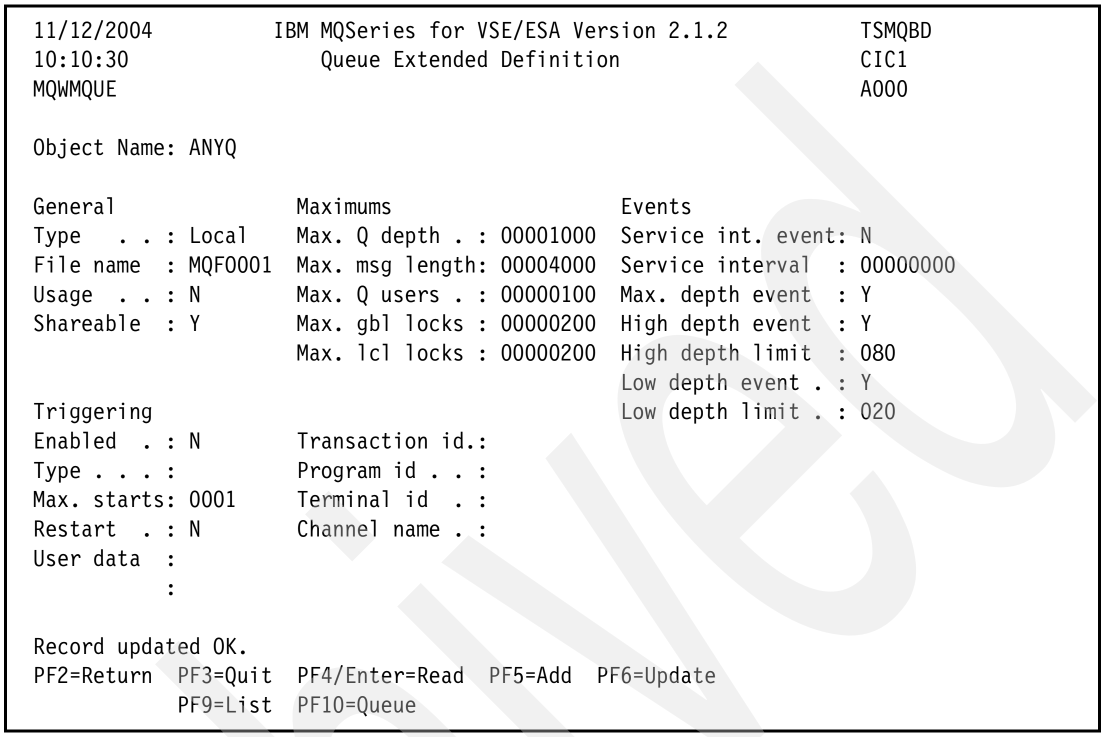


```
11/12/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:04:04                  Global System Definition                 CIC1
MQWMSYS                        Event Settings                      A000

    Event queues
    Queue manager events : SYSTEM.ADMIN.QMGR.EVENT
    Channel events . . . : SYSTEM.ADMIN.CHANNEL.EVENT
    Performance events . : SYSTEM.ADMIN.PERFM.EVENT

    Qmgr events         Channel events      Performance events
    Inhibit  . . . : N  Started  . . . : N  Queue depth  . . : Y
    Local  . . . . : N  Stopped  . . . : N  Service interval : N
    Remote . . . . : N  Conversion err : N
    Authority  . . : N
    Start/Stop . . : N






Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 5-5 Queue manager event settings configuration

For our example, we will create an application that monitors events related to queue depth. These are specifically performance events. So, to activate queue depth events, we must set the Queue depth parameter to `Y` in the queue manager's event settings.

Because queue depth events are relative to a specific queue, we also need to activate queue depth events in the queue definition. For our example, we will write an application that monitors the queue depth for a queue called ANYQ.



```
11/12/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:10:30               Queue Extended Definition                   CIC1
MQWMQUE                                                            A000

Object Name: ANYQ

General              Maximums                   Events
Type   . . : Local    Max. Q depth . : 00001000  Service int. event: N
File name  : MQF0001  Max. msg length: 00004000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : Y
Shareable  : Y        Max. gbl locks : 00000200  High depth event  : Y
                      Max. lcl locks : 00000200  High depth limit  : 080
                                                 Low depth event . : Y
Triggering                                       Low depth limit . : 020
Enabled  . : N        Transaction id.:
Type . . . :          Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Record updated OK.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure 5-6 Queue event configuration

Notice the section of the extended queue definition that pertains to events. For our example, we will configure MQSeries to generate event messages when the queue is full, when it reaches a certain depth, and when it drops to a certain depth.

To do this, we need to set the *Max. depth event* field to `Y`. This will cause an event message to be generated when the queue is full. We set *High depth event* to `Y`, and set the *High depth limit* to `80`. This means an event message will be generated when the queue is 80% full. Finally, we set the *Low depth event* to `Y`, and the *Low depth limit* to `20`, so that an event message is generated when the queue depth drops to 20% of its maximum.

Instrumentation events for queue depth are now activated for the queue ANYQ. Note that when an event message is generated by the queue manager, it is written to the event queue that corresponds to the event type. In our case, we are generating performance events, so the event messages will be written to the queue manager's performance event queue. The name of this queue is specified in the queue manager's event settings.

5.3.2 Processing event messages

Once again, if you are writing your administrative application on VSE, you can use Language Environment C, COBOL, or PL/I. You can also use Assembler language, but the Assembler program must then be called by a Language Environment main program to ensure proper program initialization. This is needed for the MQI calls your program will issue. If you are writing your application on a non-VSE system, you can use the languages supported on that platform.

For our example, we will write a VSE C program that monitors the performance event queue and writes messages to the console as events occur.

Event message data structures

Conveniently, instrumentation event messages use the same data structures as PCF messages. That is, they use the MQCFH header, followed by MQCFIN integer parameters and MQCFST string parameters.

For all performance events, the event data, following the header, contains the names of the queue manager and the queue associated with the event. Also, the event data contains statistics related to the event. You can use these statistics to analyze the behavior of a specified queue. The following table summarizes the event statistics.

Table 5-1 Performance event message statistical data

Parameter	Description
TimeSinceReset	The elapsed time since last reset
HighQDepth	Maximum queue depth since last reset
MsgEnqCount	Messages put since last reset
MsgDeqCount	Messages got since last reset

All the statistics refer to what has happened since the last time the statistics were reset. Performance event statistics are reset when a performance event occurs or the queue manager stops and restarts.

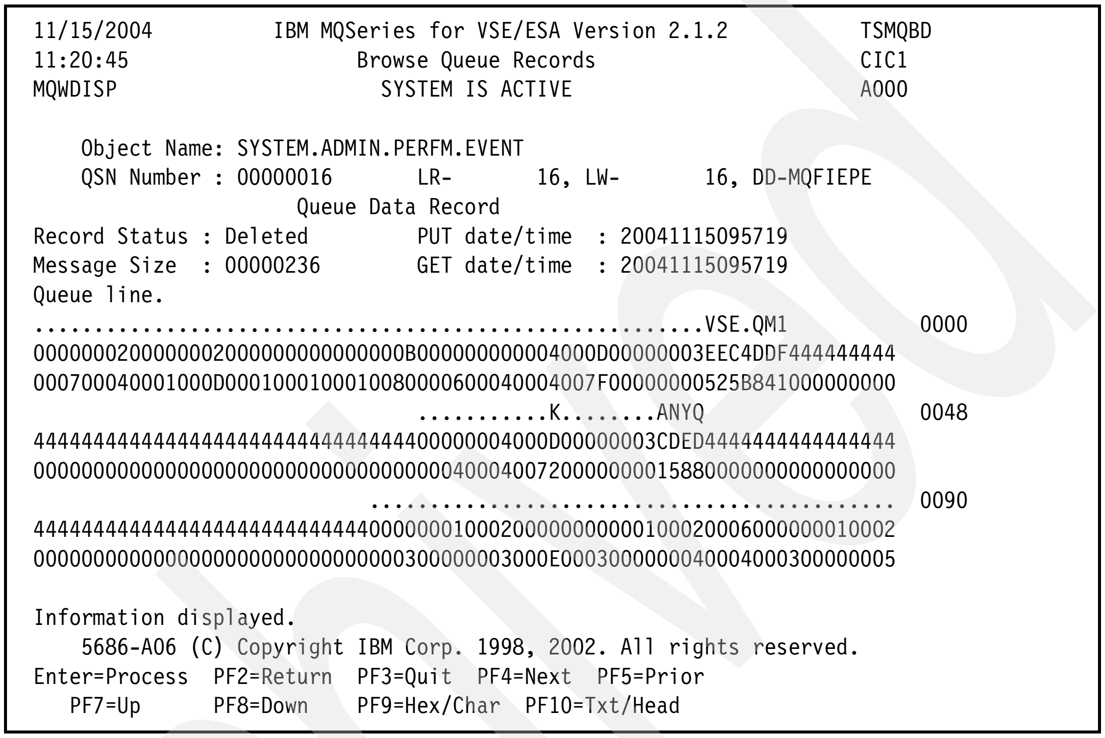
So for the performance events we plan to process, the data structure of each message is a header (MQCFH) followed by two string parameters (MQCFST) and four integer parameters (MQCFIN). The first string parameter is the queue manager name, and the second is the base queue name of the queue that the event relates to. The 4 integer parameters are the queue statistics. The queue manager name and the base queue name are always 48 characters long, so the overall structure and length of a performance event message is constant.

```
struct tagPerfEvent
{
  MQCFH    Header;
  struct
  {
    _Packed MQCFST Qmgr;
    MQCHAR QMgrName[MQ_Q_MGR_NAME_LENGTH-1];
  } QmgrParm;
  struct
  {
    _Packed MQCFST Q;
    MQCHAR QName[MQ_Q_NAME_LENGTH-1];
  } QParm;
  MQCFIN   Reset;
  MQCFIN   Depth;
  MQCFIN   Puts;
  MQCFIN   Gets;
} Evt;
```

Note that the MQCFST structures are packed (that is, _Packed). This is because the MQCFST structure does not end on a fullword boundary and is consequently padded. Since we want the QMgrName and the QName fields to be immediately contiguous with the MQCFST structures, we must pack them to avoid the padding.

Also note that the lengths of the queue manager name and the queue name string use a length constant less 1. This is because the MQCFST structure definition in C has a default string length of 1 already.

Figure 5-7 shows an example of a performance event message, in this case, a queue depth high event. The figure shows the first 216 bytes of the message.



```
11/15/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSMQBD
11:20:45                   Browse Queue Records                      CIC1
MQWDISP                      SYSTEM IS ACTIVE                        A000

    Object Name: SYSTEM.ADMIN.PERFM.EVENT
    QSN Number : 00000016       LR-       16, LW-       16, DD-MQFIEPE
                     Queue Data Record
Record Status : Deleted         PUT date/time  : 20041115095719
Message Size  : 00000236        GET date/time  : 20041115095719
Queue line.
.......................................................VSE.QM1          0000
000000020000000200000000000000000B00000000004000D00000003EEC4DDF444444444
000700040001000D000100010001008000060004000400 7F00000000525B841000000000
                                ...........K........ANYQ               0048
444444444444444444444444444444440000000004000D00000003CDED44444444444444
000000000000000000000000000000000040004007200000000158800000000000000000
                            ........................................  0090
444444444444444444444444444400000001000200000000000100020006000000010002
000000000000000000000000000000030000003000E000300000000400040003000000005

Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 5-7 Hexadecimal example of a performance event message

Processing logic for event messages

Whether you write your administrative programs in COBOL, C, or PL/I, you need to include the necessary header files or copybooks. For C programs, and for our example, we need to include the general MQI header file CMQC.H, and the CMQCFC.H header file which contains the PCF data structures. Remember that Instrumentation Event messages use the PCF data structures.

```
#include <cmqc.h>
#include <cmqcfc.h>
```

In the main function of the program, we first need to connect to the queue manager. Because our example will run as a CICS transaction local to the VSE queue manager, we can use a blank queue manager name which defaults to the local queue manager.

```
memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
MQCONN(QMName,
       &Hcon,
       &CompCode,
       &Reason);
```

Once connected, we can open the performance event queue. The default name for the performance event queue is SYSTEM.ADMIN.PERFM.EVENT, however on MQSeries for VSE, this name is configurable as part of the queue manager's event settings. We need to open the queue for input as we intend to get messages from the queue.

```
memset(od.ObjectQMgrName, ' ', MQ_Q_MGR_NAME_LENGTH);
strncpy(od.ObjectName, SYS_PERF_Q, MQ_Q_NAME_LENGTH);
O_options = MQOO_INPUT_SHARED;
MQOPEN(Hcon,
       &od,
       O_options,
       &Hobj,
       &CompCode,
       &Reason);
```

Performance event messages are written to the event queue as the events occur. Therefore, we need to set up a loop that issues the MQGET call with the MQGMO_WAIT option. This way, our program will wait until an event message is available. We can then process it and wait for the next event message.

```
processing = TRUE;
while ( processing )
{
    gmo.WaitInterval = MQWI_UNLIMITED;
    gmo.Options = MQGMO_WAIT;
    memcpy(gmo.StrucId, "GMO ", 4);
    memset(md.MsgId, 0, MQ_MSG_ID_LENGTH);
    memset(md.CorrelId, 0, MQ_CORREL_ID_LENGTH);

    MQGET(hcon, hobj, &md, &gmo, mlen,
          emsg, &dlen, &ccode, &rcode);
```

Note that we need to clear the MQMD's MsgId and CorrelId. This is because the MQGET call populates these fields when it successfully returns a message. If we leave these values in the MQMD, the next time we issue the MQGET, it will try to retrieve a message with a MsgId and CorrelId that matches the MQMD values. This will never happen, because that message has already been retrieved. Consequently, we need to clear these fields before each MQGET call. Null values in the MsgId and CorrelId fields tells the MQGET call that it can return the next available message rather than a message with a specific MsgId/CorrelId combination.

Our example program writes a message to the VSE console as queue depth events occur. So once we have successfully retrieved an event message, we need to examine it to see what type of event was generated. We can then build an appropriate console message.

```
switch ( Evt.Header.Reason )
{
    case MQRC_Q_FULL:
        strcpy(alert, "Queue full");
        break;
    case MQRC_Q_DEPTH_HIGH:
        strcpy(alert, "Queue depth high");
        break;
    case MQRC_Q_DEPTH_LOW:
        strcpy(alert, "Queue depth low");
        break;
    default:
        strcpy(alert, "Unexpected event");
        break;
}
```

The Reason field in the MQCFH header describes the event type. For our example, we have only activated queue depth events, so we only need to process these types. The message we send to the console is arbitrary, so for our purposes we will produce a message with the following format:

```
MQAlert - <alert> - for <queue name>
```

Where *<alert>* is replaced with text that describes the event type, and *<queue name>* is the name of the queue for which the event was generated. Once built, we can write the message to the console.

```
Evt.QParm.Q.String[Evt.QParm.Q.StringLength-1] = 0;
sprintf(conmsg,
        "MQAlert - %s - for %s",
        alert,
        Evt.QParm.Q.String);
conlen = strlen(conmsg);
EXEC CICS WRITE OPERATOR
          TEXT(conmsg)
          TEXTLENGTH(conlen);
```

Note that we have null terminated the queue name (that is, Q.String). This is because the queue name associated with the event is provided in an MQCFST data structure and may have a length of 48, which is the maximum length for a queue name. If the queue name has a length of 48, it will not be null terminated, so we zap the last character just in case.

The program will now continue to put messages to the console as queue depth performance events occur. We can terminate the loop in a number of ways, but for our example we will stop the loop when the MQGET call fails. It will fail when MQSeries is shut down.

```
processing = TRUE;
while ( processing )
{
    .
    .
    MQGET(...);

    if ( ccode == MQCC_OK )
    {
        .
        .
    }
    else
    {
        processing = FALSE;
    }
}
```

Having terminated the loop, we can close the queue and disconnect from the queue manager.

```
C_options = MQCO_NONE;
MQCLOSE(Hcon,
        &Hobj,
        C_options,
        &CompCode,
        &Reason);
MQDISC(&Hcon, &CompCode, &Reason);
```

A full listing of this program example is available in “Instrumentation event sample” on page 282.

5.4 VSAM reorganization

An administration task necessary with MQSeries for VSE is regular VSAM reorganization of the files that host your queues. This is particularly important for VSAM files that host busy queues, or queues that have a large volume of messages.

Because MQSeries for VSE uses a queue sequence number (QSN) as part of the key of queue messages, index space for the VSAM KSDS files is not reused. This means that VSAM KSDS indexes for files that host MQSeries queues will continue to grow unless you reorganize them periodically.

There are several ways you can reorganize your VSAM files. In this section, we discuss automatic reorganization and batch reorganization. However, you can perform your own reorganizations without using MQSeries services.

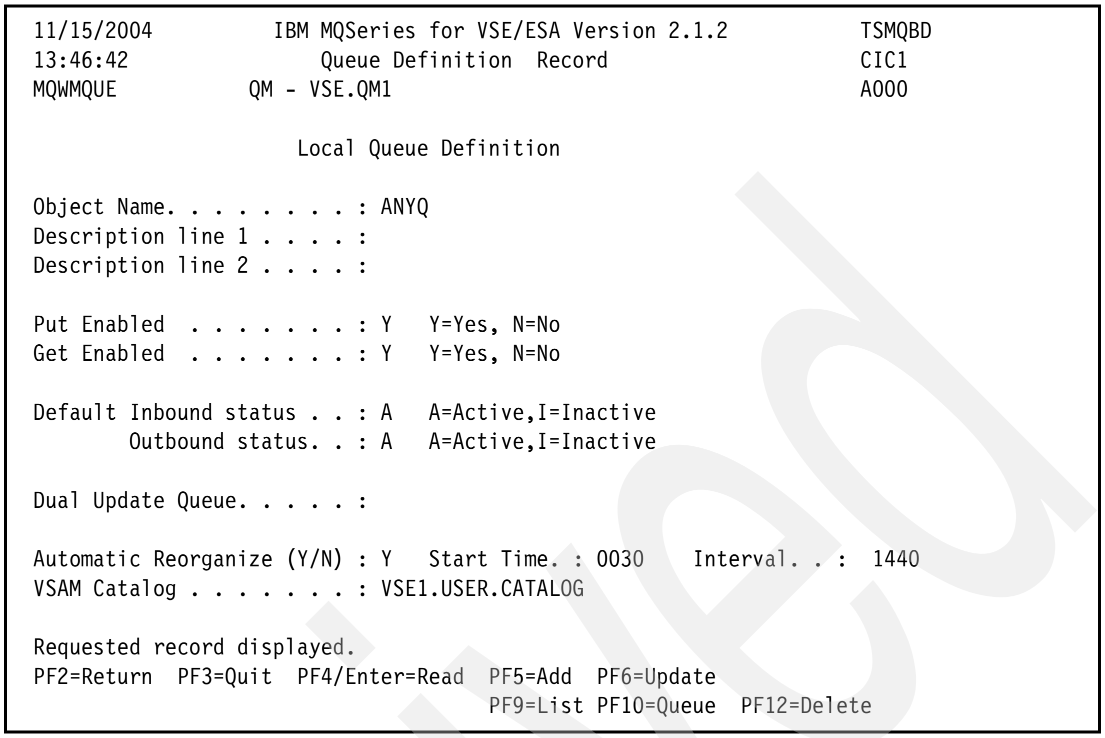
A reorganization is effectively a VSAM DELETE and DEFINE of your VSAM file. You can schedule these yourself, but you must keep some things in mind.

- If the VSAM file contains unread messages, these will be lost.
- Remember that some VSAM files may host multiple queues, although this is not recommended.
- All queues hosted by a VSAM file should be empty before reorganizing the file.
- You will not be able to DELETE and DEFINE a file while it is open in CICS.

5.4.1 Automatic reorganization

MQSeries for VSE V2.1.2 allows you to reorganize your VSAM files automatically, that is, MQSeries will perform the reorganization for you at a configurable time and frequency.

Automatic reorganization configuration forms part of the queue definition, and is only available for local queues (including transmission queues). Relevant configuration parameters are accessible using MQMT option 1.2.



```
11/15/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
13:46:42               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A000

                     Local Queue Definition

Object Name. . . . . . . . : ANYQ
Description line 1 . . . . :
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : Y   Start Time. : 0030    Interval. . :  1440
VSAM Catalog . . . . . . . : VSE1.USER.CATALOG

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure 5-8 Automatic reorganization configuration

Parameters associated with automatic reorganization are displayed toward the bottom of the screen, and include the following:

- Automatic Reorganization
- Start Time
- Interval
- VSAM Catalog

To activate automatic reorganization for a VSAM that hosts a queue, set Automatic Reorganize to `Y`. The reorganization will occur at the time specified by Start Time, and will be repeated after the number of minutes specified by the Interval parameter. The VSAM Catalog is needed because the reorganization process involves a VSAM DELETE and DEFINE, and the DEFINE step needs to know the VSAM catalog for the VSAM file.

In the example provided in Figure 5-8, we have scheduled an automatic reorganization for the queue named ANYQ. The reorganization is scheduled for 00:30, and is scheduled to run every 1440 minutes from that time (that is, daily). Automatic reorganizations will not take place if MQSeries is not active. Consequently, if you shut down MQSeries at, for example, 00:05, and then restart MQSeries at 00:35, the reorganization for ANYQ will not occur until 00:30 the next day.

The automatic reorganization process copies unread messages from the host VSAM file to an intermediate file, MQFREOR. Logically deleted messages, those that have already been processed, are discarded. There is only one MQFREOR file, so only one reorganization can take place at a time.

Therefore, you should not configure your automatic reorganizations to take place at the same time. Instead, you should schedule them to run minutes apart. The time it takes to reorganize a VSAM file depends on the number of read (logically deleted) and unread messages in the file. The reorganization process redefines the MQFREOR file to match the definition of the

VSAM file being reorganized. It then reads through the file looking for unread messages. These are copied to the MQFREOR file, each message with a new QSN starting from 1. The original file is then deleted and the MQFREOR file is renamed to the name of the original file. The MQFREOR file is then recreated for the next automatic reorganization. Knowing this process can help you estimate how long an automatic reorganization will take, and you can then schedule your reorganizations appropriately.

You must also remember that a reorganization cannot take place for a queue that is currently open. Therefore, you need to schedule your reorganizations to take place at a time when the queue is not in use.

In addition, you cannot reorganize a VSAM file that hosts multiple queues. If you configure an automatic reorganization for a queue hosted by a VSAM file that hosts other queues, MQSeries will cancel the reorganization as soon as it detects keyed records for a second queue.

You should take all of these things into consideration when you configure your queues for automatic reorganization.

5.4.2 Batch reorganization

The VSAM files that host your queues can also be reorganized from batch. MQSeries for VSE provides a batch utility program, MQPREORG, for this purpose. In addition, the MQSeries for VSE installation sublibrary includes a sample batch job, MQJREORG.Z.

The MQPREORG program copies unread messages from the VSAM file to a temporary KSDS. The VSAM file can then be deleted and redefined, and the contents of the temporary KSDS can be copied back using REPRO.

The sample batch job, MQJREORG.Z, includes a job step to define the temporary KSDS before MQPREORG is executed. After the execution of MQPREORG, the sample job includes a step to delete and redefine the original VSAM file and REPRO the temporary file back into the original file.

Consequently, you need to change the sample batch job so that the redefinition of your original VSAM file matches its original definition.

The MQPREORG program reads a single input card from SYSIPT. This card can either identify a particular queue to be reorganized, or you can specify `ALL` so that all queues in the VSAM file are reorganized. For example:

```
// LIBDEF PHASE,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPREORG,SIZE=AUTO
ALL
/*
```

Generally, you are not advised to have more than one queue associated with a single VSAM file. However, unlike the automatic reorganization process, if you do have multiple queues in a single VSAM file, the batch reorganization process will still work.

Finally, you cannot run the batch MQPREORG utility while the VSAM files are open in CICS. So to use the batch reorganization method, you need to shut down MQSeries and close the relevant files in CICS.

Distributed queuing

In this chapter, we look at distributed queuing. More specifically, we examine the objects involved in distributed queuing, how they interact, when they are used, and how they work. In Appendix A, "Distributed queuing examples" on page 207, we cover specific examples of distributed queuing across similar and dissimilar architectures.

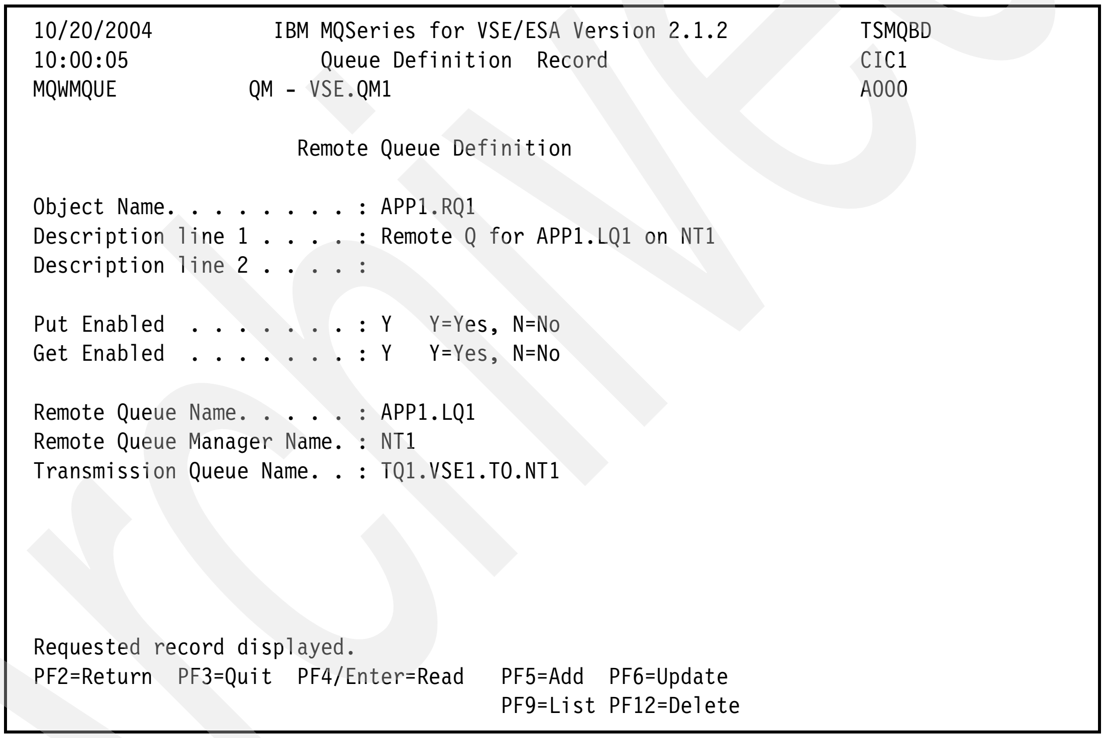
6.1 Remote queues

A remote queue can be viewed as a local definition of a local queue on a remote queue manager. A remote queue always mirrors a local definition on a remote queue manager. The remote queue manager can be on the same VSE system, a remote VSE system, or a dissimilar platform.

A remote queue definition is simple. Apart from the remote queue name, which must be unique for the local queue manager, the definition identifies:

- The local queue definition on the remote queue manager
- The remote queue manager name
- The transmission queue to use for temporarily storing messages prior to being sent to the remote queue manager

An example of a remote queue definition with key fields is shown in Figure 6-1.



```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:00:05               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE.QM1                                      A000

                      Remote Queue Definition

Object Name. . . . . . . . : APP1.RQ1
Description line 1 . . . . : Remote Q for APP1.LQ1 on NT1
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Remote Queue Name. . . . . : APP1.LQ1
Remote Queue Manager Name. : NT1
Transmission Queue Name. . : TQ1.VSE1.TO.NT1





Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read   PF5=Add  PF6=Update
                                       PF9=List PF12=Delete
```

Figure 6-1 Remote Queue Definition screen

An application program that wants to use a remote queue issues an MQOPEN or MQPUT1 call that identifies the remote queue name. This is exactly the same as using a local queue. In the case of putting messages to a remote queue, the application then issues an MQPUT or MQPUT1 call. In this instance, no messages are actually put on the remote queue. Instead, the messages are put on the transmission queue identified in the remote queue definition. The messages remain on the transmission queue until they are sent to the remote queue manager by the sender Message Channel Agent (MCA). This is usually instantaneous. For details about MCAs, refer to 6.4, “Message channel agents and communication programs” on page 115.

The sender MCA for MQSeries for VSE is the MQPSEND program. The MQPSEND program is invoked as a trigger program when a message is written to the transmission queue. For this reason, transmission queues should always identify the MQPSEND program as the trigger program. The transmission queue definition also identifies the MQ channel definition to use for establishing a connection to the remote host. The remote queue manager specified in the remote queue definition must reside on the host identified by the channel definition.

When the sender MCA establishes a connection with a remote host, it causes a partner transaction or program to execute on the remote host. This is the receiver MCA. For MQSeries for VSE, this is the MQPRECV program.

Having established a connection, the sender MCA begins to take messages from the transmission queue and send them to the receiver MCA. The receiver MCA then puts these messages on the local queue identified by the original remote queue definition. By local queue, we mean a local queue definition.

It should be noted that it is not possible to *get* messages from a remote queue. Applications can only *get* messages from local queues.

Figure 6-2 illustrates an application of distributed queuing.

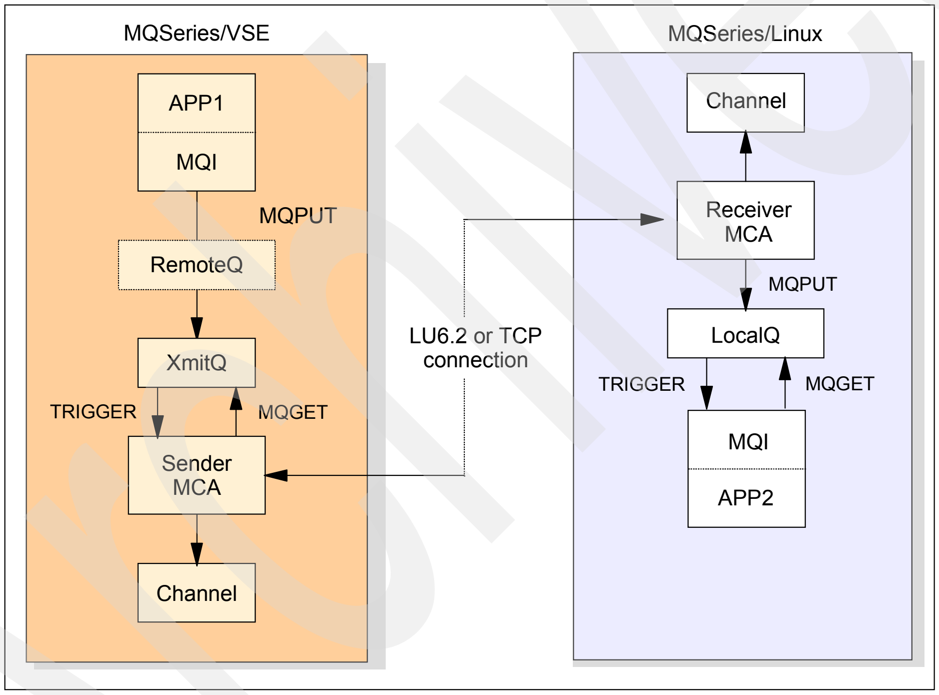


Figure 6-2 Diagram of distributed queuing

In this diagram, we have two application programs exploiting distributed queuing: APP1 and APP2. APP1 on the MQSeries/VSE system puts messages to a remote queue using MQPUT calls. The remote queue definition is used to identify a transmission queue as well as the target queue manager and target queue. This information is appended to messages written to the transmission queue. The transmission queue specifies a trigger to be fired when messages arrive. For distributed queuing, this is always the MQPSEND program, the sender MCA.

The sender MCA uses the channel definition identified in the transmission queue definitions and attempts to establish a connection with the remote host identified in the channel definitions. This connection may use LU 6.2 or TCP/IP. Establishing a connection causes the receiver MCA to be started on the remote host. The receiver MCA also uses a channel definition in establishing the connection. The channel name must be the same for both the sender and the receiver.

Once a connection is established, the sender MCA begins to read messages from the transmission queue and send them to the receiver MCA. This process continues until the sender MCA detects that no more messages are available to be sent. It then terminates.

The receiver MCA writes the received messages to the local queue definition identified in header information appended to the messages. For this example, the local queue definition is a local queue with a trigger to start application program APP2. The application program then begins to read messages from the local queue using MQGET calls. Refer to 8.3, “Trigger application programs” on page 158 for details on writing trigger programs.

Each of the MQ objects involved in the remote queuing process is described in further detail in the following sections.

6.2 Transmission queues

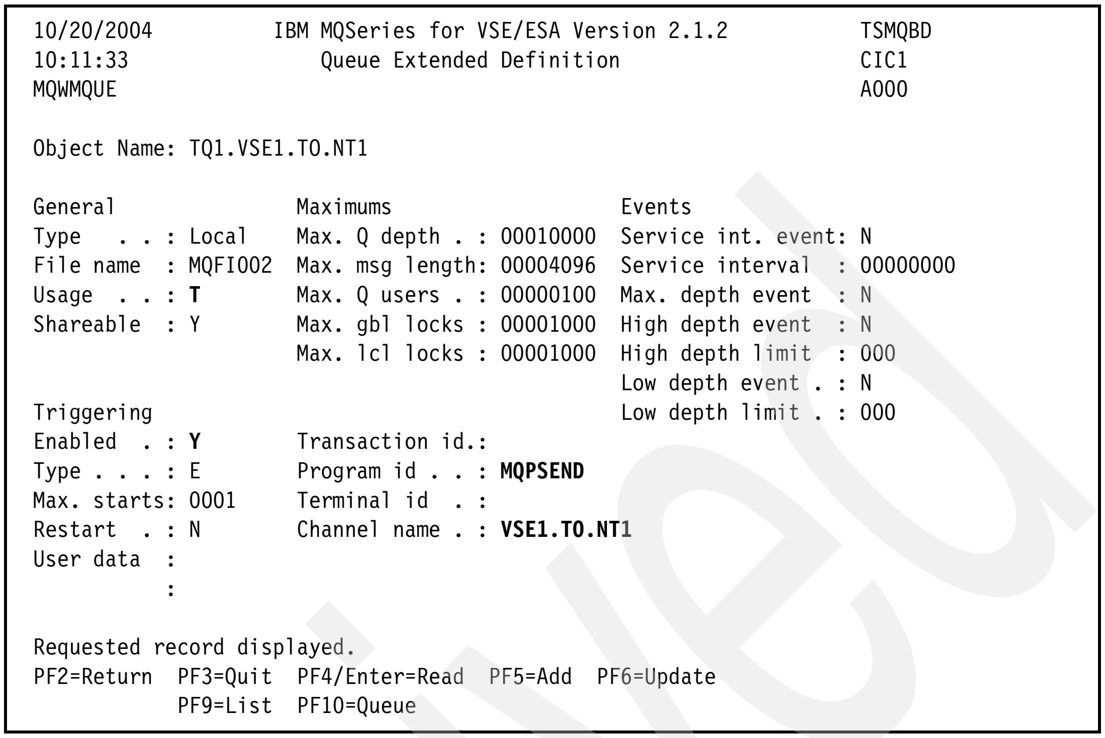
From the perspective of MQSeries for VSE, transmission queues are local queues. However, there are three significant differences between transmission queues and local queues:

- Transmission queues are defined with a usage mode of T (for transmission) in their extended queue definition.
- Transmission queues always have a trigger program enabled (MQPSEND).
- Transmission queues always identify a channel name.

It is important that transmission queues be defined with triggering enabled. Otherwise, messages written to remote queues (but actually to the transmission queue) will remain unprocessed indefinitely. It is also important that the trigger program specified in the extended queue definition is MQPSEND. MQPSEND is the sender MCA, and it alone knows the formats and protocols to use when talking to other MQ message channel agents. If messages are not intended for a remote queue, they should not be written to a transmission queue. Consequently, no trigger program other than MQPSEND is applicable when defining transmission queues.

The channel name must identify a valid and existing channel definition. This definition identifies a remote host and the communication protocol to use when talking to that host. In the case of transmission queues, the channel specified will always be a sender channel.

An example of a transmission queue extended definition with key fields in bold is shown in Figure 6-3 on page 105.



```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
10:11:33               Queue Extended Definition                   CIC1
MQWMQUE                                                            A000

Object Name: TQ1.VSE1.TO.NT1

General               Maximums                   Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI002  Max. msg length: 00004096  Service interval  : 00000000
Usage  . . : T        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00001000  High depth event  : N
                      Max. lcl locks : 00001000  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.:
Type . . . : E        Program id . . : MQPSEND
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . : VSE1.TO.NT1
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure 6-3 Transmission Queue Extended Definition example

Transmission queues are only used for outgoing messages. In other words, the receiver MCA does not write messages to a transmission queue; it writes messages directly to the local queue (or local remote queue) identified by the remote queue manager's remote queue definition.

In regard to triggering, the trigger type can be F (First) or E (Every); however, the maximum trigger starts should always be 1. If you specify trigger type `E` and maximum trigger starts `1`, the effect is the same as specifying trigger type `F`. One instance of the sender MCA is started to process messages arriving on the transmission queue.

Whether you use E or F may depend on the channel definition. If, in the channel definition, you specify `0` retries and `0` delay time for get retries, the sender MCA may terminate prematurely with a trigger type of F. This is because the sender MCA may detect that the transmission queue is empty *mid-batch* and terminate because there is no retry or delay time for checking for new messages. By mid-batch, we mean the occurrence of an application program writing a batch of messages to a queue during which time the sender MCA reads off messages faster than they are being written. In such a case, the sender MCA could terminate prematurely if get retries are defined as 0.

The recommended trigger configuration for a transmission queue is as follows:

Enabled	`Y`
Type	`E`
Program id	`MQPSEND`
Max starts	`0001`
Restart	`N`

… with an appropriate channel name. The channel should specify an appropriate get retry such as 3 and an appropriate get retry delay such as 10 seconds. This means the sender MCA will check three times, waiting 10 seconds between each check, for messages to arrive on the transmission queue before deeming the transmission complete and stopping the channel.

6.3 Channels

Channels are MQ communication definitions for connecting to remote hosts. For this reason, it is only meaningful to define channels to remote hosts running MQSeries. For MQSeries for VSE, there are three types of channels:

- Sender channels
- Receiver channels
- Client channels (SVRCONN)

Each of these is described in further detail in the following sections.

6.3.1 Sender channels

Sender channels define a communication path *to* a remote host. The significant values in the channel definition vary depending on the communication protocol selected for the communication path. For MQSeries for VSE, there are two communication protocols available:

- SNA LU 6.2
- TCP/IP

LU 6.2 sender channels

LU 6.2 sender channels require additional definitions in CICS other than the channel definition itself. These are connection and session definitions. There are also VTAM definitions that should be taken care of by your VTAM system programmer. For MQSeries, the relevant VTAM definitions include Logical Units (LU) for remote systems or CICS systems.

It is not necessary to define your CICS definitions first; however, the channel definition identifies a connection name, so it makes some sense to define your connections and sessions before defining your MQSeries LU 6.2 sender channels.

You can use the CEDA transaction to create connection definitions. For example:

```
CEDA DEF CON GR(group)
```

This opens a CEDA screen for defining a connection. In terms of communication with a remote host, the most significant values for connection definitions are:

- Connection
- Netname
- Accessmethod
- Protocol
- Autoconnect

The connection is a unique name for the CONNECTION definition. This name is used by the MQSeries LU 6.2 sender channel definition.

The *Netname* parameter must identify the LU for a remote MQSeries host or the APPLID of a remote CICS system running MQSeries. This name must be defined to VTAM.

The *Accessmethod* should be `Vtam`, and the *Protocol* should be `Appc`. *Autoconnect* should be set to `Yes`. Defaults for all other values should be acceptable. You should check with your CICS system programmer if you are unsure.

An example of a CICS CONNECTION definition for an LU 6.2 connection with key fields in bold is shown in Figure 6-4.

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0410
 CEDA  View Connection( VSE2 )
  Connection     : VSE2
  Group          : MQM
  DEscription    : CONNECTION TO CICS ON VSE2
 CONNECTION IDENTIFIERS
  Netname        : IYBPZS01
  INDsys         :
 REMOTE ATTRIBUTES
  REMOTESYSTem   :
  REMOTEName     :
  REMOTESYSNet   :
 CONNECTION PROPERTIES
  ACcessmethod   : Vtam              Vtam | IRc | INdirect
  PRotocol       : Appc              Appc | Lu61 | Exci
  Conntype       :                   Generic | Specific
  SInglesess     : No                No | Yes
  DAtastream     : User              User | 3270 | SCs | STrfield | Lms
  RECordformat   : U                 U | Vb
  Queuelimit     : No                No | 0-9999
  Maxqtime       : No                No | 0-9999
 OPERATIONAL PROPERTIES
  AUtoconnect    : Yes               No | Yes | All
  INService      : Yes               Yes | No
 SECURITY
  SEcurityname   :
  ATtachsec      : Local             Local | Identify | Verify | Persistent
                                     | Mixidpe
  BINDPassword   :                   PASSWORD NOT SPECIFIED
  BINDSecurity   : No                No | Yes
  Usedfltuser    : No                No | Yes
 RECOVERY
  PSrecovery     : Sysdefault        Sysdefault | None
```

Figure 6-4 Example of a CICS CONNECTION definition for an APPC channel

You can also use the CEDA transaction to create SESSION definitions. For example:

```
CEDA DEF SESS GR(group)
```

This opens a CEDA screen for defining a session. In terms of connections and sessions for MQSeries channels, the most significant values for SESSION definitions are:

- Connection
- Protocol
- Maximum
- Sendsize
- Receivesize
- Autoconnect

The Connection parameter must match the unique name of the CONNECTION definition. Protocol should be `Appc` and Autoconnect should be `Yes`.

The *Maximum* parameter represents two values. The first value represents the maximum number of sessions in the group. The second value represents the maximum number of sessions that are to be supported as contention winners. You should check with your CICS system programmer when choosing these values.

The Sendsize and Receivesize values are relevant to the VTAM buffer allocation. For APPC sessions, these values must be between 256 and 30720. Increasing either value causes more storage to be allocated by VTAM for the session but may decrease the number of physical messages sent. Once again, you should consult your CICS system programmer when choosing these values.

Figure 6-5 illustrates an example of a CICS SESSION definition for an LU 6.2 session with key fields in bold.

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0410
 CEDA  View Sessions( VSE1VSE2 )
  Sessions       : VSE1VSE2
  Group          : MQM
  DEscription    : VSE1 TO VSE2
 SESSION IDENTIFIERS
  Connection     : VSE2
  SESSName       :
  NETnameq       :
  MOdename       :
 SESSION PROPERTIES
  Protocol       : Appc              Appc | Lu61 | Exci
  MAximum        : 012 , 006         0-999
  RECEIVEPfx     :
  RECEIVECount   :                   1-999
  SENDPfx        :
  SENDCount      :                   1-999
  SENDSize       : 04096             1-30720
  RECEIVESize    : 04096             1-30720
  SESSPriority   : 000               0-255
  Transaction    :
 OPERATOR DEFAULTS
  OPERId         :
  OPERPriority   : 000               0-255
  OPERRsl        : 0                                            0-24,..
  OPERSecurity   : 1                                            1-64,..
 PRESET SECURITY
  USERId         :
 OPERATIONAL PROPERTIES
  Autoconnect    : Yes               No | Yes | All
  INservice      :                   No | Yes
  Buildchain     : Yes               Yes | No
  USERArealen    : 000               0-255
  NEPclass       : 000               0-255
 RECOVERY
  RECOvoption    : Sysdefault        Sysdefault | Clearconv | Releasesess
                                     | Uncondrel | None
```

Figure 6-5 Example of a CICS SESSION for an APPC connection

Once you have defined your connections and sessions, you can check if the connection is valid using CEMT. For example:

```
CEMT INQ CONN(conn)
```

This should display the connection as INSERVICE and ACQUIRED. For example:

```
INQ CONN(VSE2)
STATUS:  RESULTS - OVERTYPE TO MODIFY
Con(VSE2) Net(IYBPZS01)     Ins Acq     Xok Vta Appc
```

If you cannot acquire the connection, speak to your VTAM system programmer. You cannot use an MQSeries LU 6.2 channel until you can acquire the connection that it specifies.

Having created and verified your connection and session definitions in CICS, you can now define the MQSeries LU 6.2 sender channel.

In terms of communication with a remote host, the most significant values for LU 6.2 sender channels are:

- Protocol
- Type
- Connection
- TP Name
- Get retry number
- Get retry delay

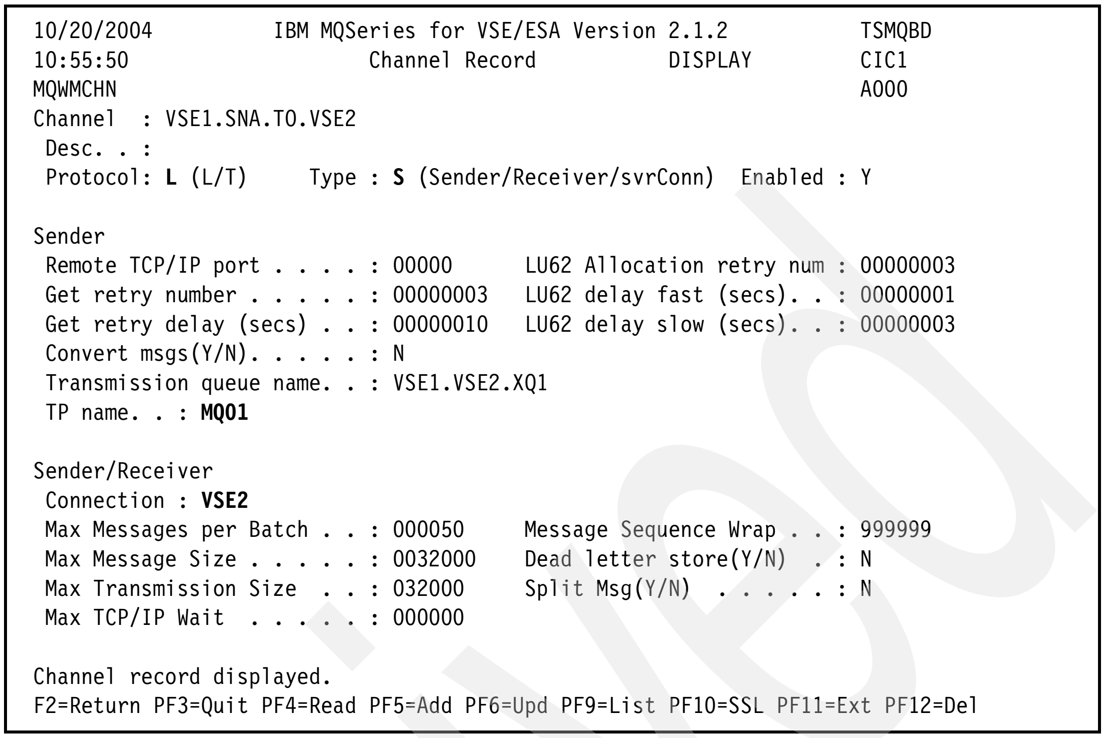
The *Protocol* parameter for LU 6.2 sender channels should be set to `L` for LU 6.2. The sender MCA examines this value to decide which protocol to use when communicating with the remote host. The *Type* for sender channels is `S`.

The *Connection* parameter is the name of your CICS connection definition. If you are defining your MQSeries channels before your connection and session definitions ensure that you create a connection with the name you choose for this parameter. You can change your channel definition later.

The *TP Name* value must identify a task ID for the remote receiver MCA. For a remote system running MQSeries for VSE, this value should be `MQ01`, which is the transaction name for the receiver MCA. For non-VSE MQSeries systems, this may be a TP profile name that identifies the receiver MCA program on that host. For example, on OS/2, the receiver MCA program is AMQCRS6A.EXE. Consequently, in the case of sending to MQSeries on OS/2, the TP Name for your LU 6.2 sender channel should identify a TP profile that includes a program specification for AMQCRS6A.EXE. You should consult relevant SNA communications documentation for defining such profiles on the remote host.

The *Get retry number* and *Get retry delay* parameters affect the operation of the sender MCA. The sender MCA continues to take messages from the transmission queue until the queue is empty. At this point, the sender MCA waits the delay time and reexamines the transmission queue. The sender MCA will wait and reexamine the transmission queue up to the get retry number for a message to arrive. If no messages arrive, the sender MCA terminates.

Figure 6-6 on page 110 shows an example of an LU 6.2 sender channel with key fields in bold:



```
 10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
 10:55:50                   Channel Record          DISPLAY        CIC1
 MQWMCHN                                                           A000
 Channel  : VSE1.SNA.TO.VSE2
  Desc. . :
  Protocol: L (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

 Sender
  Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000003
  Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000001
  Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000003
  Convert msgs(Y/N). . . . . : N
  Transmission queue name. . : VSE1.VSE2.XQ1
  TP name. . : MQ01

 Sender/Receiver
  Connection : VSE2
  Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
  Max Message Size . . . . . : 0032000    Dead letter store(Y/N)  . : N
  Max Transmission Size  . . : 032000     Split Msg(Y/N)  . . . . . : N
  Max TCP/IP Wait  . . . . . : 000000

 Channel record displayed.
 F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 6-6 Example of an LU 6.2 sender channel definition

In this example, sender channel VSE1.SNA.TO.VSE2 specifies an LU 6.2 communication path to a remote VSE system using CONNECTION VSE2. This connection identifies a CICS APPLID as the VTAM netname, and so in this case, transaction MQ01 is initiated when the session is connected by the sender MCA. From that point on, the sender and receiver MCAs can communicate over the allocated session.

TCP/IP sender channels

Unlike LU 6.2 channels, the complete definition of TCP/IP channels is contained within the sender channel definition. There are no additional definitions required within CICS. (Other than the CSD definitions required for the TCP/IP product, such definitions should have been established during TCP/IP installation for CICS.)

In terms of communication with a remote host, the most significant values for TCP/IP sender channels are:

- Protocol
- Port
- Type
- Connection
- Get retry number
- Get retry delay

The *Protocol* parameter for TCP/IP sender channels should be set to T for TCP/IP. The sender MCA examines this value to decide which protocol to use when communicating with the remote host.

The *Port* number identifies the TCP/IP port to communicate with when establishing a connection with the remote host. For MQSeries, the default port number for TCP/IP connections is 1414. It is not mandatory to use this value. However, it is important that you specify a port number that is actively bound by an MQ listener program on the remote host. If it is not, an MQ connection will not be established. For this reason, MQ administrators must agree on MQ port numbers across platforms. It is also important to note that some port numbers are reserved for common services (for example, FTP, Telnet, SENDMAIL, and so forth).

The *Type* parameter for sender channels is always S (for sender).

The *Connection* parameter identifies the remote host's name or IP address. An IP address is a 4-part dotted decimal value that uniquely identifies a host to the TCP/IP network, for example, 123.1.2.254. The partner value can optionally be a host name rather than an IP address. In this case, it is important that your TCP/IP configuration is correct so that the sender MCA can resolve the actual IP address from the specified host name.

The *Get retry number* and *Get retry delay* parameters have the same effect and relevance as LU 6.2 channels. They allow the sender MCA to retry a transmission queue after it has been detected as empty. If after it retries it remains empty, the sender MCA terminates.

Sender and receiver channel definitions always come in pairs. In other words, for a given sender channel definition, there must be a receiver channel definition on the remote host with the *same* channel name. This is so that the receiver MCA can identify the correct channel to use in managing the communication path.

Figure 6-7 shows an example of a TCP/IP sender channel with key fields in bold.

```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
13:58:26                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A000
Channel  : CH1.VSE1.TO.NT1
 Desc. . :
 Protocol: T (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 01414      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : Y
 Transmission queue name. . : TQ1.VSE1.TO.NT1
 TP name. . :

Sender/Receiver
 Connection : 123.1.2.254
 Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0010000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 6-7 Example of a TCP/IP sender channel definition

In this example, sender channel CH1.VSE1.TO.NT1 specifies a TCP/IP communication path to TCP/IP host 123.1.2.254, and communication is to be attempted via TCP/IP port number 1414.

For this channel to be effective, the following must be true:

- An MQ application program must put a message on a remote queue.
- The remote queue definition must identify a local transmission queue.
- The local transmission queue must identify sender channel CH1.VSE1.TO.NT1. It must also specify a trigger for program MQPSEND.
- MQSeries must be running on host 123.1.2.254.
- MQSeries on host 123.1.2.254 must be configured to accept communication requests on port 1414.
- MQSeries on host 123.1.2.254 must have a correctly defined receiver channel named CH1.VSE1.TO.NT1.

6.3.2 Receiver channels

Receiver channels define a communication path *from* a remote host. The significant values in the channel definition vary depending on the communication protocol selected for the communication path. Once again, for MQSeries for VSE, there are two communication protocols available:

- SNA LU 6.2
- TCP/IP

LU 6.2 receiver channels

You do not need connection and session definitions for LU 6.2 receiver channels. However, you do need valid VTAM definitions that recognize any two hosts intended for remote MQSeries queuing. Your VTAM system programmer can verify that your MQSeries hosts can converse over an SNA network.

LU 6.2 receiver channels are passive in the sense that they do not identify a remote host, nor do they identify a TP Name. They are used by the receiver MCA to determine communication parameters, such as message batch sizes, transmission sizes, and maximum message lengths.

In terms of communication with a remote host, the most significant values for LU 6.2 receiver channels are:

- Protocol
- Type

The *Protocol* parameter for LU 6.2 receiver channels should be set to `L` for LU 6.2, and the *Type* parameter is always `R`.

Figure 6-8 on page 113 shows an example of an LU 6.2 receiver channel with key fields in bold.

```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
14:43:52                   Channel Record          DISPLAY        CIC1
MQWMCHN                                                           A000
Channel  : VSEP.SNA.TO.VSE1
 Desc. . :
 Protocol: L (L/T)     Type : R (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000003
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000001
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000003
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0005000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032000     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 6-8 Example of an LU 6.2 receiver channel definition

For LU 6.2 receiver channels, the Port, Partner, Transmission Queue Name and TP Name parameters are not relevant. The receiver MCA is always started, provided the SNA link between the sender and receiver hosts is correctly defined in VTAM.

TCP/IP receiver channels

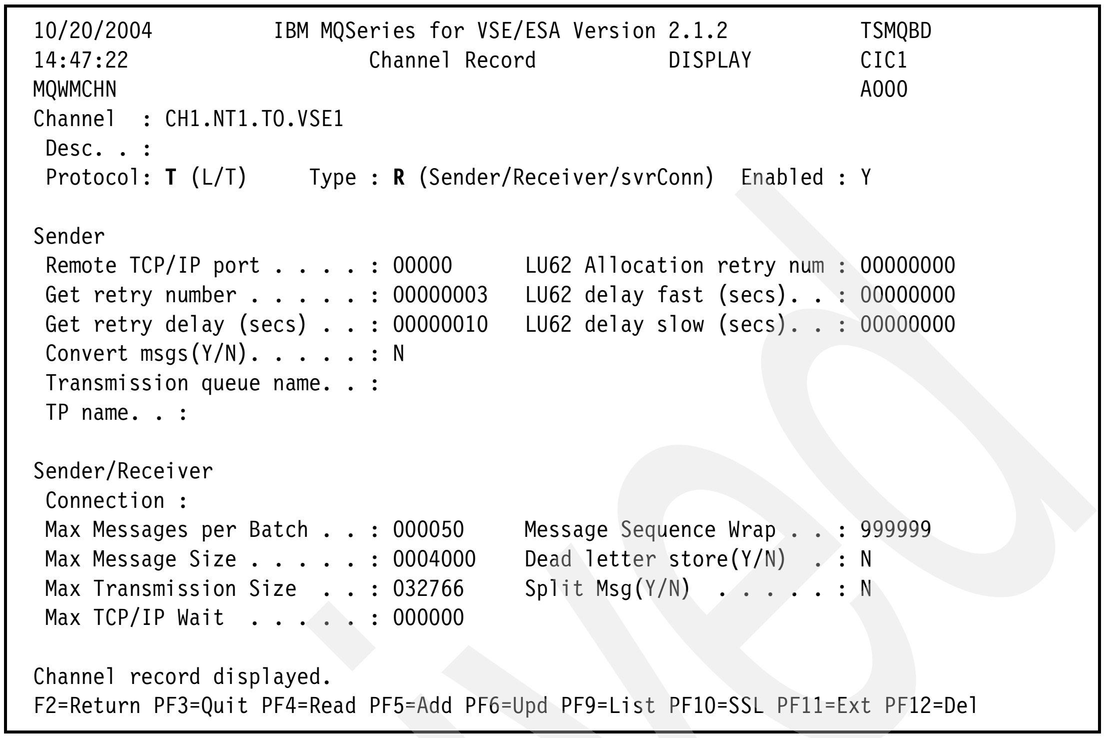
Like LU 6.2 receiver channels, TCP/IP receiver channels are passive in the sense that they do not identify a remote host. They are used by the receiver MCA to determine communication parameters, such as batch sizes, transmission sizes, and maximum message lengths.

In terms of communication with a remote host, the most significant values for TCP/IP receiver channels are:

- Protocol
- Type

The Protocol parameter for TCP/IP receiver channels should be set to `T` for TCP/IP, and the Type parameter should always be `R`.

Figure 6-9 on page 114 shows an example of a TCP/IP receiver channel with key fields in bold.



```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
14:47:22                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A000
Channel  : CH1.NT1.TO.VSE1
 Desc. . :
 Protocol: T (L/T)     Type : R (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000050     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0004000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 6-9 Example of a TCP/IP receiver channel definition

For TCP/IP receiver channels, the Port and Partner parameters are not relevant. The receiver MCA will accept connections from any host that issues a TCP/IP connect request to the configured port number for the local queue manager. For MQSeries for VSE, the port number for the queue manager is specified in the global system definition.

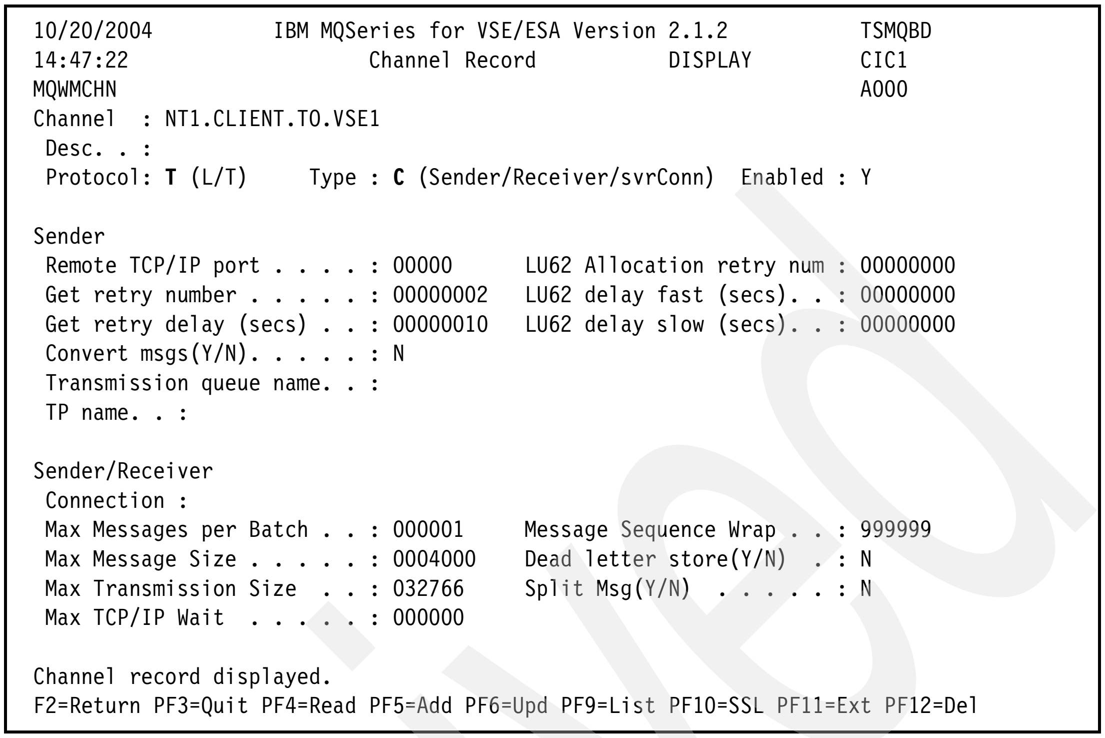
6.3.3 Client channels (SVRCONN)

Client channels are similar to receiver channels. For MQSeries for VSE, client channels must be TCP/IP channels, because this is the only protocol implemented to support client connections.

Like TCP/IP receiver channels, client channels do not specify a partner or port number. The receiver MCA automatically detects that a connection is for a client program and starts a server program under CICS. The client channel definition is used by the server program to determine communication parameters, such as timeout and retry values. Because client channels are not used for message transmission, values such as batch size, maximum message size, and sequence wrap are irrelevant.

With client channels, there is no need for a definition pair as there is with sender and receiver channels. The client channel definition only needs to exist for the server queue manager. There is no need to mirror the definition on the client queue manager. In fact, the client may not have a local queue manager.

Figure 6-10 on page 115 shows an example of a TCP/IP client channel with key fields in bold.



```
10/20/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
14:47:22                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A000
Channel  : NT1.CLIENT.TO.VSE1
 Desc. . :
 Protocol: T (L/T)     Type : C (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000002   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0004000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure 6-10   Example of a TCP/IP client channel definition

Instances of sender and receiver channels cannot be concurrent. In other words, if sender channel CHAN1 is active, a second instance of CHAN1 cannot be started. If an attempt is made, for example another transmission queue uses CHAN1, the error `CHANNEL BUSY` is raised. This is not the case with client, or SVRCONN, channels. Multiple instances of a client channel can be active at the same time.

6.4 Message channel agents and communication programs

Message channel agents are programs that facilitate distributed queuing. In MQSeries for VSE, there are two Message Channel Agents (MCA):

- Sender MCA
- Receiver MCA

For communication purposes, and specifically for the MQSeries TCP/IP implementation, there are two additional programs worthy of note:

- TCP/IP listener
- TCP/IP server

Each of these programs is explained in further detail in the following sections.

6.4.1 Sender MCA

For MQSeries for VSE, the sender MCA is program MQPSEND, which runs as transaction MQ02. MQPSEND is intended to be a trigger program initiated when messages are written to a transmission queue. Messages are written to a transmission queue when an MQ application program issues an MQPUT or MQPUT1 call to a remote queue.

If the transmission queue is defined correctly, the MQPSEND program is initiated automatically as soon as the first message is written to the transmission queue. When a batch of messages are written to the transmission queue, only one instance of the sender MCA is instantiated. In other words, MQPSEND runs only once while a transmission queue is logically active. There is a timeout period while MQPSEND waits for more messages to arrive on the transmission queue. Once this timeout has expired, the queue is considered logically inactive and the sender MCA terminates. The sender MCA is restarted whenever a transmission queue receives a new message or batch of messages.

The task of the sender MCA is to establish a connection with a remote host and, subsequently, send messages from the transmission queue to a receiver MCA running on the remote host. The relevant transmission queue is the one that caused the sender MCA to be initiated in the first place.

To establish a connection with a remote host, the sender MCA uses the channel definition identified in the transmission queue definition. The channel definition specifies all necessary details for locating and communicating with the remote host. Once a connection is established, the sender and receiver MCAs exchange initial negotiation parameters to determine how the connection will operate. If these negotiations are accepted by both hosts, the sender MCA begins to send the messages from the local transmission queue. The receiver MCA puts these messages on a local queue (local definition to the receiver MCA) identified during the initial negotiations.

In the event that a connection cannot be established with the intended remote host, messages remain on the local transmission queue. Such messages remain on the transmission queue until the queue is stopped and restarted or subsequent messages are written to the transmission queue. For this reason, it is best to specify a trigger type of `E` in the transmission queue extended definition. If you specify a trigger type of `F` for the sender MCA, messages pending on a transmission queue will not be processed until the queue is stopped and restarted. This is because a trigger type of F tells MQSeries to only start the sender MCA when the queue message count moves from 0 to 1.

6.4.2 Receiver MCA

For MQSeries for VSE, the receiver MCA is the MQPRECV program, which runs as transaction MQ01. Depending on the communication protocol in use to connect the sender and receiver hosts, the MQPRECV program is initiated in one of two ways. For SNA LU 6.2, MQPRECV is started automatically by VTAM and CICS. For TCP/IP, MQPRECV is started by the TCP/IP listener program. For this reason, the TCP/IP listener program must be running at all times in CICS if TCP/IP connections are to be established. This is true for client connections as well. Regardless of which way the receiver MCA is started, the connection is always initiated by the sender MCA.

Once started, the receiver MCA exchanges negotiation parameters with the sender MCA. Included in these parameters is the channel name applicable to the connection. The channel definition must exist for the receiver MCA queue manager. The channel definition specifies communication parameters that determine the operation of the connection. If the negotiations are accepted by both MCAs, the receiver begins to receive messages sent by the sender MCA.

Messages received by the receiver MCA are written to a local queue (local definition to the receiver queue manager) identified during the initial negotiations. If, for any reason, messages cannot be written to the target queue, the receiver MCA puts the messages on a dead letter queue. The dead letter queue is identified in the global system definition and should be a queue locally defined to the receiver queue manager. In the event that the dead letter queue is not properly specified or does not exist, the receiver MCA notifies the sender

MCA that an error has occurred and the messages are retained on the sender's transmission queue. Consequently, you should always define and maintain your dead letter queues.

Messages received by the receiver MCA are committed by SYNCPOINT after the successful receipt of a message batch. The number of messages in a batch is specified in the channel definition. The receiver MCA notifies the sender that the batch has been successfully committed, at which point, the sender also commits the deletion of messages from the transmission queue. MCAs also use a Message Sequence Number (MSN) to keep track of messages successfully sent between sender and receiver MCAs. The MSN ensures duplicates are not sent after an abnormal termination between SYNCPOINTS. This means messages cannot be lost in the event of an abnormal program termination at either end of the communication link.

The receiver MCA continues to process messages until the sender MCA notifies the receiver that all messages have been sent. The receiver acknowledges the completion and both MCAs terminate. At this point, all the messages originally put on the remote queue are now on the local queue (local definition) of the remote queue manager.

An additional task of the receiver MCA is to determine whether a connection is from a client program. This is determined during initial negotiation. A client program is one that functions as though the connected queue manager is local to the program when, in fact, it is remote. This fact is transparent to the client program.

The receiver MCA does not manage a conversation with the client program as though it is a sender MCA. Instead, it starts a server program to handle the conversation. The server program continues to run under transaction MQ01. There is one instance of the server program for each active client connection.

6.4.3 TCP/IP listener

The TCP/IP listener program is MQPTCPLN that runs as transaction MQTL. Its purpose is to wait for TCP/IP connection requests on a configurable TCP/IP port and start the receiver MCA. It then continues to wait for further connection requests. It continues to run until MQSeries is stopped. If the listener program is not running on your CICS system, no TCP/IP connections are possible.

The listener program starts automatically when MQSeries is started if a port number is specified in the global system definition. If the port number in the global system definition is set to `0`, the listener will not start. When automatically started at MQSeries startup, the listener program is passed the port number and a special flag that the listener program obtains using EXEC CICS RETRIEVE. For this reason, the MQTL transaction should not be started manually in native CICS.

The special flag is a binary value that tells the listener whether to start forcibly. If the flag is true, the listener will search for a TCP/IP socket bound to the specified port number and close it. It will then attempt to bind to the port number itself. The listener can be started forcibly via the MQIT transaction (MQSeries startup transaction) with the `F` option.

For example:

```
MQIT F
```

This can only be done from native CICS after MQSeries has been stopped. You may need to start the listener program forcibly if the listener continually reports a BIND error to the system log on startup.

In a production system, it may be undesirable to stop MQSeries just to start or restart the listener program. It is possible to get around this limitation by using the CICS CECI transaction. For example:

```
start trans(MQTL) fr('0000')
STATUS:  COMMAND EXECUTION COMPLETE                        NAME=
 EXEC CICS  START
  TRansid( X'D4D8E3D3' )
  < Interval( X'0000000C' ) | TIme() | (( AFter | AT ) < Hours() >
    < Minutes() > < SEconds() > > )
  < FRom( X'05860000' ) < Length( X'0004' ) < FMh > > >
  < TErmid() >
  < SYsid() >
  < RTRansid() >
  < RTErmid() >
  < Queue() >
  < Nocheck >
  < Protect >
  < REqid() >
```

Note that the display has been switched to HEX (using **PF2**). This allows you to overtype the *from* field with a hexadecimal value. The first two bytes represent the port number as 1414 (hexadecimal 0586); the second two bytes represent the force flag as false (hexadecimal 0000). To start the listener forcibly, the force flag should be set to true (hexadecimal 0001).

For MQSeries for VSE, you can run only one TCP/IP listener program per queue manager. However, if you have multiple CICS systems on your VSE system, each with its own MQSeries, you can run one listener program per CICS system. The only limitation on this is that each listener program must be configured to use a different TCP/IP port number.

6.4.4 TCP/IP server

The TCP/IP server program is MQPTCPSV and runs as transaction MQ01 (the receiver MCA transaction). Its purpose is to process MQ client connections.

Client connections are initiated when a client program on a remote host attempts to connect to a queue manager. At this point, the MQI call (MQCONN) attempts to establish a communication link with the remote host using configuration parameters on the client machine. For MQSeries for VSE, such connections are only possible using TCP/IP. The configuration parameters identify the remote host name or address, TCP/IP port number, and the queue manager name on the remote host. They also identify a channel name for the remote host to use for the overall operation of the communication link.

When a client program is attempting to connect to a VSE queue manager, the connection can only be successful if the TCP/IP listener program is running under CICS on the VSE host. This is transaction MQTL. In addition, MQSeries must be configured to accept TCP/IP connections on the port number specified by the client configuration parameters.

If the client parameters match the remote queue manager’s configuration, the connection request is accepted by the listener program and the receiver MCA is started (program MQPRECV). At this point, the TCP/IP connection is established. The client program, still in the MQCONN call, then exchanges MQ negotiation details with the receiver MCA. The initial negotiation data includes a flag that tells the receiver MCA that the connected program is a client. The receiver MCA then starts the TCP/IP server program to process the client connection.

Having established a TCP/IP and MQ client/server connection, the MQCONN call proceeds to request a connection to the VSE queue manager. From this point on, the server program acts as a proxy to the client program, issuing MQI calls on its behalf. The results of each call are returned to the client program. This cycle continues until the client program issues an MQDISC or the TCP/IP connection is broken due to an unexpected error.

6.5 Communication protocols

The two communication protocols available to MQSeries for VSE 2.1 are SNA LU6.2 and TCP/IP. Which protocol is applicable is determined by your channel definitions and, consequently, each individual MCA or client connection. There is no reason why MQSeries for VSE cannot manage both protocols simultaneously as long as your VSE system is configured for both protocols.

In terms of remote queuing, there are limitations on transmission sizes, which vary between the two protocols. Transmission sizes are also specified in your channel definitions. For SNA LU 6.2, the maximum transmission size is 32000, and for TCP/IP, it is 65535. These are limitations of the protocols themselves, not of MQSeries.

6.6 Clients

For MQSeries for VSE 2.1, there is no such thing as an MQSeries for VSE client. However there is support for remote, or non-VSE, MQ clients. In other words, you can create an MQ client that connects to a VSE queue manager, but you cannot create a VSE client that connects to other queue managers.

To create client programs, MQSeries provides a series of linkable objects that include the additional communication logic required by clients issuing MQI calls. These objects do not exist for MQSeries for VSE.

Support for non-VSE clients is provided by the TCP/IP server program MQPTCPSV described in 6.4.4, “TCP/IP server” on page 118. An important function performed by the server program is code page conversion.

6.6.1 Client code page conversion

A problem exists in remote queuing when communicating hosts are using different code pages. The problem is that data, such as the initial negotiation parameters, can not be interpreted by either MCA if they are using different code pages. To get around this problem, the initial negotiation parameters include code page details so that conversion can take place. In the MQSeries client/server model, it is always the responsibility of the server to convert MQ system data, such as the initial negotiation parameters and subsequent MQ header data.

To convert MQ system data, MQSeries for VSE uses Language Environment/VSE services. This means the client code page and the server code page must represent a pair that Language Environment/VSE is capable of converting. Language Environment/VSE is shipped with a number of default conversion tables. These are documented in the *Language Environment V1R4 C Run-Time Programming Guide*, SC33-6688.

Client programs are not limited to the default code page conversion tables supplied with Language Environment/VSE. It is possible to create your own conversion tables using the Language Environment/VSE code page conversion utilities.

Creating code page conversion tables

The code page conversion utilities can be reviewed in the *Language Environment V1R4 C Run-Time Programming Guide*, SC33-6688. For convenience, we will go through the steps involved.

Before we begin, we need to be aware of the various aspects involved in the process. These are:

- Code page numbers
- Code page translation table naming conventions
- Code page translation table source files
- The GENXLT utility
- CSD definitions

Code pages have their own unique number, for example, 037 for USA and Canada, 273 for Germany and Austria, and 285 for the United Kingdom. You should determine the numbers of the code pages you want to convert. Refer to Figure 2-1 on page 32 for a list of standard code pages supported by Language Environment/VSE. Note that MQSeries for VSE only supports conversion for code pages identified by numbers. For example, ISO8859-1 is not supported.

This is because MQSeries identifies code pages using a numeric value passed in data headers when the client connects to the server. Although code page names like ISO8859-1 are not supported, such code pages often have a numeric alias. For example, 819 is an alias for ISO8859-1. So if you want to use code page ISO8859-1, the client program need to be running with code page 819. There are several ways to do this. One is to set the MQCCSID to 819 in the program's environment. For example:

```
c:\>set MQCCSID=819
```

If your client program uses Java classes, you can set the CCSID attribute of the MQEnvironment object. For example:

```
MQEnvironment.CCSID = 819;
```

The Language Environment/VSE source file EDCUCSNM.A, in PRD2.SCEEBASE, documents some of the code page aliases.

Code page translation tables have a strict naming convention. This is because the appropriate translation table name is built dynamically by Language Environment/VSE based on the two code page numbers involved in the translation. Translation tables follow the naming convention EDCU*fftt*, where *ff* is a two character synonym for the *from* code page, and *tt* is a two character synonym for the *to* code page. For example, the two character synonym for code page 37 is EA and the synonym for code page 273 is EB. Consequently, the translation table name for 37 to 273 is EDCUEAEB and the translation table for 273 to 37 is EDCUEBEA.

The two character synonyms for code page numbers are documented in the *Language Environment V1R4 C Run-Time Programming Guide*, SC33-6688. They are also coded in a phase used by Language Environment/VSE to dynamically build the translation table name from the code page numbers. This phase is EDCUCSNM and it is used by Language Environment/VSE when code page conversion services are employed. When these services are used, as they are by MQSeries for VSE, only the code page numbers are identified. Language Environment/VSE uses the EDCUCSNM to dynamically build the name of the translation table that is subsequently used for code page conversion. The following is an example of EDCUCSNM source entries:

```
EDCCSNAM TYPE=ENTRY,CODESET='IBM-037',CODE='EA'
EDCCSNAM TYPE=ENTRY,CODESET='IBM-273',CODE='EB'
EDCCSNAM TYPE=ENTRY,CODESET='IBM-274',CODE='EC'
EDCCSNAM TYPE=ENTRY,CODESET='IBM-275',CODE='ED'
EDCCSNAM TYPE=ENTRY,CODESET='IBM-277',CODE='EE'
EDCCSNAM TYPE=ENTRY,CODESET='IBM-278',CODE='EF'
```

The source for EDCUCSNM is found in PRD2.SCEEBASE library, EDCUCSNM.A file.

Code page translation source files can be found in the PRD2.SCEEBASE library. These source files are prefixed EDCU and have an X extension, for example, EDCUAAEY.X. These can be used as a basis for new conversion tables. The source files contain three columns. The first two columns contain hexadecimal values from 0x00 to 0xFF; the third column documents the relevant character being converted. For example, showing only the first 16 entries:

```
0x00    0x00   <NUL>
0x01    0x01   <SOH>
0x02    0x02   <STX>
0x03    0x03   <ETX>
0x04    0xdc   <SEL>
0x05    0x09   <tab>
0x06    0xc3   <RNL>
0x07    0x1c   <DEL>
0x08    0xca   <GE>
0x09    0xb2   <SPS>
0x0a    0xd5   <RPT>
0x0b    0x0b   <vertical-tab>
0x0c    0x0c   <form-feed>
0x0d    0x0d   <carriage-return>
0x0e    0x0e   <SO>
0x0f    0x0f   <SI>
```

The source files are assembled using the GENXLT utility and then link edited to produce executable phases. Each phase represents a conversion from one code page to another. They are mono-directional, so there should be two phases for each translation needed.

Because MQSeries for VSE runs under CICS, the EDCUCSNM and the translation table phases must be known to CICS. Consequently, you need CSD program entries for these phases. The language type is Assembler, and the CSD group for default tables is CEE. You can use the CEDA transaction to look at existing entries. For example:

```
OBJECT CHARACTERISTICS
 CEDA  View
  PROGram        : EDCUAAEY
  Group          : CEE
  Language       : Assembler          CObol | Assembler | C | Pli | Rpg
  RELoad         : No                 No | Yes
  RESident       : No                 No | Yes
  RSl            : 00                 0-24 | Public
  Status         : Enabled            Enabled | Disabled
 REMOTE ATTRIBUTES
  REMOTESystem   :
  REMOTEName     :
  Transid        :
  Executionset   : Fullapi            Fullapi | Dplsubset
```

With this background, we can now go through the steps involved in creating your own code page conversion table. This process is only necessary if you have an MQ client that uses a code page that does not have a default translation table to and from your VSE code page. In our example, we create conversion tables for 37 to 273 (US to Germany) and 273 to 37 (Germany to US). The steps are as follows:

1. Identify a default translation table source file that approximates the target conversion table. For example, EDCUEA??. We do not care what ?? is because we change this part of the source file. We also need a source file for the reverse translation, so we arbitrarily choose EDCUEAEO and EDCUEBEO. We use the following JCL to get these source files on our reader where we can save them to a disk:

   ```
   // JOB LIBPUN
   // EXEC LIBR
   ACC SUBLIB=PRD2.SCEEBASE
   PUNCH EDCUEAEO.X
   PUNCH EDCUEBEO.X
   /*
   /&
   ```

   If you are using the VSE Interactive User Interface, you can use LIBRP to retrieve these two files.

2. Once we have these files, we need to change the file names from EDCUEAEO to EDCUEAEB and EDCUEBEO to EDCUEBEA. We can now edit the source files to change the second column in each file. The first column is simply values 0x00 to 0xFF and does not need to be changed. To determine the correct values for the second column, one approach is compare each file and identify the hexadecimal values for each character notated in the third column. This is a laborious matching process. When the source files are ready, we need to catalog them in our user library. It is probably best not to put them in PRD2.SCEEBASE.

3. Having created and catalogued our translation source files, we build two executable phases using the GENXLT utility. The following JCL does the job:

   ```
   // JOB GENXLT
   // LIBDEF *,SEARCH=user.lib
   // LIBDEF PHASE,CATALOG=user.lib
   // OPTION LINK,CATAL
   // EXEC EDCGNXLT,PARM='IFILE(DD:user.lib(EDCUEAEB.X)),NODBCS,         X
                  NAME(EDCUEAEB)'
    ENTRY TABLENAM
   // EXEC LNKEDT
   /*
   // EXEC EDCGNXLT,PARM='IFILE(DD:user.lib(EDCUEBEA.X)),NODBCS,         X
                  NAME(EDCUEBEA)'
    ENTRY TABLENAM
   // EXEC LNKEDT
   /*
   /&
   ```

4. We now need to check that the two character synonyms for code pages 37 and 273 are defined in EDCUCSNM. The source for EDCUCSNM is found in PRD2.SCEEBASE. If we need to add entries, they should look like the following:

   ```
   EDCCSNAM TYPE=ENTRY,CODESET='IBM-037',CODE='EA'
   EDCCSNAM TYPE=ENTRY,CODESET='IBM-273',CODE='EB'
   ```

   In our case, both entries are shipped with Language Environment and do not need to be added. If you do change the EDCUCSNM source, it can be rebuilt using the following JCL:

```
// JOB BLDCSNM
// LIBDEF *,CATALOG=user.lib
// OPTION CATAL,LIST
 PHASE EDCUCSNM,*
// EXEC ASMA90
* EDCUCSNM source code goes here
/*
// EXEC LNKEDT
/*
/&
```

5. The applicable phases are now built and ready to use. Before they can be used by MQSeries for VSE, we need to add CSD entries in CICS. There should already be a program entry for EDCUCSNM. You can check this using the CEDA or CEMT transaction. If you have changed EDCUCSNM, you need to load the new program. For example:

   ```
   s prog(EDCUCSNM) new
   STATUS:  RESULTS - OVERTYPE TO MODIFY
    Pro(EDCUCSNM) Len(0000714) Ass Pro Ena Ful                  NEW COPY
       Res(000) Use(000000)
   ```

   You can add the new translation table programs using CEDA. For example:

   ```
   OBJECT CHARACTERISTICS
    CEDA  View
     PROGram        : EDCUEAEB
     Group          : CEE
     Language       : Assembler           CObol | Assembler | C | Pli | Rpg
     RELoad         : No                  No | Yes
     RESident       : No                  No | Yes
     RSl            : 00                  0-24 | Public
     Status         : Enabled             Enabled | Disabled
    REMOTE ATTRIBUTES
     REMOTESystem   :
     REMOTEName     :
     Transid        :
     Executionset   : Fullapi             Fullapi | Dplsubset
   ```

6. It should be noted that in this example, we used user.lib for our source and phases. This was done to avoid putting non-Language Environment files in the PRD2.SCEEBASE library. Since we are using user.lib, we must make sure user.lib is in the LIBDEF for our CICS startup JCL. If we have changed EDCUCSNM, then the user.lib must appear before PRD2.SCEEBASE in the LIBDEF. For example:

   ```
   // LIBDEF     *,SEARCH=(user.lib,
                   PRD2.MQSERIES,
                   PRD2.CONFIG,
                   PRD1.BASE,
                   PRD2.SCEEBASE,
                   PRD2.PROD,
                   PRD2.DBASE)
   // EXEC DFHSIP,SIZE=8M,PARM='SIT=...'
   ```

   Otherwise, CICS will use the old EDCUCSNM phase in PRD2.SCEEBASE. If you have changed your LIBDEF, you must stop and restart CICS before you can use the new conversion tables.

This completes our example. MQSeries for VSE now accepts clients using code page 37 when the server code page is 273, and clients using code page 273 when the server code page is 37.

7

Extended channel features

In this chapter, we look at some of the features relevant to channels and channel operation. Specifically, we examine:

- SSL-enabled channels
- Channel exits

Each of these features is also described in the *MQSeries for VSE System Management Guide*, GC34-5364. In this section, we step through actual examples.

7.1 SSL-enabled channels

SSL stands for Secure Sockets Layer. As described in 9.5, “SSL-enabled channels” on page 181, SSL provides a means of encrypting data sent across a TCP/IP network. You can use SSL-enabled channels to ensure messages sent across an open network, like the internet, are secure. Secure in this sense means that even if the messages are intercepted, they are encrypted. You can also use SSL-enabled channels to verify that the remote partner is trusted—that is, they are known to the local queue manager.

SSL uses the Public Key Infrastructure (PKI). PKI involves X.509 certificates that contain information about the user. It also involves public and private keys. These keys are essentially very large integer numbers. The keys are used to encrypt and decrypt data sent over an SSL connection.

With MQSeries, if you plan to use SSL-enabled channels, you need your own digital certificate with its associated public and private keys. When you connect to or from another queue manager using an SSL-enabled channel, the two queue managers exchange certificates. This way each queue manager knows the remote partner's public key and can use it to encrypt data. The partner can then use its private key to decrypt the data. It is generally accepted that current cryptographic technology ensures that data encrypted with a public key can only be decrypted with a corresponding private key.

Generally, certificates are created in three phases. Firstly, you generate the public and private keys. These are always created together because they work in conjunction with each other. Secondly, the public key is used to build a new certificate request. Finally, the new certificate request is used to create the certificate itself. This last stage is generally performed by a Certificate Authority (CA). A Certificate Authority is a third party organization that specializes in this area, and provides a range of certificate services.

When the certificate is created, the certificate issuer (the CA) digitally “signs” the certificate. The public keys of known Certificate Authorities are generally available, and these can be used by SSL services to verify that the certificate was indeed issued by the CA.

On most MQSeries platforms that support SSL-enabled channels, the queue manager has a key repository that contains the public keys of recognized Certificate Authorities. Each platform has its own tools and services to create certificates and manage the public and private keys.

In the following sections, we provide an example of setting up an SSL-enabled channel, and step through the process of creating a digital certificate.

7.1.1 Creating a CA-issued certificate

For the purpose of explanation, we will step through an example of creating a CA-issued certificate using services available with the SSL for VSE product, provided by CSI International. If you are using TCP/IP and SSL services from a different supplier, you should examine their documentation for procedures equivalent to those discussed here.

As a prerequisite, SSL for VSE must be installed and available on your VSE system. Your system programmer should take care of this. Core elements of the SSL service are shipped with the TCP/IP for VSE product. Additional elements are available for download from CSI International. In particular, you can download zip file SSL4VSE.ZIP.

The SSL4VSE.ZIP file contains utilities and sample jobs to create and manage certificates. Instructions on how to use these utilities and samples are included in the ZIP file. The first step in these instructions is to run sample job $SSL4VSE.JCL. This job catalogs a PROC needed by the other sample jobs.

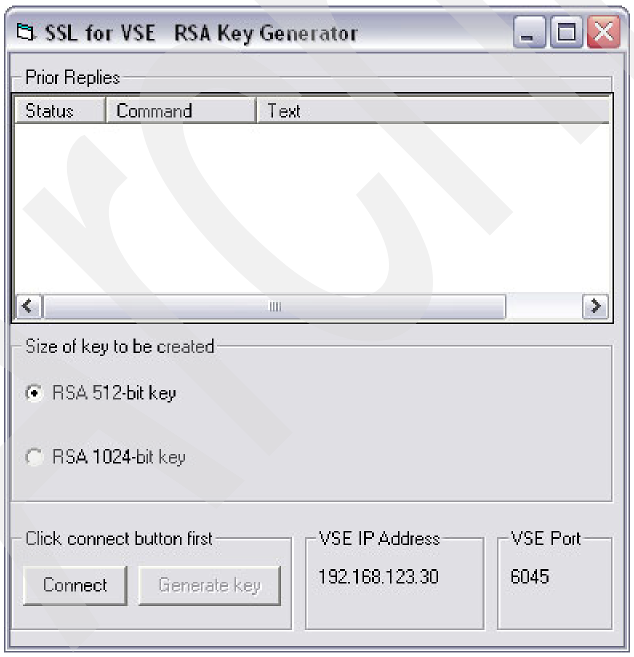
As already mentioned, the first step in creating a certificate is to generate a public and private key. To do this, you can run sample job CIALSRVR.JCL on your VSE system.

```
// JOB CIALSRVR
// OPTION SYSPARM='00'
// EXEC PROC=$SSL4VSE
// LIBDEF PHASE,SEARCH=PRD1.BASE
// EXEC CIALSRVR,SIZE=CIALSRVR,PARM='MQM.SSLKEYS.MQVSEC'
SETPORT 6045
/*
/&
```

This job starts a server program running in a batch partition. The program waits for client requests to generate and store public/private key details in the SSL key-ring sublibrary. The PARM on the EXEC card identifies the key-ring sublibrary, MQM.SSLKEYS, and the target member name, MQVSEC, to store our public and private keys.

On the client side, in our case on Windows XP, we need to install the client program. We need to run SETUP.EXE, provided in the ZIP file, to install the client key generating program. This will install program CIALCLNT.EXE in a designated directory on Windows XP. The installation directory also includes a file called VSESYS.TXT. This file must be modified to identify the IP address of your VSE system and the port number being used by the server program. As per the JCL to run the server, we are using port number 6045.

Because the server is already running, we can now run CIALCLNT.EXE which displays the window shown in Figure 7-1:


SSL for VSE   RSA Key Generator
Prior Replies
Status
Command
Text
Size of key to be created
RSA 512-bit key
RSA 1024-bit key
Click connect button first
Connect
Generate key

VSE IP Address: 192.168.123.30

VSE Port: 6045

Figure 7-1   SSL for VSE RSA Key Generator window

If we have modified the VSESYS.TXT file to correctly identify our VSE system, clicking the **Connect** button causes the client program to connect to the server running on VSE. We can then use the radio buttons and the **Generate key** button to generate a 512-bit or a 1024-bit key.

The choice of key size is important because it determines which cipher specifications we can use later on. For example, you can only use Triple DES encryption with a 1024-bit key.

After you have generated the key, you are given to option to send it to VSE. When it has been sent, the server will create a new member, in our case with the name MQVSEC, in the key-ring sublibrary. Once the key member is created, the server job ends automatically, and you can exit the client program.

Next, we run sample job CIALCREQ.JCL provided in the ZIP file. The CIALCREQ job allows us to generate a new certificate request and specify identifying information that will be included in the certificate when it is created.

```
// JOB CIALCREQ
// OPTION SYSPARM='00'
// EXEC PROC=$SSL4VSE
// LIBDEF PHASE,SEARCH=PRD1.BASE
// EXEC CIALCREQ,SIZE=CIALCREQ,PARM='MQM.SSLKEYS.MQVSEC'
Common-name: w3.ibm.com/mq/2004/11/17
Organization Unit: MQ/VSE Development
Organization: IBM GSA (APC)
Locality: Perth
State: Western Australia
Country: AU
/*
/&
```

When we run this job, if successful, the job output will include the new certificate request. The request is generated in universally printable characters. In our case, the output appears as follows:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBOjCCATsCAQAwgZExCzAJBgNVBAYTAkFVMRowGAYDVQQIExFXZXNOZXJuIEF1
c3RyYWxpYTEOMAwGA1UEBxMFUGVydGgxFjAUBgNVBAoTDUlCTSBHUOEgKEFQQykx
GzAZBgNVBAsTEk1RL1ZTRSBEZXZlbG9wbWVudDEhMB8GA1UEAxMYdzMuaWJtLmNv
bS9tcS8yMDA0LzExLzE3MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCruxDU
GJYtiY2aK2s7kbpexCrRnPe3PTVfZz7XMlBW5BDJWvP3Hz6bBJkXXhGo4C+5d9rM
5HQOfH3yYjU9yajY3JPgnNaTYcNzjwItYIo5/Qfdf28ZnhyXRAEPmYDNzHKtLSE9
k9ZHOAMMNtj1UoM+wQzCzEdrjF9lqj6NF5ATDQIDAQABoAAwDQYJKoZIhvcNAQEE
BQADgYEAk6IbUOcdhY70SbTq9Bc9zI3Y1WIw/QFnXT7YfneKrjKuSafZKT1q3As/
l74j7WUPudww+BnavBVtyoeoehQ/fgIZI9RDqvhXJemy/WMlx75BczVQx7S23o2C
5nu1Vn2LlW/NfWulkigi5t7KoEBbXpPtSXHmw86b/rSuB27hPSA=
-----END NEW CERTIFICATE REQUEST-----
```

We can now use this request and apply for a certificate from a Certificate Authority. Each CA has its own procedures for creating a signed certificate. Often, these procedures are available over the internet. As some point in the procedures, the CA will ask for the new certificate request. At this time you can cut and paste the request into the online dialogue.

The CA will then create the certificate and send it electronically. Once again, the certificate is sent as a universally printable text, for example:

```
-----BEGIN CERTIFICATE-----
MIICeTCCAiMCEEfJSJ+jttwohQ2J1XaFCw8wDQYJKoZIhvcNAQEEBQAwgakxFjAU
BgNVBAoTDVZlcmlTaWduLCBJbmMxRzBFBgNVBAsTPnd3dy52ZXJpc2lnbi5jb20v
cmVwb3NpdG9yeS9UZXNOQ1BTIEluY29ycC4gQnkgUmVmLiBMaWFiLiBMVEQuMUYw
RAYDVQQLEz1Gb3IgVmVyaVNpZ24gYXVOaG9yaXplZCBOZXNOaW5nIG9ubHkuIE5v
MDAzLzAyLzAOMIGfMAOGCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMAHNzmhJzetdM
CSkO4D3EUF/xJ2IBUCWjMkU4kerzoqccfKGSO8mlYE+F+UgazgTZYlF7HKl+zPUm
ggepytS+ofBulW8en4tXFtBIjXO5dSotUmLWqTfT8e7nBLP5PxVfPOnaeU4cAzFv
4NkHBE8WQazXppFoeLbcOYNh1ZSSuwIDAQABMAOGCSqGSIb3DQEBBAUAAOEAQAsy
ZF8VxtoaOGjJAwMQEOs1YB/GfzedaXEOFHkDIAogUOoHLvuVZOZGopAwuYsyBROZ
naDu1guD2xcNmuWukA==
-----END CERTIFICATE-----
```

We can now transfer the certificate to VSE using sample job CIALCERT.JCL. The CIALCERT job stores the certificate in the VSE key-ring sublibrary. We need to cut and paste the certificate sent by the CA into the CIALCERT JCL, and make sure the PARM on the EXEC card names our target certificate name, MQVSEC.

```
// JOB CIALCERT
// OPTION SYSPARM='00'
// EXEC PROC=$SSL4VSE
// LIBDEF PHASE,SEARCH=PRD1.BASE
// EXEC CIALCERT,SIZE=CIALCERT,PARM='MQM.SSLKEYS.MQVSEC'
-----BEGIN CERTIFICATE-----
MIICeTCCAiMCEEfJSJ+jttwohQ2J1XaFCw8wDQYJKoZIhvcNAQEEBQAwgakxFjAU
BgNVBAoTDVZlcmlTaWduLCBJbmMxRzBFBgNVBAsTPnd3dy52ZXJpc2lnbi5jb20v
cmVwb3NpdG9yeS9UZXNOQ1BTIEluY29ycC4gQnkgUmVmLiBMaWFiLiBMVEQuMUYw
RAYDVQQLEz1Gb3IgVmVyaVNpZ24gYXVOaG9yaXplZCBOZXNOaW5nIG9ubHkuIE5v
IGFzc3VyYW5jZXMgKEMpVlMxOTk3MB4XDTAzMDIwNDAwMDAwMFoXDTAzMDIxODIz
BAsUEk1RL1ZTRSBEZXZlbG9wbWVudDEhMB8GA1UEAxQYdzMuaWJtLmNvbS9tcS8y
MDAzLzAyLzAOMIGfMAOGCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMAHNzmhJzetdM
CSkO4D3EUF/xJ2IBUCWjMkU4kerzoqccfKGSO8mlYE+F+UgazgTZYlF7HKl+zPUm
ggepytS+ofBulW8en4tXFtBIjXO5dSotUmLWqTfT8e7nBLP5PxVfPOnaeU4cAzFv
4NkHBE8WQazXppFoeLbcOYNh1ZSSuwIDAQABMAOGCSqGSIb3DQEBBAUAAOEAQAsy
naDu1guD2xcNmuWukA==
-----END CERTIFICATE-----
/*
/&
```

Our MQSeries for VSE queue manager now has a signed certificate ready for use by SSL-enabled channels. The next step is to configure MQSeries and a channel for SSL. But first, we will look at an alternative way of creating certificates-self-issued, or self-signed, certificates.

7.1.2 Creating a self-issued certificate

Rather than using a third-party Certificate Authority, you can create and sign your own certificates. For this example, we use the Keyman/VSE product, which can be downloaded from the IBM Web site.

Keyman/VSE is a VSE-specific SSL key management tool. It can directly upload generated RSA keys and certificates to VSE and keep the server side key-ring in sync with client side key-ring files.

Keyman/VSE is a Java-based tool, and requires Java runtime 1.4 or later as a prerequisite. It also uses the VSE Connector client and the VSE Connector server. The VSE Connector server is provided with VSE 2.5 and later. The VSE Connector client can be downloaded from the IBM Web site. The Keyman/VSE tool works in conjunction with CSI International's TCP/IP for VSE stack.

In this example, we use Windows XP. Once you have downloaded and installed the prerequisites, you can install Keyman/VSE. One step to remember is that you need to change the .BAT file that runs the Keyman/VSE tool. The RUN.BAT Windows XP batch file sets a classpath which must include the directory path where you installed the VSE Connector client, as well as the path specific to the VSEConnector.jar file (also in the VSE Connector installation directory).

The VSE Connector server must also be running on your VSE system. This uses a configurable port number, 2893 by default. If you are already using other VSE e-business connectors, the server will probably already be running. If not, speak to your VSE system programmer to get the VSE Connector server running.

1. We can now start the Keyman/VSE tool by running RUN.BAT from the Keyman/VSE installation directory. This displays the following window:

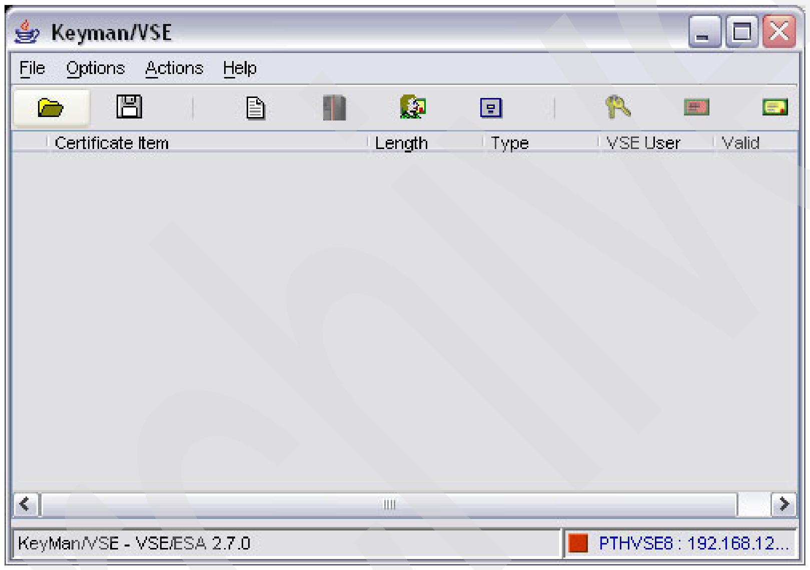
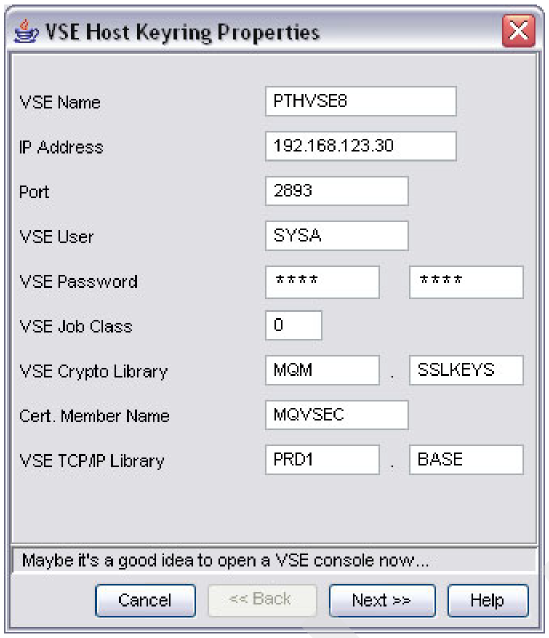


Figure 7-2 Keyman/VSE main window

2. Click **Actions** → **Create self-signed key ring** to display the VSE Host Keyring Properties window.

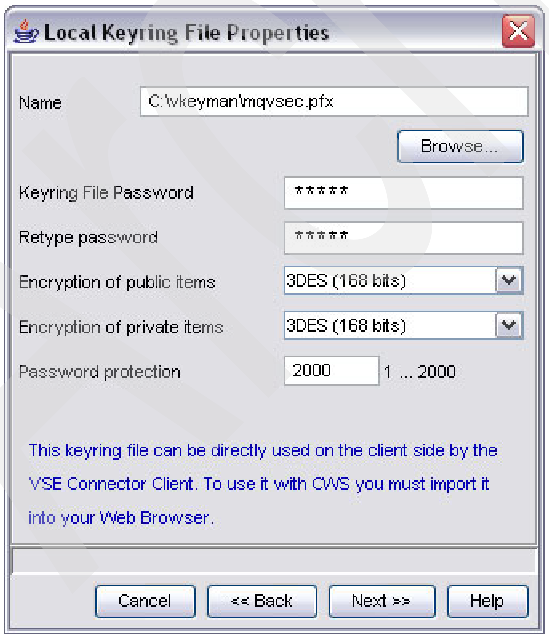

VSE Host Keyring Properties

VSE Name	PTHVSE8	
IP Address	192.168.123.30	
Port	2893	
VSE User	SYSA	
VSE Password	****	****
VSE Job Class	0	
VSE Crypto Library	MQM	. SSLKEYS
Cert. Member Name	MQVSEC	
VSE TCP/IP Library	PRD1	. BASE

Maybe it's a good idea to open a VSE console now...

Figure 7-3 Keyman/VSE Host Keyring Properties window

3. The details requested in this window are fairly straightforward. You need to identify your VSE host by providing its name and IP address and the port number of the VSE Connector server. You need to specify a user ID and password to connect to VSE, a class for running jobs in a batch partition, the SSL key-ring sublibrary name, the name of the certificate you want to create, and the installation sublibrary for TCP/IP. After you have provided these, click **Next >>** to advance to the next window.


Local Keyring File Properties

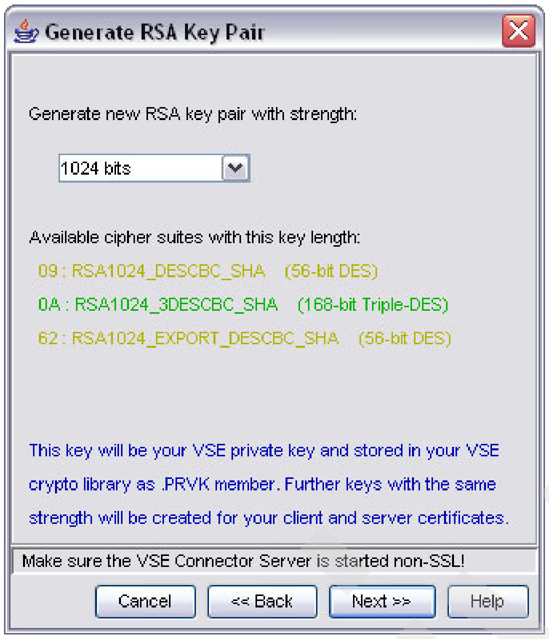
Name	C:\vkeyman\mqvsec.pfx
Keyring File Password	*****
Retype password	*****
Encryption of public items	3DES (168 bits)
Encryption of private items	3DES (168 bits)
Password protection	2000 1 ... 2000

This keyring file can be directly used on the client side by the VSE Connector Client. To use it with CWS you must import it into your Web Browser.

Figure 7-4 Keyman/VSE Local Keyring File Properties window

4. In this window, specify the name of a .PFX file that will be created by Keyman/VSE. Later, the .PFX file will be loaded into WebSphere® MQ. This file will be password protected by

Keyman/VSE, so provide a password. The other fields can retain their defaults. Once again, click **Next >>** to proceed.



Generate new RSA key pair with strength:

1024 bits

Available cipher suites with this key length:

09 : RSA1024_DESCBC_SHA (56-bit DES)

0A : RSA1024_3DESCBC_SHA (168-bit Triple-DES)

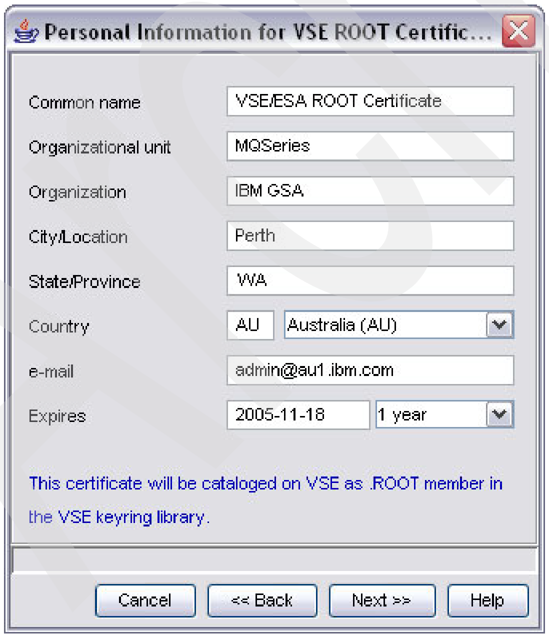
62 : RSA1024_EXPORT_DESCBC_SHA (56-bit DES)

This key will be your VSE private key and stored in your VSE crypto library as .PRVK member. Further keys with the same strength will be created for your client and server certificates.

Make sure the VSE Connector Server is started non-SSL!

Figure 7-5 Keyman/VSE Generate RSA Key Pair window

5. Notice in this window that you can select 512-bit or 1024-bit key generation. Because we plan to use the Triple DES cipher specification, choose to generate a 1024-bit key pair. You can then click **Next >>** to proceed.

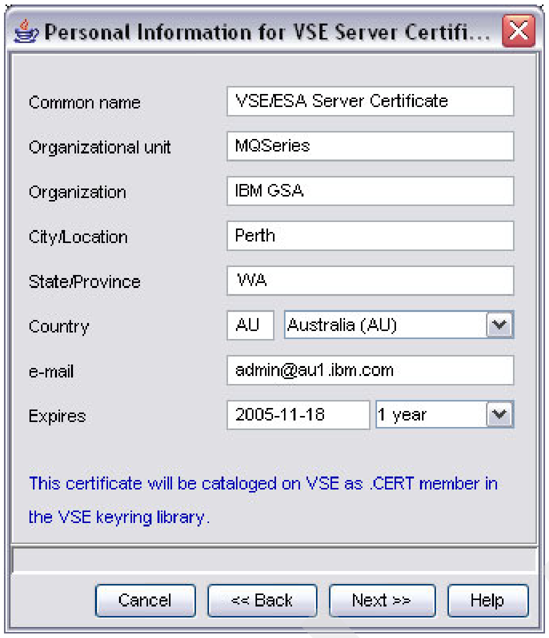


Common name	VSE/ESA ROOT Certificate
Organizational unit	MQSeries
Organization	IBM GSA
City/Location	Perth
State/Province	WA
Country	AU Australia (AU)
e-mail	admin@au1.ibm.com
Expires	2005-11-18 1 year

This certificate will be cataloged on VSE as .ROOT member in the VSE keyring library.

Figure 7-6 Keyman/VSE VSE Root Certificate Information window

6. The ROOT certificate will become the equivalent of a CA certificate. It will be used to sign our VSE certificate. The ROOT certificate will later be loaded into WebSphere MQ so that it can be used to validate the VSE certificate when our SSL-enabled channel is started. Supply the name and other details, then click **Next >>** to proceed.

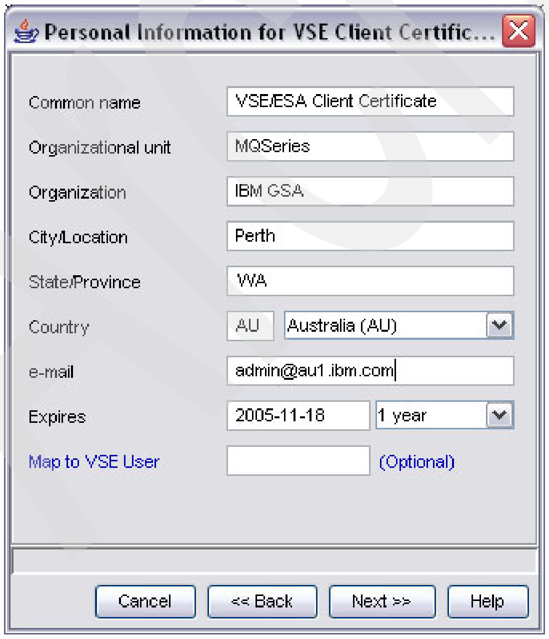


Common name	VSE/ESA Server Certificate
Organizational unit	MQSeries
Organization	IBM GSA
City/Location	Perth
State/Province	WA
Country	AU Australia (AU)
e-mail	admin@au1.ibm.com
Expires	2005-11-18 1 year

This certificate will be cataloged on VSE as .CERT member in the VSE keyring library.

Figure 7-7 Keyman/VSE VSE Server Certificate Information window

7. The VSE Server certificate is the certificate MQSeries for VSE will use when starting SSL-enabled channels. It will be signed by the ROOT certificate. On the Host Keyring Properties window, we chose that our server certificate would have the name MQVSEC. Ultimately, Keyman/VSE will create a member called MQVSEC.CERT and MQVSEC.PRVK in the VSE key-ring sublibrary. These two members will contain our server certificate and its private key respectively. Supply the name and other details, then click **Next >>** to proceed.

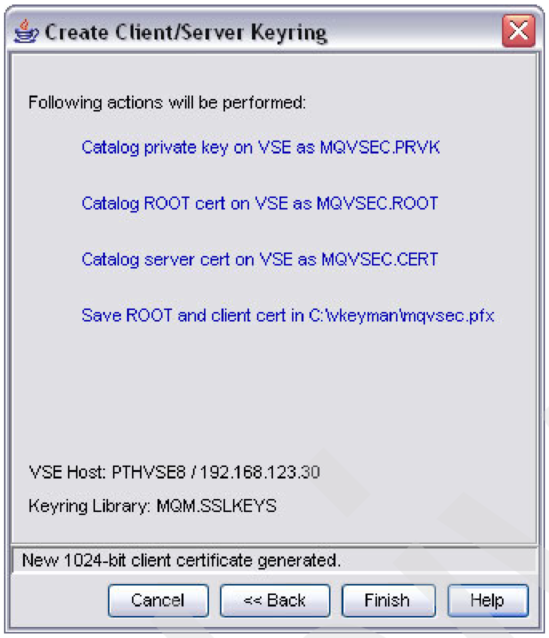


Common name	VSE/ESA Client Certificate
Organizational unit	MQSeries
Organization	IBM GSA
City/Location	Perth
State/Province	WA
Country	AU Australia (AU)
e-mail	admin@au1.ibm.com
Expires	2005-11-18 1 year
Map to VSE User	(Optional)

Figure 7-8 Keyman/VSE Client Certificate Information window

8. Finally, you specify the name and details for a client certificate. The client certificate can be used by WebSphere MQ to connect to our VSE queue manager. The client certificate, like the server certificate will be signed by the ROOT certificate.

   Click **Next >>**. Keyman/VSE displays a window that lists the actions it will now perform.



Following actions will be performed:

Catalog private key on VSE as MQVSEC.PRVK

Catalog ROOT cert on VSE as MQVSEC.ROOT

Catalog server cert on VSE as MQVSEC.CERT

Save ROOT and client cert in C:\vkeyman\mqvsec.pfx

VSE Host: PTHVSE8 / 192.168.123.30

Keyring Library: MQM.SSLKEYS

New 1024-bit client certificate generated.

Figure 7-9 Keyman/VSE Create Client/Server Keyring window

9. Click **Finish** to have Keyman/VSE perform these actions. The ROOT certificate will now exist in your SSL sublibrary. You can download it using Keyman/VSE, and export it to a Windows folder by using **File** → **Download**, and then right-clicking to export. Once you have done this, click **File** → **Save** and exit from Keyman/VSE.

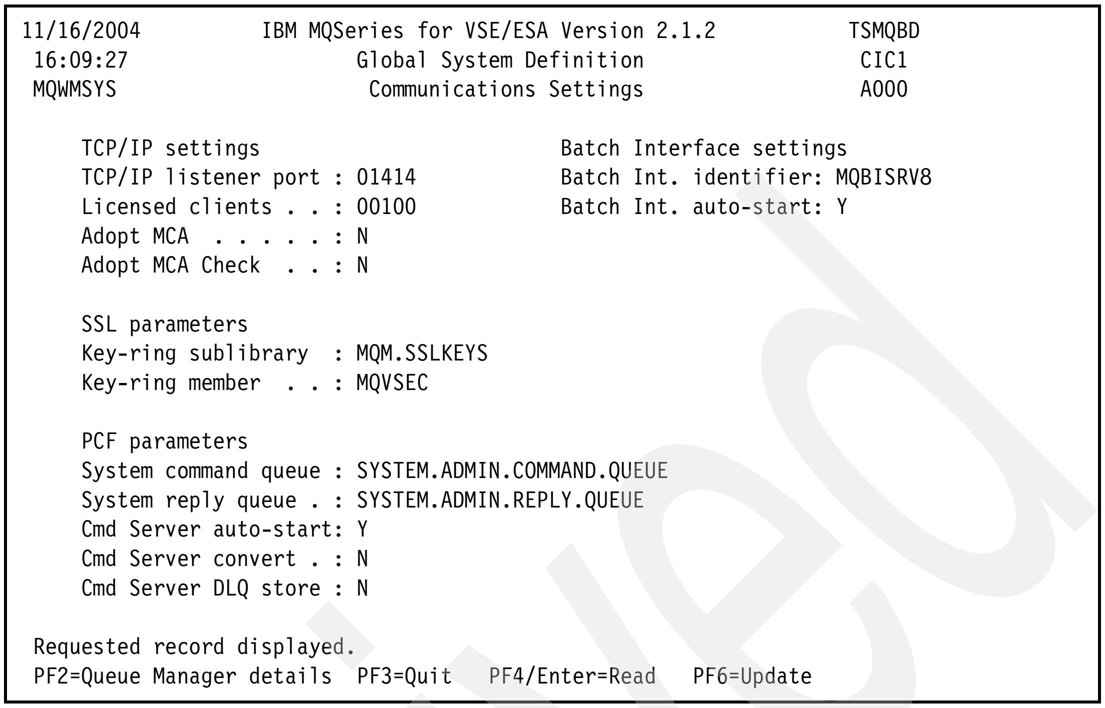
You can now configure the VSE queue manager and WebSphere MQ for an SSL-enabled channel.

7.1.3 SSL configuration

SSL must be configured on both ends of a channel. In the previous sections, we have seen two ways to create a digital certificate. The certificate will be used by MQSeries when an SSL-enabled channel is started, and the public and private keys will be used to encrypt and decrypt data sent over the channel while it is active.

Configuring MQSeries for VSE

For an SSL-enabled channel to work correctly, the queue manager must be configured for SSL. This is done by setting the SSL key-ring sublibrary and member names in the queue manager's communication settings using MQMT option 1.1 or **PF9**.



```
11/16/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
 16:09:27                  Global System Definition                CIC1
 MQWMSYS                    Communications Settings                A000

     TCP/IP settings                        Batch Interface settings
     TCP/IP listener port : 01414           Batch Int. identifier: MQBISRV8
     Licensed clients . . : 00100           Batch Int. auto-start: Y
     Adopt MCA  . . . . . : N
     Adopt MCA Check  . . : N

     SSL parameters
     Key-ring sublibrary  : MQM.SSLKEYS
     Key-ring member  . . : MQVSEC

     PCF parameters
     System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
     System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
     Cmd Server auto-start: Y
     Cmd Server convert . : N
     Cmd Server DLQ store : N

 Requested record displayed.
 PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

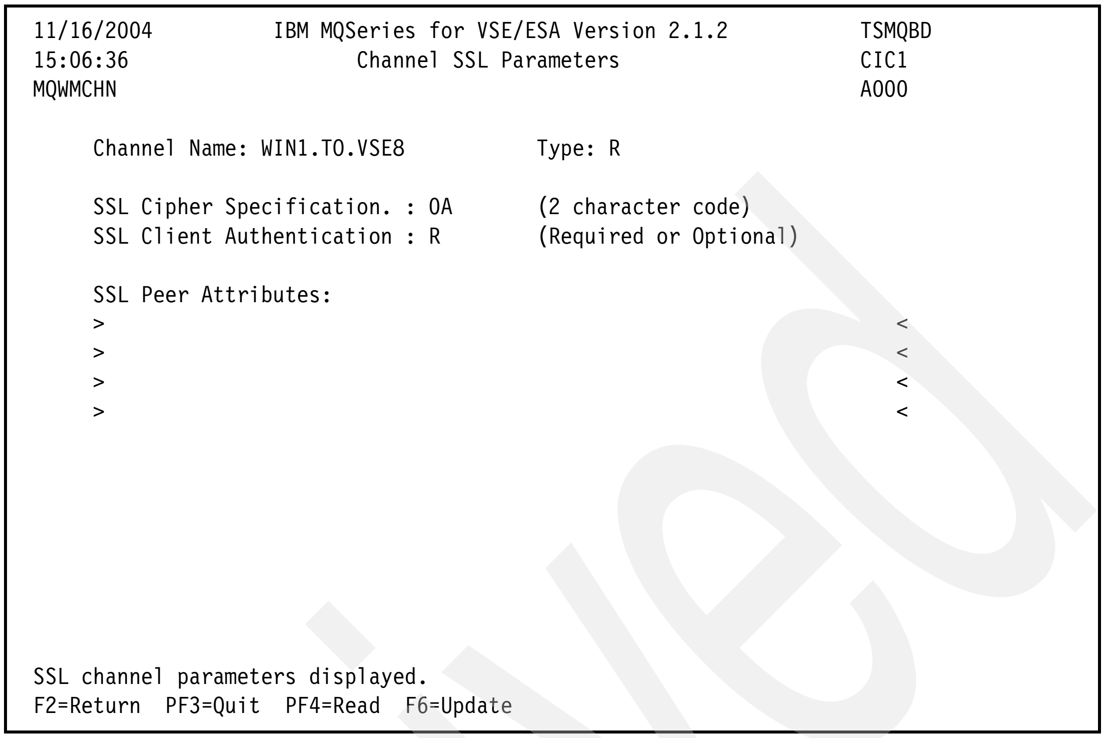
Figure 7-10 Queue manager SSL parameters

In our example, the queue manager key-ring sublibrary is MQM.SSLKEYS, and the key-ring member is MQVSEC. Remember that MQVSEC is the name of the certificate we created using Keyman/VSE.

The key-ring member name you specify in the queue manager's SSL parameters names the certificate in the key-ring sublibrary that MQSeries for VSE will use for all SSL-enabled channels. If you change these parameters, you must stop and restart MQSeries because MQSeries uses this information at startup to initialize the SSL environment.

A channel on MQSeries for VSE is considered SSL-enabled if its definition includes an SSL cipher specification. If you don't want a channel to be SSL-enabled, leave the channel SSL parameters blank.

You can configure a channel's SSL parameters in the channel definition screen using MQMT option 1.3 or **PF10**.



```
11/16/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
15:06:36                 Channel SSL Parameters                    CIC1
MQWMCHN                                                            A000

     Channel Name: WIN1.TO.VSE8           Type: R

     SSL Cipher Specification. : OA       (2 character code)
     SSL Client Authentication : R        (Required or Optional)

     SSL Peer Attributes:
     >                                                                  <
     >                                                                  <
     >                                                                  <
     >                                                                  <








SSL channel parameters displayed.
F2=Return  PF3=Quit  PF4=Read  F6=Update
```

Figure 7-11 Channel SSL parameters example

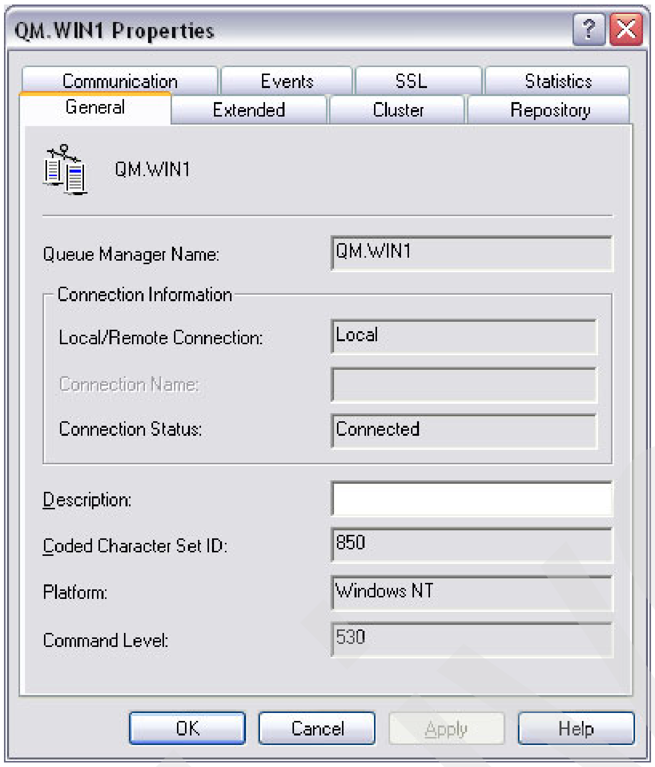
In this example, we have enabled a receiver channel called WIN1.TO.VSE8, and have selected a cipher specification of `0A` which, for SSL for VSE, identifies the RSA1024_3DESCBC_SHA, or Triple DES, cipher.

MQSeries for VSE is now configured for SSL, and the WIN1.TO.WIN8 channel is SSL-enabled. Next, for our example, we need to configure WebSphere MQ.

Configuring WebSphere MQ

The principles for configuring SSL for MQSeries for VSE are the same for WebSphere MQ on Windows XP. You need to configure the queue manager and those channels that you want to be SSL-enabled.

1. You can configure the queue manager using WebSphere MQ Explorer. Right-click the queue manager icon and select **Properties** to display the following window.


QM.WIN1 Properties

Queue Manager Name: QM.WIN1

Connection Information

Local/Remote Connection: Local

Connection Name:

Connection Status: Connected

Description:

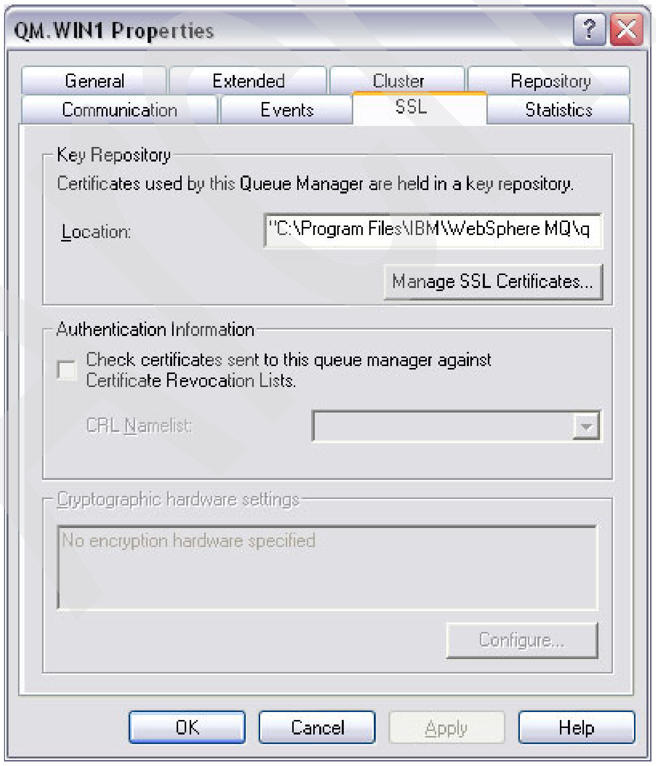
Coded Character Set ID: 850

Platform: Windows NT

Command Level: 530

Figure 7-12 WebSphere MQ Explorer queue manager properties

2. Click the **SSL** tab to display the queue manager SSL properties.


QM.WIN1 Properties

Key Repository

Certificates used by this Queue Manager are held in a key repository.

Location: "C:\Program Files\IBM\WebSphere MQ\q

Authentication Information

Check certificates sent to this queue manager against Certificate Revocation Lists.

CRL Namelist:

Cryptographic hardware settings

No encryption hardware specified

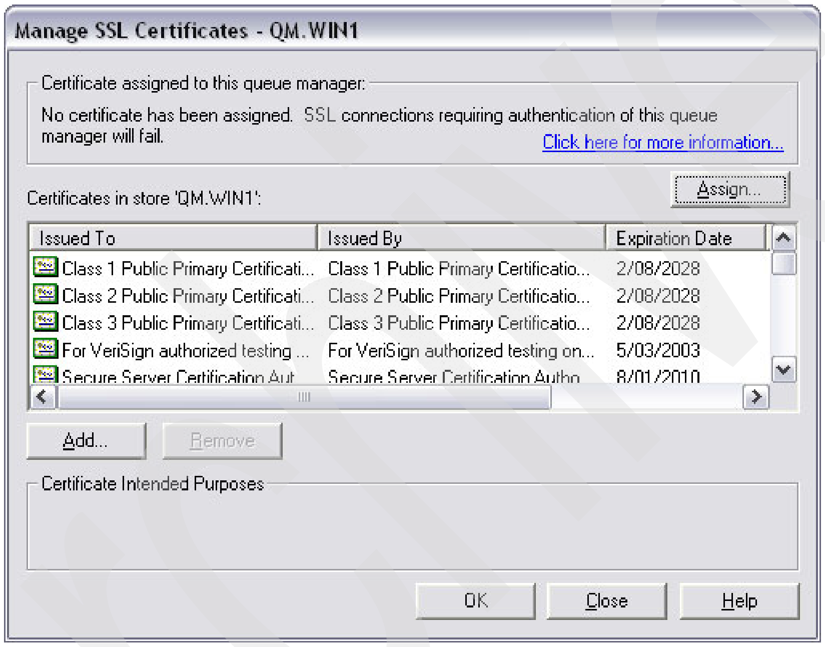
Figure 7-13 WebSphere MQ Explorer SSL properties

WebSphere MQ has a default key repository which contains certificates that can be used by SSL processes. The repository, by default, already contains certificates for some recognized Certificate Authorities.

We need to add a "client" certificate for use by SSL-enabled channels. We also need to add the ROOT certificate we created with Keyman/VSE because it is the CA for both the client and server certificates.

The term "client" pertains to WebSphere MQ in this example because it acts as the sender. The term "server" pertains to MQSeries for VSE in this example because it acts as the receiver. You do not need additional certificates if your channel operates in the other direction. The important thing is that both queue managers have their own certificate, and for WebSphere MQ, it also has the ROOT certificate because it is the CA for both certificates.

3. Click the **Manage SSL Certificates** button to display the WebSphere MQ Explorer Manage SSL Certificates window.


Manage SSL Certificates - QM.WIN1
Certificate assigned to this queue manager:
Click here for more information...
Assign...
Add...
Remove
Certificate Intended Purposes
OK
Close
Help

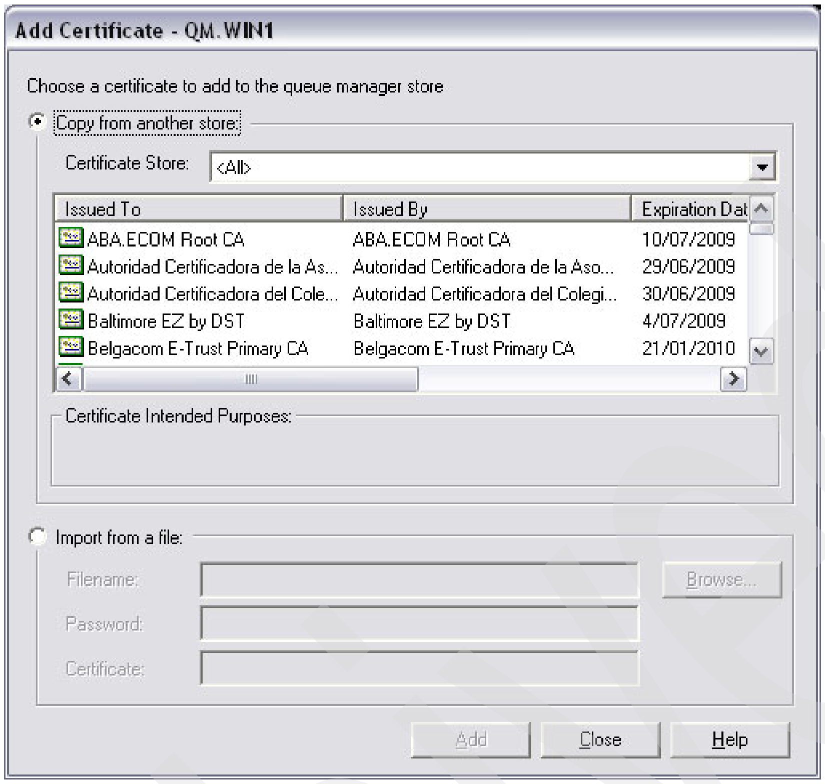
No certificate has been assigned. SSL connections requiring authentication of this queue manager will fail.

Certificates in store 'QM.WIN1':

Issued To	Issued By	Expiration Date
Class 1 Public Primary Certificati...	Class 1 Public Primary Certificatio...	2/08/2028
Class 2 Public Primary Certificati...	Class 2 Public Primary Certificatio...	2/08/2028
Class 3 Public Primary Certificati...	Class 3 Public Primary Certificatio...	2/08/2028
For VeriSign authorized testing ...	For VeriSign authorized testing on...	5/03/2003
Secure Server Certification Aut	Secure Server Certification Autho	8/01/2010

Figure 7-14 WebSphere MQ Explorer Manage SSL Certificates window

4. From this window, click **Add** to display the Add Certificate dialog.



Add Certificate - QM.WIN1

Choose a certificate to add to the queue manager store

(● Copy from another store:)

Certificate Store: <All>

Issued To	Issued By	Expiration Dat
ABA.ECOM Root CA	ABA.ECOM Root CA	10/07/2009
Autoridad Certificadora de la As...	Autoridad Certificadora de la Aso...	29/06/2009
Autoridad Certificadora del Cole...	Autoridad Certificadora del Colegi...	30/06/2009
Baltimore EZ by DST	Baltimore EZ by DST	4/07/2009
Belgacom E-Trust Primary CA	Belgacom E-Trust Primary CA	21/01/2010

Certificate Intended Purposes:

○ Import from a file:

Filename:

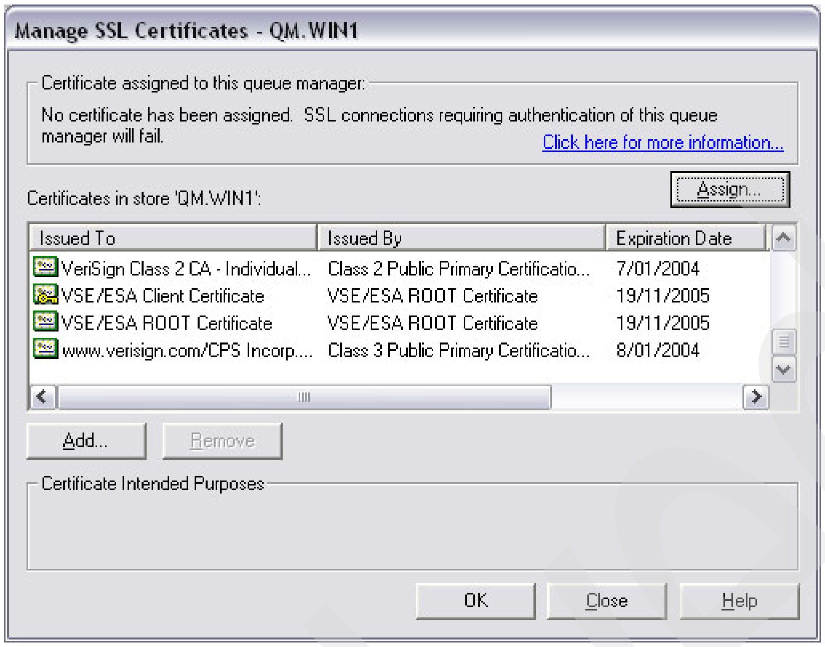
Password:

Certificate:

Figure 7-15 WebSphere MQ Explorer Add Certificate window

5. From this window, click **Import from a file** to browse your Keyman/VSE installation directory. It now contains a file called MQVSEC.PFX. Select this file to return to the Add Certificate window. You must enter the password you chose for the certificate when it was created by Keyman/VSE.
6. In the Add Certificate window, click **Add**. This adds the client certificate to the WebSphere MQ key repository. This also returns you to the Manage SSL Certificates window, where your new client certificate is listed in the “Certificates in store” list box.
7. Next, we need to add the ROOT certificate. To do so, repeat the steps you just followed to add the client certificate. However, after you click **Import from a file** and browse the Keyman/VSE installation directory, you need to change the “Files of type” browse option to “`*.CER`". This displays all the certificate files in the Keyman/VSE directory including our root certificate. Select the root certificate and click **Add**.

   If you scroll through the “Certificates in store” list box, you can now see the client and root certificates in the repository.


Manage SSL Certificates - QM.WIN1

Certificate assigned to this queue manager:

No certificate has been assigned. SSL connections requiring authentication of this queue manager will fail.

Click here for more information...

Certificates in store 'QM.WIN1':

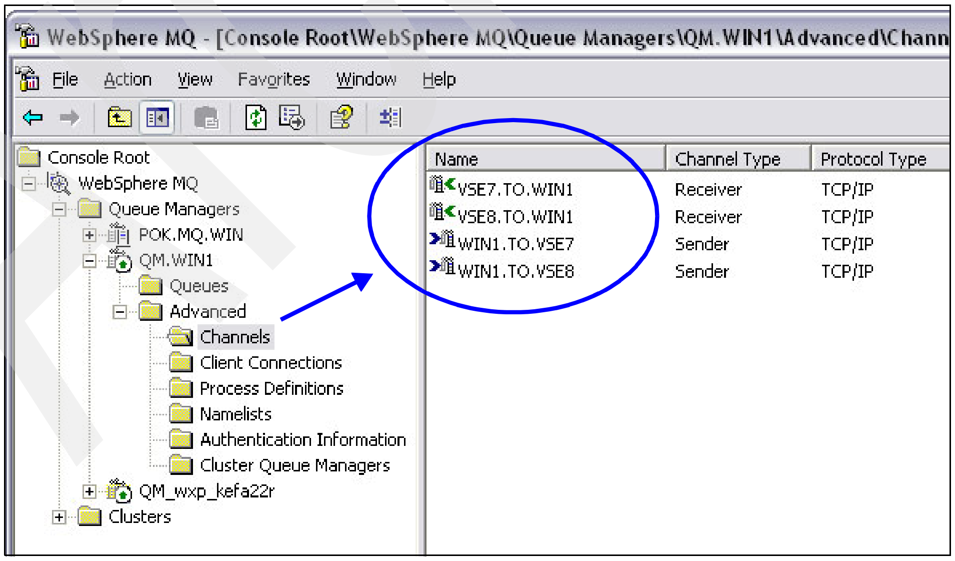
Issued To	Issued By	Expiration Date
VeriSign Class 2 CA - Individual...	Class 2 Public Primary Certificatio...	7/01/2004
VSE/ESA Client Certificate	VSE/ESA ROOT Certificate	19/11/2005
VSE/ESA ROOT Certificate	VSE/ESA ROOT Certificate	19/11/2005
www.verisign.com/CPS Incorp....	Class 3 Public Primary Certificatio...	8/01/2004

Figure 7-16 WebSphere MQ Explorer certificate repository

8. Finally, to configure the queue manager, click **Assign** to choose the certificate our WebSphere MQ channels will use when SSL-enabled. For our purposes, we assign the client certificate.

9. Next, we need to SSL-enable a channel. On MQSeries for VSE, we have already configured channel WIN1.TO.VSE8 receiver channel to be SSL-enabled. So we need to configure the matching sender channel on WebSphere MQ.

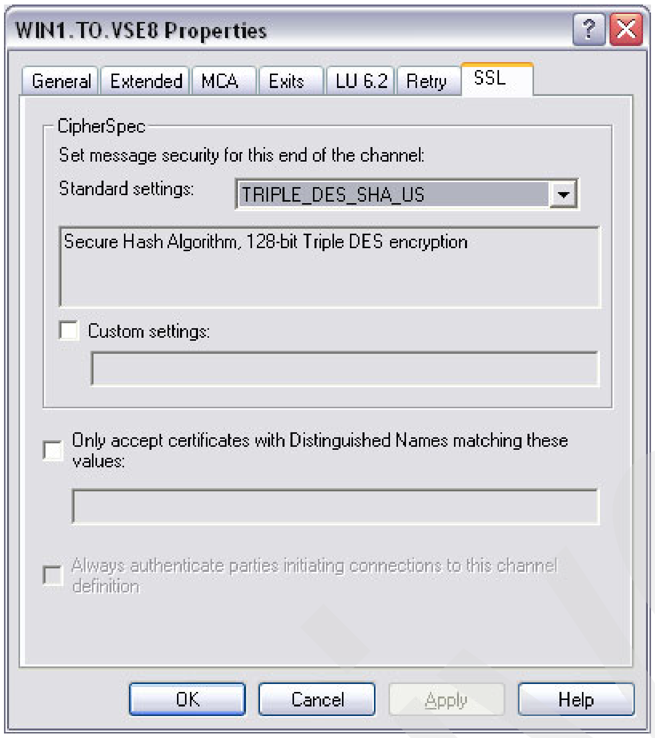
   From the WebSphere MQ Explorer main console window, expand the queue manager and its advanced objects, and select Channels. This displays a list of the queue manager's defined channels.


WebSphere MQ - [Console Root\WebSphere MQ\Queue Managers\QM.WIN1\Advanced\Chann

Name	Channel Type	Protocol Type
VSE7.TO.WIN1	Receiver	TCP/IP
VSE8.TO.WIN1	Receiver	TCP/IP
WIN1.TO.VSE7	Sender	TCP/IP
WIN1.TO.VSE8	Sender	TCP/IP

Figure 7-17 WebSphere MQ Explorer channels window

10. In this window, right-click the **WIN1.TO.VSE8** channel and click **Properties**. In the channel's properties window, click the **SSL** tab.


WIN1.TO.VSE8 Properties

CipherSpec

Set message security for this end of the channel:

Standard settings: TRIPLE_DES_SHA_US

Secure Hash Algorithm, 128-bit Triple DES encryption

☐ Custom settings:

☐ Only accept certificates with Distinguished Names matching these values:

☐ Always authenticate parties initiating connections to this channel definition

Figure 7-18 Windows MQ Explorer channel SSL parameters

Remember that the cipher specifications at both ends of the channel must match. On MQSeries for VSE we chose the Triple DES cipher specification, so we must do the same on WebSphere MQ. When you click **OK**, the channel is SSL-enabled and can be used for secure transmission.

7.2 Channel exits

Channel exits are customer-written programs that are run by MQSeries at predefined times during channel operation. Currently, MQSeries for VSE supports four types of channel exit:

- Send exits
- Receive exits
- Security exits
- Message exits

Some other MQ platforms also support message retry, auto-definition, and transport retry exits. At the time this book was written, these exits were not supported by MQSeries for VSE.

MQSeries for VSE is shipped with a sample channel exit program called MQPCHNX.Z. This is a sample COBOL program that can be used as a skeleton for any of the supported exits.

In the following sections, we explain some of the key elements of this sample.

7.2.1 Channel exit data structures

All channel exits are passed the same parameters on invocation. Because channel exit programs are started by MQSeries using CICS LINK, the parameters are passed via the COMMAREA. The parameters are:

```
01  DFHCOMMAREA.
    10  EXIT-PARMS.
        15  CHANNELEXITPARMS   USAGE POINTER.
        15  CHANNELDEFINITION  USAGE POINTER.
        15  DATALENGTH         USAGE POINTER.
        15  AGENTBUFFERLENGTH  USAGE POINTER.
        15  AGENTBUFFER        USAGE POINTER.
        15  EXITBUFFERLENGTH   USAGE POINTER.
        15  EXITBUFFERADDR     USAGE POINTER.
```

The first parameter is a pointer to the MQCXP data structure. For COBOL, this structure is defined in the CMQCXPL and CMQCXPV copybooks, and includes the following fields:

```
10  MQCXP.
    15 MQCXP-STRUCID           PIC X(4).
    15 MQCXP-VERSION           PIC S9(9) BINARY.
    15 MQCXP-EXITID            PIC S9(9) BINARY.
    15 MQCXP-EXITREASON        PIC S9(9) BINARY.
    15 MQCXP-EXITRESPONSE      PIC S9(9) BINARY.
    15 MQCXP-EXITRESPONSE2     PIC S9(9) BINARY.
    15 MQCXP-FEEDBACK          PIC S9(9) BINARY.
    15 MQCXP-MAXSEGMENTLENGTH  PIC S9(9) BINARY.
    15 MQCXP-EXITUSERAREA      PIC X(16).
    15 MQCXP-EXITDATA          PIC X(32).
```

Of immediate importance to a channel exit program are the ExitReason, ExitResponse, and ExitResponse2 fields. The ExitReason tells the exit why it was called. For example, exits are called when a channel starts and stops with an ExitReason MQXR_INIT and MQXR_TERM respectively.

The ExitResponse and ExitResponse2 fields are set by the exit on return, to indicate to the queue manager how it should proceed with the channel. For example, an exit may set the ExitResponse to MQXCC_CLOSE_CHANNEL. Typically, a channel exit will set the ExitResponse to MQXCC_OK, and channel operation will continue normally.

The MQCXP data structure and its fields are described in the *MQSeries Intercommunication Guide*, SC33-1872.

The second parameter is a pointer to the MQCD data structure. For COBOL, this structure is defined in the CMQCDL and CMQCDV copybooks. The MQCD data structure contains information about the channel definition, and includes fields for most of the channels attributes. An exit program can use MQCD if it needs information about the channel it is associated with. The MQCD has the following fields:

```
10  MQCD.
    15 MQCD-CHANNELNAME          PIC X(20).
    15 MQCD-VERSION              PIC S9(9) BINARY.
    15 MQCD-CHANNELTYPE          PIC S9(9) BINARY.
    15 MQCD-TRANSPORTTYPE        PIC S9(9) BINARY.
    15 MQCD-DESC                 PIC X(64).
    15 MQCD-QMGRNAME             PIC X(48).
    15 MQCD-XMITQNAME            PIC X(48).
    15 MQCD-SHORTCONNECTIONNAME  PIC X(20).
    15 MQCD-MCANAME              PIC X(20).
    15 MQCD-MODENAME             PIC X(8).
    15 MQCD-TPNAME               PIC X(64).
    15 MQCD-BATCHSIZE            PIC S9(9) COMP.
    15 MQCD-DISCINTERVAL         PIC S9(9) COMP.
    15 MQCD-SHORTRETRYCOUNT      PIC S9(9) COMP.
    15 MQCD-SHORTRETRYINTERVAL   PIC S9(9) COMP.
    15 MQCD-LONGRETRYCOUNT       PIC S9(9) COMP.
    15 MQCD-LONGRETRYINTERVAL    PIC S9(9) COMP.
    15 MQCD-SECURITYEXIT         PIC X(8).
    15 MQCD-MSGEXIT              PIC X(8).
    15 MQCD-SENDEXIT             PIC X(8).
    15 MQCD-RECEIVEEXIT          PIC X(8).
    15 MQCD-SEQNUMBERWRAP        PIC S9(9) COMP.
    15 MQCD-MAXMSGLENGTH         PIC S9(9) COMP.
    15 MQCD-PUTAUTHORITY         PIC S9(9) COMP.
    15 MQCD-DATACONVERSION       PIC S9(9) COMP.
    15 MQCD-SECURITYUSERDATA     PIC X(32).
    15 MQCD-MSGUSERDATA          PIC X(32).
    15 MQCD-SENDUSERDATA         PIC X(32).
    15 MQCD-RECEIVEUSERDATA      PIC X(32).
    15 MQCD-USERIDENTIFIER       PIC X(12).
    15 MQCD-PASSWORD             PIC X(12).
    15 MQCD-MCAUSERIDENTIFIER    PIC X(12).
    15 MQCD-MCATYPE              PIC S9(9) BINARY.
    15 MQCD-CONNECTIONNAME       PIC X(264).
    15 MQCD-REMOTEUSERIDENTIFIER PIC X(12).
    15 MQCD-REMOTEPASSWORD       PIC X(12).
```

Once again, the MQCD data structure and its fields are described in the *MQSeries Intercommunication Guide*, SC33-1872.

The third parameter, DataLength, is a pointer to an input/output parameter. It contains the length of any data passed to the exit in the AgentBuffer, and should be changed by the exit if it changes the length of the data in the AgentBuffer on return. The exit program can use this length to know exactly how many bytes of data it needs to process.

The AgentBuffer and AgentBufferLength parameters are pointers to any data being passed to the exit and its length. For example, for a send exit, AgentBuffer typically contains data that is about to be sent to a remote queue manager in a single transmission. The AgentBufferLength points to the length of this data.

Finally, the ExitBufferAddr and ExitBufferLength parameters are pointers to an alternative buffer and length. These pointers are managed by the exit program, and are can be used if the AgentBuffer is too small. It is the responsibility of the exit program to allocate and free storage associate with the buffer, and set the ExitBufferAddr appropriately. The exit program can set the ExitResponse2 field to MQXR2_USE_EXIT_BUFFER to instruct the queue manager to use the ExitBufferAddr rather than the AgentBuffer.

7.2.2 Channel exit programming

The MQPCHNX.Z sample program is a generic sample that can be used for any channel exit type. Consequently, the main processing paragraph of the sample uses an EVALUATE statement to determine whether the exit has been started as a send, receive, security, or message exit.

```
EVALUATE TRUE
    WHEN MQCXP-EXITID = MQXT-CHANNEL-SEND-EXIT
        PERFORM 2100-SEND-EXIT
    WHEN MQCXP-EXITID = MQXT-CHANNEL-RCV-EXIT
        PERFORM 2200-RECV-EXIT
    WHEN MQCXP-EXITID = MQXT-CHANNEL-SEC-EXIT
        PERFORM 2300-SECR-EXIT
    WHEN MQCXP-EXITID = MQXT-CHANNEL-MSG-EXIT
        PERFORM 2400-MESS-EXIT
    WHEN OTHER
        MOVE MQXCC-CLOSE-CHANNEL TO MQCXP-EXITRESPONSE
END-EVALUATE.
```

If you are writing an exit for only one purpose, for example a send exit, you can skip this logic and go straight to logic relevant to the exit type. You can do this because you know that your exit will only ever be configured in the channel's exit parameters as a send exit, and so it will only ever be invoked as a send exit.

The first thing your exit needs to do is to examine the ExitReason field in the MQCXP to determine why the exit was called. So, to proceed with the send exit example, we perform the following logic:

```
EVALUATE TRUE
    WHEN MQCXP-EXITREASON = MQXR-INIT
        PERFORM 2100-SEND-INIT
    WHEN MQCXP-EXITREASON = MQXR-TERM
        PERFORM 2100-SEND-TERM
    WHEN MQCXP-EXITREASON = MQXR-XMIT
        PERFORM 2100-SEND-XMIT
    WHEN OTHER
        MOVE MQXCC-CLOSE-CHANNEL TO MQCXP-EXITRESPONSE
END-EVALUATE.
```

As already mentioned, exits are always called at channel initialization and termination. So, whichever exit type we are programming, we can always expect the exit to be called with the ExitReason set to MQXR-INIT and MQXR-TERM. In the case of initialization, you can include initialization logic such as storage allocation, and then during termination, free the storage.

For a send exit, the exit is called with the ExitReason set to MQXR-XMIT immediately before the queue manager begins sending data over the channel. It is at this point that you can change the data before it is sent. For a receive exit, the exit is called with the ExitReason set to MQXR-XMIT immediately after receiving data over the channel. It is at this point that you can restore the data back to what it was before it was changed by the send exit. Clearly, send and receive exits work in pairs.

Message exits are slightly different. Apart from being called with MQXR-INIT and MQXR-TERM, a message exit is also called with the ExitReason set to MQXR-MSG. Message exits are called on the sender side when a message is ready to be sent, but before it is transmitted. Message exits are called on the receiver side when the message has been transmitted, but before it is placed on a target queue. The message exit at either end of the channel can change message data, including the MQMD of the message.

Security exits are different again. As well as being called at channel initialization and termination, a security exit can be called with the ExitReason set to MQXR-INIT-SEC and MQXR-SEC-MSG. Therefore, logic for security exits is slightly more complex.

```
EVALUATE TRUE
    WHEN MQCXP-EXITREASON = MQXR-INIT
        PERFORM 2300-SECR-INIT
    WHEN MQCXP-EXITREASON = MQXR-TERM
        PERFORM 2300-SECR-TERM
    WHEN MQCXP-EXITREASON = MQXR-INIT-SEC
        PERFORM 2300-SECR-INITSEC
    WHEN MQCXP-EXITREASON = MQXR-SEC-MSG
        PERFORM 2300-SECR-SECMSG
    WHEN OTHER
        MOVE MQXCC-CLOSE-CHANNEL TO MQCXP-EXITRESPONSE
END-EVALUATE.
```

A security exit is called with MQXR-INIT-SEC if it is the *initiating* exit. If a receiver channel has an exit, it is always the initiating exit. If the receiver channel does not have a security exit and the sender channel does, the sender's exit is the initiating exit.

Regardless of which exit is called with MQXR-INIT-SEC, all subsequent invocations (except for MQXR-TERM) are with the ExitReason set to MQXR-SEC-MSG.

As with all exits, the exit program should check the DataLength to see if there is any data to process. Processing logic will differ for each exit type, but more particularly, logic will differ depending on your particular need for the exit. For example, you might include encryption logic in a send exit and decryption logic in a receive exit.

As an example, in the MQPCHNX.Z sample, when processing the invocation for ExitReason MQXR-INIT-SEC, the logic places a user ID and password in the agent buffer and sets the ExitResponse to MQXCC-SEND-SEC-MSG.

```
MOVE 'SMITHJ' TO WS-SECR-USERID
MOVE 'S3CR3T' TO WS-SECR-PASSWD
MOVE WS-SECRDATA TO LK-AGENTBUF
MOVE LENGTH OF WS-SECRDATA TO LK-DATA-LEN
MOVE MQXCC-SEND-SEC-MSG TO MQCXP-EXITRESPONSE
```

The ExitResponse tells the queue manager, in this case, to send the contents of the AgentBuffer over the channel. The remote partner will recognize the security data flow and invoke its own security exit (if one is defined for the channel), and pass it the AgentBuffer. The remote security exit can then process the AgentBuffer and either send more security data back, or terminate the channel. Data can flow between the two security exits until one closes the channel, or the initiating exit tells the queue manager to proceed with normal channel operation by setting the ExitResponse to MQXCC-OK.

Once you have performed your logic, it is essential that you set the ExitResponse and ExitResponse2 fields in the MQCXP correctly. These fields tell the queue manager how to proceed when the exit returns. For example, you can tell the queue manager to continue or to stop the channel. For message exits, you can tell the queue manager to suppress a message before it is sent, or before it is placed on a target queue. Valid values for these fields and a description of their meaning are described in the *MQSeries Intercommunication Guide*, SC33-1872.

When your exit is ready to return control to the queue manager, it should use CICS RETURN.

Application programming

Having installed and configured MQSeries for VSE, it is likely that you want to write application programs to exploit its services. Generally speaking, there are three types of application programs that exploit MQSeries for VSE services. These are:

- CICS application programs
- Batch application programs
- Trigger application programs

Client application programs can also use MQSeries for VSE services; however, these are written for non-VSE platforms and details for such programs should be reviewed in the relevant manuals.

The MQ CICS Bridge also uses MQSeries resources when supporting remote applications that use the bridge to emulate CICS DPL and 3270 programs.

Application programs exploit MQSeries services using the Message Queue Interface (MQI). Details on using MQI calls are found in the following manuals:

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Application Programming Guide*, SC33-0807

In the following sections, we provide an overview of application programming for MQSeries for VSE including how to build your application programs.

8.1 CICS application programs

Because MQSeries for VSE runs as a CICS subsystem, most of your applications are probably CICS programs. Before writing MQSeries CICS applications you should be familiar with CICS programming in general. A good place to begin is with the following two guides:

- *CICS Application Programming Reference*
- *CICS Application Programming Guide*

The relevant CICS manuals will be determined by which CICS system you are using.

MQSeries for VSE is shipped with a number of sample programs. Among them is TTPTST2.COBOL. This is a CICS application program that runs as transaction TST2. You can use this program as a basis for your own MQSeries applications. One thing to be aware of is that this program hardcodes declarations that are normally in copybooks. This is because TTPTST2 is listed in full in several manuals and it is difficult to read without the full expansion of the copybooks. Your programs should use the supplied copybooks.

MQI calls made from application programs involve parameter lists of data types and structures. These types and structures are declared in supplied copybooks and header files. An application program needs to copy or include the appropriate files so that the data types and structures are available to the program. Details of MQSeries for VSE copybooks and header files are discussed in the following section.

8.1.1 Copybooks and header files

MQSeries copybooks and header files for application programs are documented in the *MQSeries Application Programming Reference*, SC33-1673; however, only a subset of these are relevant to MQSeries for VSE. The following tables describe MQSeries for VSE copybooks and header files including their names, descriptions, and practical uses.

COBOL copybooks

Table 8-1 describes COBOL copybooks.

Table 8-1 COBOL copybooks for application programs

Name	Type	Description	Use
CMQV	C-COPY	Contains default values for all MQI parameters.	Used for all MQI calls for setting and checking parameter values.
CMQMDL CMQMDV	C-COPY	Contains structure definition for message descriptions.	Used for MQGET, MQPUT, and MQPUT1 calls.
CMQGMOL CMQGMOV	C-COPY	Contains structure definition for get message options.	Used for MQGET call.
CMQPMOL CMQPMOV	C-COPY	Contains structure definition for put message options.	Used for MQGPUT and MQPUT1 calls.
CMQODL CMQODV	C-COPY	Contains structure definition for object descriptor.	Used for MQOPEN and MQPUT1 calls.
CMQTML CMQTMV	C-COPY	Contains structure definition for the Trigger Message header.	Used by trigger programs.
CMQXV	C-COPY	Contains constants for channel definitions and exits.	Used by channel exit programs.

Name	Type	Description	Use
CMQCDL CMQCDV	C-COPY	Contains structure definition for the Channel Descriptor.	Used by channel exit programs.
CMQCXPL CMQCXPV	C-COPY	Contains structure definition for the Channel Exit Parameters.	Used by channel exit programs.
CMQXQHL CMQXQHV	C-COPY	Contains structure definition for the Transmission Queue Header.	Used by channel exit programs.
CMQCFHL CMQCFHV	C-COPY	Contains structure definition for the Command Format Header.	Used by PCF programs.
CMQCFINL CMQCFINV	C-COPY	Contains structure definition for the Command Format Integers.	Used by PCF programs.
CMQCFSTL CMQCFSTV	C-COPY	Contains structure definition for the Command Format Strings.	Used by PCF programs.
CMQCFILL CMQCFILV	C-COPY	Contains structure definition for the Command Format Integer Lists.	Used by PCF programs.
CMQCFSLL CMQCFSLV	C-COPY	Contains structure definition for the Command Format String Lists.	Used by PCF programs.
CMQCFV	C-COPY	Contains definitions for PCF commands, return codes and structure constants.	Used by PCF programs.
CMQCIHL CMQCIHV	C-COPY	Contains structure definition for the CICS Information header.	Used by MQ CICS Bridge programs.

Generally, in Table 8-1 on page 148, copybooks ending with the letter L contain data structure definitions, and copybooks ending with the letter V contain data structure definitions with default values.

C header files

Table 8-2 describes C header files.

Table 8-2 C language header files for application programs

Name	Type	Description	Use
CMQC	H	Contains structure definitions for all MQI parameters. Contains default values for all MQI parameters.	Used for all MQI calls for defining parameters and setting and checking parameter values.
CMQCFC	H	Contains definitions for PCF commands, return codes and structure constants.	Used by PCF programs.
CMQXC	H	Contains definitions for channel exits and data conversion exits.	Used by channel exits and data conversion programs.

PL/I copybooks

Table 8-3 describes PL/I copybooks.

Table 8-3 PL/I copybooks for application programs

Name	Type	Description	Use
CMQP	P	Contains structure definitions for all MQI parameters. Contains default values for all MQI parameters.	Used for all MQI calls for defining parameters and setting and checking parameter values.
CMQCFP	P	Contains definitions for PCF commands, return codes and structure constants.	Used by PCF programs.
CMQXP	P	Contains definitions for channel exits and data conversion exits.	Used by channel exits and data conversion programs.
CMQEPP	P	Contains entry definitions for MQI calls.	Used by MQI programs.

Assembler copybooks

For MQSeries for VSE 2.1, there are no MQI copybooks for Assembler. However, we have provided sample CICS Assembler code in “Assembler language programs” on page 268.

There is no support for Language Environment/VSE-conforming Assembler main routines under CICS. For this reason, CICS Assembler programs must be called from either a COBOL, C, or PL/I main routine. For example, you can write your entire program in Assembler and then call the Assembler entry point from a COBOL, C, or PL/I program. The following is a COBOL example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   MQASMAIN
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
PROCEDURE DIVISION.
0000-MAIN-LINE.
    CALL 'MQASMPRG'.
    EXEC CICS RETURN END-EXEC.
```

8.1.2 Building CICS application programs

Building MQSeries CICS applications is no different from building other CICS programs except that you must ensure your LIBDEF SEARCH includes the PRD2.MQSERIES library. This is necessary for the compile step to find the MQI copybooks or header files. It is also necessary for the link-edit step to find the MQI linkable objects.

Sample JCL to build COBOL, C, PL/I, and Assembler programs is found in Appendix C, “Sample JCL” on page 297. It should be noted that CICS C language programs will probably be reentrant. The build process for such programs includes a pre-link step.

CICS application programs have an additional requirement for CSD entries. Once again, these are no different than other CICS applications.

8.2 Batch application programs

MQ batch programs use MQI calls exactly the same way as MQ CICS programs. For this reason, you can use the same copybooks and header files described in 8.1.1, “Copybooks and header files” on page 148.

Sample JCL to build an MQ COBOL batch program is found in “JCL to build a batch COBOL program” on page 299. The important thing to remember is that MQ batch programs must be linked with the object file MQBIBTCH.

Having built your MQ batch program, you can only run it while the MQSeries for VSE batch interface is active in CICS.

8.2.1 Batch interface overview

Unlike other platforms, MQSeries for VSE is implemented as a CICS subsystem. This means access to MQ objects using the Message Queue Interface (MQI) is restricted to CICS applications. To get around this limitation, MQSeries for VSE provides an interface for batch programs.

The batch interface is designed to standardize the programming style of CICS and batch programs. From a programming point of view, batch programs use MQI calls exactly the same way as CICS programs. In other words, MQ batch programs issue the calls MQCONN, MQOPEN, MQPUT, and so forth to access MQ objects.

Since MQ objects are ultimately under the control of the CICS subsystem, MQI calls issued by batch programs are passed to the CICS partition for processing. This is achieved using Cross Partition Communication Calls (XPCC). Batch programs are not concerned with XPCC since all relevant logic is built into the MQI calls.

MQSeries for VSE provides a special CICS transaction (MQBI) that must be running to process MQI calls issued by batch programs. This transaction must be running for the batch interface to be available. MQBI waits for MQCONN calls issued by batch programs. When these are received, MQBI starts a second transaction, MQBX, that issues all MQI calls on behalf of the batch program. There is one MQBX instance for each active batch connection.

The MQBX transaction runs for the duration of the logical MQ connection. In other words, it runs until the batch program issues an MQDISC. If a batch program issues a second MQCONN call, the batch interface starts a second MQBX transaction for the duration of the MQ connection. This design allows batch programs to create logical units of work. It also means multiple batch programs (including multiple VSE subtasks) can establish concurrent connections to the MQ queue manager.

There is a slight programming difference when issuing the MQCMIT and MQBACK MQI calls from a batch program. In a CICS application, the program must issue CICS SYNCPOINT and SYNCPOINT ROLLBACK calls in addition to the corresponding MQI calls. This is because the MQI calls themselves do not issue CICS SYNCPOINT commands. For batch programs, this is not the case. The MQCMIT and MQBACK MQI calls, when issued from a batch program, do result in an appropriate CICS SYNCPOINT command being issued. This is because batch programs have no other way to issue such commands. In addition, the MQDISC MQI call issues a CICS SYNCPOINT by default.

It should be noted that since MQI calls issued from batch programs are transferred to mirror CICS transactions, there is a performance overhead in using the batch interface.

Another difference between MQI programs in CICS and batch is that CICS applications can get and put messages with lengths up to 4 megabytes. Using the batch interface, programs are restricted to maximum message lengths of 250 kilobytes.

The diagram shown in Figure 8-1 illustrates the batch interface and the interrelation of MQSeries objects.

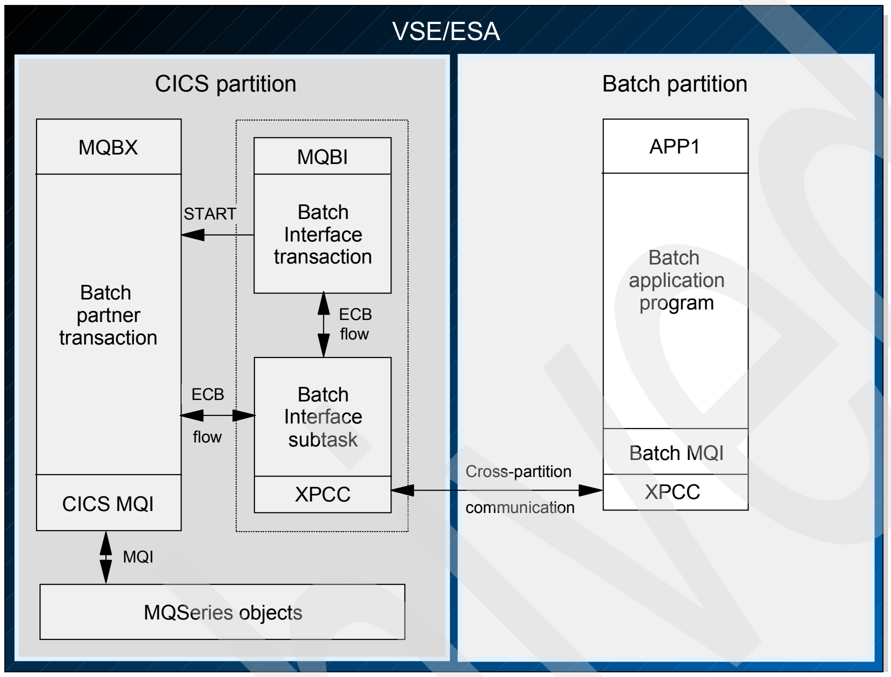


Figure 8-1 Batch interface and interrelated MQSeries objects

8.2.2 Starting the batch interface

The batch interface is started by running the MQBI transaction. This can be done manually in native CICS or automatically via the queue manager's communication settings. Alternatively, the interface can be started by configuring CICS to run the batch interface start program during post initialization. MQBI is a long-running CICS transaction that coordinates multiple, simultaneous batch connections to the MQ queue manager.

Starting in native CICS

To start the batch interface in native CICS, enter:

```
MQBI
```

If successful, the following is displayed on the terminal:

```
MQSeries Batch Interface start request issued.
```

The following is displayed on the console:

```
MQI0100I - MQS Batch Interface (mqbiname) started
```

Note that the `mqbiname` in the above message is the batch interface identifier specified in your queue manager's communication settings. This name identifies the batch interface service to batch applications, which specify which queue manager they want to connect to using a // SETPARM card, For example,

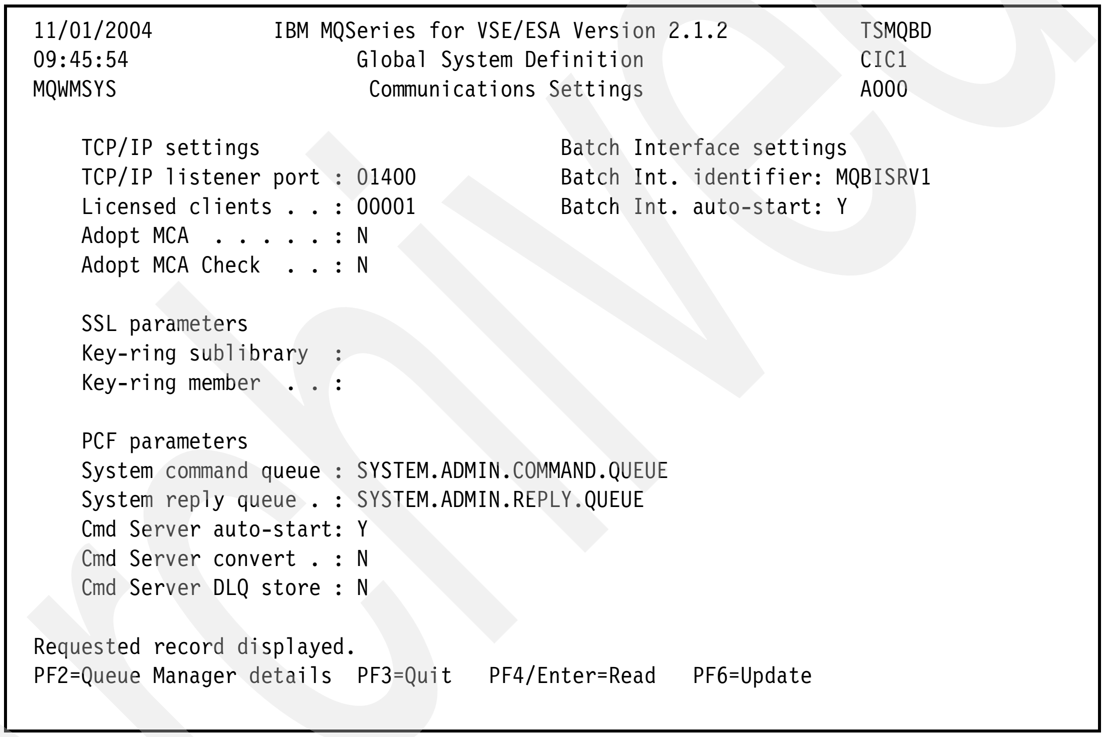
```
// SETPARM MQBISRV=MQBISRV1
```

Each queue manager running on your VSE system can have its own interface. Each interface must have its own unique batch interface identifier. In this example, we are using MQBISRV1 as the interface name. If you had two queue managers running in two different CICS regions, batch programs could connect to either by setting the // SETPARM card appropriately.

If your batch job does not include a // SETPARM card, the batch program will attempt to connect to the default batch identifier name, MQBISERV. The batch interface uses Language Environment services to process the // SETPARM card, so you must include the Language Environment installation library in your job's LIBDEF SEARCH.

Starting automatically

The batch interface can be started automatically when MQSeries is started. This can be configured via the queue manager's communication settings, accessible using MQMT option 1.1, and **PF9**.



```
11/01/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
09:45:54                  Global System Definition                 CIC1
MQWMSYS                    Communications Settings                 A000

    TCP/IP settings                         Batch Interface settings
    TCP/IP listener port : 01400            Batch Int. identifier: MQBISRV1
    Licensed clients . . : 00001            Batch Int. auto-start: Y
    Adopt MCA  . . . . . : N
    Adopt MCA Check  . . : N

    SSL parameters
    Key-ring sublibrary  :
    Key-ring member  . . :

    PCF parameters
    System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
    System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
    Cmd Server auto-start: Y
    Cmd Server convert . : N
    Cmd Server DLQ store : N

Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure 8-2 Batch interface auto-start configuration

The Batch Interface settings indicate whether the batch interface should be started automatically when the queue manager is started. Automatic start occurs if the auto-start parameter is set to Y.

Regardless of how the interface is started, the batch interface identifier parameter specifies the name of the service. The name of the service is important to batch applications, which specify the interface name via a // SETPARM card in their JCL.

When the batch interface is started automatically, the following message is displayed on the VSE console (unless it is suppressed by your log and trace settings):

```
MQI0100I - MQS Batch Interface (mqbiname) started
```

Once again, the “mqbiname” in the message is the name of the started batch interface.

Starting during post initialization

To start the batch interface during CICS post initialization, change your DFHPLTPI definition to include the following entry:

```
DFHPLT TYPE=ENTRY,PROGRAM=MQPSTBI
```

The PLTPI SIT parameter must identify the phase built from the preceding definition. If successful, the following is displayed on the console when CICS is started:

```
MQI0100I - MQS Batch Interface (mqbiname) started
```

Note that the PLTPI definition must follow the entries that start MQSeries for VSE. For example:

```
DFHPLT TYPE=ENTRY,PROGRAM=MQPSENV
DFHPLT TYPE=ENTRY,PROGRAM=MQPSTART
DFHPLT TYPE=ENTRY,PROGRAM=MQPSTBI
```

After successfully starting the batch interface, the MQBI transaction is running and can be verified by using the CEMT transaction. For example:

```
i task
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(00075) Tra(MQBI)           Act Tas
 Tas(00043) Tra(MQTL)           Act Tas
 Tas(00076) Tra(CEMT) Fac(SFC2) Act Ter
 Tas(00044) Tra(MQSM)           Sus Tas
```

8.2.3 Stopping the batch interface

The batch interface can be stopped normally in one of four ways: in native CICS, from a batch program, or during MQSeries or CICS shutdown. Of course, it can also be stopped abnormally by shutting down CICS with the immediate option or by purging or forcing the MQBI transaction.

Stopping in native CICS

To stop the batch interface in native CICS, enter:

```
MQBI X
```

If successful, the following is displayed on the terminal:

```
MQSeries Batch Interface stop requested.
```

And the following is displayed on the console:

```
MQI0102I - MQS Batch Interface (mqbiname) stop requested
MQI0104I - MQS Batch Interface (mqbiname) ended
```

The stop parameter is not case sensitive.

Stopping from a batch program

To stop the batch interface from a batch program, use the special MQBIEND call. For example:

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.  MQBISTOP.
       AUTHOR.      IBM.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
```

```
PROCEDURE DIVISION.
    CALL 'MQBIEND'.
    GOBACK.
```

If successful, the following is displayed on the console:

```
MQI0102I - MQS Batch Interface (mqbiname) stop requested
MQI0104I - MQS Batch Interface (mqbiname) ended
```

Stopping during MQSeries shutdown

If you specified that the batch interface should start automatically when MQSeries is started (that is, via the queue manager's communications settings), then the interface will stop automatically when MQSeries is shut down.

Stopping during CICS shutdown

To stop the batch interface during CICS shutdown, change your DFHPLTSD definition to include the following entry:

```
DFHPLT TYPE=ENTRY,PROGRAM=MQPSPBI
```

The PLTSD SIT parameter must identify the phase built from the preceding definition. If successful, when CICS is shutdown, the following is displayed on the console:

```
MQI0102I - MQS Batch Interface (mqbiname) stop requested
MQI0104I - MQS Batch Interface (mqbiname) ended
```

Note that the PLTSD definition must precede the entry that stops MQSeries for VSE. For example:

```
DFHPLT TYPE=ENTRY,PROGRAM=MQPSPBI
DFHPLT TYPE=ENTRY,PROGRAM=MQPSTOP
```

Normal versus abnormal termination

If the batch interface is stopped normally, the MQBI transaction continues to run until all active batch connections are completed. In other words, it will not forcibly terminate active MQ batch connections. Once a stop request has been registered, the batch interface does not accept any new batch connections.

If the batch interface is stopped abnormally, the interface forcibly terminates all active batch connections. Each logical unit of work is rolled back to its last syncpoint.

In the event that an MQI batch program abends, the XPCC connection is lost. This is detected by the batch interface subtask (which handles XPCC communication), which, in turn, informs the partner transaction (MQBX) to terminate. MQBX issues a SYNCPOINT ROLLBACK to the last committed work and terminates cleanly.

8.2.4 Writing MQI batch programs

MQI batch programs use the same MQI programming concepts as CICS applications. The communication between the batch partition and CICS is handled by MQI call logic and is transparent to the batch program.

MQSeries for VSE is shipped with a sample MQ batch program called MQBICALL. This is a COBOL program that accepts PUT or GET commands for a variable number of test messages to or from a specified queue. Commands are read from SYSIPT. A JCL example to run the MQBICALL is as follows:

```
// JOB MQBITEST
// SETPARM MQBISRV=MQBISRV1
// LIBDEF *,SEARCH=(USER.LIB, PRD2.SCEEBASE)
// EXEC MQBICALL
PUT 010 TEST.QUEUE
/*
/&
```

This example puts 10 test messages on the queue TEST.QUEUE, owned by the queue manager identified by interface MQBISRV1.

Although this test program only supports PUT and GET commands, the batch interface supports the following MQI calls:

```
MQCONN
MQDISC
MQOPEN
MQCLOSE
MQPUT
MQPUT1
MQGET
MQINQ
MQCMIT
MQBACK
MQBIEND
```

Details of these MQI calls (excluding MQBIEND) can be found in the *MQSeries Application Programming Reference*, SC33-1673.

8.2.5 Building MQI batch programs

MQI calls for batch programs use different logic than the MQI calls of CICS application programs. Consequently, batch programs must be linked with the MQ batch interface object file MQBIBTCH.OBJ. For example:

```
// OPTION CATAL
   PHASE MQBITEST,*
   INCLUDE MQBITEST
   INCLUDE MQBIBTCH
// EXEC LNKEDT
```

The MQBIBTCH object contains an entry point for each of the MQI calls applicable to the batch interface.

8.2.6 Running the MQI batch example

Now that we have seen an overview of the batch interface and its operation, we can go through, step-by-step, the full process of running an MQI batch program. To this end, we use the MQI batch example program MQBICALL.

We start with the assumption that MQSeries for VSE is correctly installed and running under CICS. The steps are as follows:

1. The first thing we need to do is build an executable phase. You will find the source code for the MQBICALL COBOL program in your MQSeries sublibrary (PRD2.MQSERIES). We do not need to make any changes to the source code, so we will use it as is in the sublibrary. An example of the JCL to compile and link the program is as follows:

```
// JOB MQBICBLD
// LIBDEF PHASE,CATALOG=user.lib
// LIBDEF *,SEARCH=PRD2.MQSERIES
// OPTION CATAL
   PHASE MQBICALL,*
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM=''
  PROCESS LIB,APOST,NOADV,RENT,BUF(4096),NODYNAM
  PROCESS NOSEQ,TRUNC(OPT)
* $$ SLI MEM=MQBICALL.COBOL,S=PRD2.MQSERIES
/*
   INCLUDE MQBIBTCH
/*
// EXEC LNKEDT
/&
```

2. We now have an executable phase in our user.lib. Before we can run the program, we need to manually start the MQ batch interface. To do this, we start the MQBI transaction in native CICS. We can check that the interface has successfully started by checking that the MQBI transaction is running using CEMT `i task`. The result should be similar to the following:

```
i task
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(00264) Tra(MQBI)           Act Tas
 Tas(00031) Tra(MQTL)           Act Tas
 Tas(00265) Tra(CEMT) Fac(SFC2) Act Ter
 Tas(00032) Tra(MQSM)           Sus Tas
```

3. Before we can run the sample program, we need to ensure that we have a test queue for messages. For this example, we use a local queue named TEST.QUEUE, owned by the queue manager identified by interface identifier MQBISRV1. To create the test queue, we use the MQMT transaction in native CICS. Rather than going through all the steps involved in creating a local queue, please refer to *MQSeries for VSE System Management Guide*, GC34-5364.

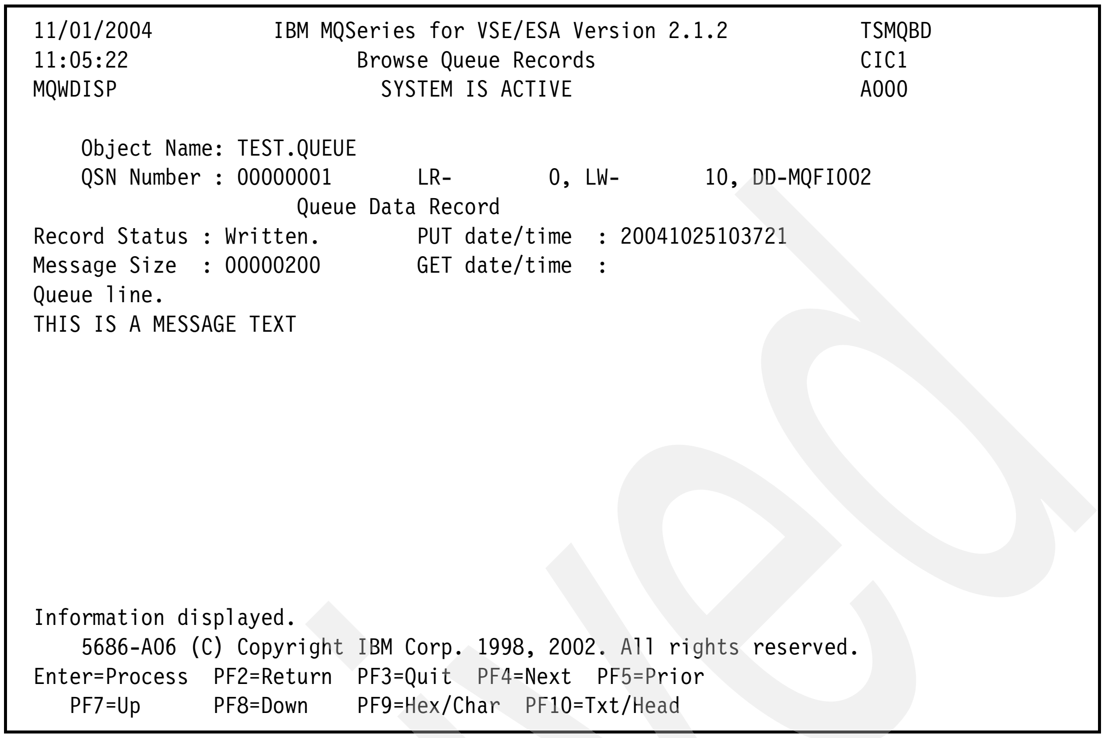
4. We can now run the sample program. For this example, we put 10 test messages to the test queue. We run the program using the following JCL:

```
// JOB MQBICRUN
// SETPARM MQBISRV=MQBISRV1
// LIBDEF *,SEARCH=(user.lib, PRD2.SCEEBASE)
/*
// EXEC MQBICALL
PUT 010 TEST.QUEUE
/*
/&
```

   The sample program writes a number of messages to the VSE console. These should appear as follows:

```
ENTERING CALLER PROGRAM
PUT 010 TEST.QUEUE
ABOUT TO CONNECT TO MQ MGR
ABOUT TO OPEN Q
PUT 0010 MESSAGES INTO QUEUE TEST.QUEUE
ABOUT TO CLOSE Q
ABOUT TO DISCONNECT FROM MQ MGR
```

5. We can further check the success of the sample program by examining the test queue to see if the test messages exist. To do this, we use MQMT option 4, Browse Queue Records. When browsed, we should see a screen similar to the one shown in Figure 8-3 on page 158.



```
11/01/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:05:22                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A000

    Object Name: TEST.QUEUE
    QSN Number : 00000001       LR-        0, LW-       10, DD-MQFI002
                      Queue Data Record
Record Status : Written.        PUT date/time  : 20041025103721
Message Size  : 00000200        GET date/time  :
Queue line.
THIS IS A MESSAGE TEXT









Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 8-3 Example of a test message queued via the batch interface.

6. In a production system that uses the batch interface, we would normally leave the MQ batch interface running for further batch programs. In this example, we stop the interface in native CICS by running the MQBI transaction with the X option. For example:

   ```
   MQBI X
   ```

   To verify that the batch interface has ended, we can use CEMT `i task` to ensure that the MQBI transaction is no longer running.

This completes our example.

8.3 Trigger application programs

An MQ trigger program is a special case of an MQ CICS program. You use MQI calls and build the program in exactly the same way. The difference is the way in which the trigger program is initiated. Trigger programs are initiated when a trigger event occurs.

Trigger events can occur when:

- A message put on a local or transmission queue
- A queue is started

Messages can be put on a queue in two ways:

- By an MQPUT or MQPUT1 call
- By a Message Channel Agent

When a queue is started, a trigger event occurs if the queue depth is greater than zero and the queue is configured for a trigger event.

When a trigger event occurs, a trigger program is initiated with trigger-related data. This data can be accessed using EXEC CICS RETRIEVE if the trigger is defined as a transaction or via the COMMAREA if it is defined as a program. The following COBOL definition shows the structure of the trigger-related data:

```
10 MQTM.
   15 MQTM-STRUCID     PIC X(4).
   15 MQTM-VERSION     PIC S9(9) BINARY.
   15 MQTM-QNAME.
      25  MQI-PROC-LOCAL-QUEUE-NAME PIC  X(48).
   15 MQTM-PROCESSNAME PIC X(48).
   15 MQTM-TRIGGERDATA PIC X(64).
   15 MQTM-TRIGGERDATA-RED  REDEFINES MQTM-TRIGGERDATA.
      25  MQI-PROC-TRANS-ID         PIC  X(4).
      25  MQI-PROC-PROGRAM-ID       PIC  X(8).
      25  MQI-PROC-TRIGGER-EVENT    PIC  X.
          88 MQI-PROC-TRIGGER-FIRST       VALUE 'F'.
          88 MQI-PROC-TRIGGER-EVERY       VALUE 'E'.
          88 MQI-PROC-TRIGGER-STOP        VALUE 'S'.
   15 MQTM-APPLTYPE    PIC S9(9) BINARY.
   15 MQTM-APPLID      PIC X(256).
   15 MQTM-ENVDATA     PIC X(128).
   15 MQTM-USERDATA    PIC X(128).
```

This definition is found in the copybooks and header files shown in Table 8-4.

Table 8-4 Copybooks and header files for trigger programs

Name	Type	Language	Use
CMQTML CMQC CMQP	C-COPY H P	COBOL C PL/I	Contains structure definition for trigger related data.

Once a trigger program has been initiated, it is the responsibility of the program to ensure its processing is logically consistent with its *trigger type*. For example, if the trigger type is F, and the trigger program is expected to processes batches of messages written to a queue in volume, the program should include logic to wait for messages rather than terminate after the first NO_MSG_AVIALABLE. This is because the trigger program may get messages faster than they are being put to the target queue.

Similarly, with a trigger type of E, the trigger program should only process one message from the target queue. This may not be the case if the maximum trigger starts value is 1. In such a case, the program may want to include wait logic similar to programs started with trigger type F. It is the responsibility of the trigger program to use the trigger start data correctly.

With a trigger type of E, the Maximum Trigger Starts parameter is also important. For example, if you have a Maximum Trigger Starts value of 10, MQSeries will start up to 10 simultaneous trigger instances as messages arrive on the queue. Typically, each instance will retrieve only one message from the queue, process it, and terminate. If there are more messages on the queue for which a trigger instance has not already been started, and the number of instances is less than the Maximum Trigger Starts, MQSeries continues to start further trigger instances until all the messages are processed.

This can lead to a problem if for any reason a trigger instance cannot process a message on the queue. If the instance terminates, and a message remains on the queue, MQSeries will start a further trigger instance. This can lead to a loop. MQSeries for VSE provides a means to avoid this problem via the MQI-PROC-TRIGGER-EVENT variable in the MQTM data structure. For trigger programs only, the MQTM data structure is passed in the program's

COMMAREA, which is passed back to MQSeries when the trigger program terminates. If your trigger program sets this variable to (S)top, MQSeries will not attempt to start a further trigger instance. Because triggered transactions are started, there is no means to return the COMMAREA to MQSeries, so triggered transactions cannot use this facility. Instead, you can use the MQSET API call to set the Triggering Enabled flag to “N”. (Refer to program sample MQPECHO, described in the *MQSeries for VSE System Management Guide*, GC34-5364, for and example of how this is done.)

Another thing to consider is the maximum trigger starts parameter of your trigger specification. This parameter tells the trigger mechanism how many concurrent trigger instances can be started at any one time. It is only relevant to a trigger type of E. Because MQSeries uses an EXEC CICS START when the trigger is a transaction, and EXEC CICS LINK when it is a program, the maximum trigger starts is treated differently in each case.

For transactions, MQSeries starts up to the number of transactions specified by maximum trigger starts. When one of these transactions terminates, MQSeries does *not* start a new trigger to maintain the maximum. This is because MQSeries has no way of knowing a STARTed transaction has ended. These are application programs after all. Instead, the MQSeries system monitor transaction (MQSM), which runs while MQSeries is active, detects that there are messages on the queue, and the number of active trigger instances has fallen under the Maximum Trigger Starts. When this is detected, the system monitor task starts an appropriate number of trigger instances, up to the Maximum Trigger Starts value. Because the system monitor makes this check at a frequency specified by the queue manager's System Wait Interval, the number of active trigger instances can rise and fall during trigger processing. If this is unacceptable, you should use triggered programs rather than transactions.

For triggered programs, MQSeries can keep track of trigger termination, and consequently, it continually triggers the program to keep the number of instances at the maximum. Remember that triggering occurs as messages are written to the queue, and as trigger instances terminate.

8.4 Application design and MQSeries locking

When designing applications it is helpful to understand how MQSeries manages concurrent access to queues.

MQSeries keeps track of a queue sequence number (QSN) for each active queue. The QSN is incremented by 1 for each message placed on the queue. When a message is placed on a queue, using MQPUT or MQPUT1, the current QSN is used in the key of the VSAM record that stores the message. If an application rolls back, the QSNs of those rolled-back messages are not reused. The QSN is always incremented until the VSAM file that hosts the queue is reorganized.

When multiple applications are putting to the same queue, messages are keyed by the current QSN. If two applications are putting messages to the same queue, the messages may be interleaved. For example:

```
App1 → Message 1
       Message 2 ← App2
App1 → Message 3
       Message 4 ← App2
App1 → Message 5
```

In the above example, if App2 decides to roll back, the queue would retain the following messages (assume the message number is the QSN):

```
App1 → Message 1
App1 → Message 3
App1 → Message 5
```

This is important if the order of messages on a queue is important to the application or applications that process them.

MQSeries for VSE uses its lock tables to manage concurrent access. The lock tables are internal data tables allocated by MQSeries when queues are opened. Each queue has a global lock table, and a local lock table for each active MQOPEN issued by applications. The size of these tables is determined by the queue definition. Generally, the size of these tables can be small, unless a queue is expected to be targeted by multiple concurrent applications, or multiple applications issuing direct or random gets (that is, using the MSGID and CORRELID fields to get specific messages).

The lock tables keep track of message ranges that have been retrieved from the queue, so that concurrent applications do not retrieve the same message. As is the case when putting messages, the retrieval of messages can also be interleaved. For example:

```
App1 → Message 1
App1 → Message 2
       Message 3 → App2
App1 → Message 4
```

An application that has retrieved messages may roll back, freeing ranges of messages, that can subsequently be retrieved by other applications. In such cases, this means that messages are not necessarily retrieved in the order they are placed on the queue. For example:

```
App1 → Message 1
App1 → Message 2
       Message 3 → App2
App1 → Message 4
```

At this point, if we assume App2 rolls back:

```
App1 → Message 3
```

So from the perspective of App1, the messages were retrieved in the following order:

```
App1 → Message 1
App1 → Message 2
App1 → Message 4
App1 → Message 3
```

Your application design needs to take these idiosyncrasies into consideration. If it is imperative that messages are processed in the order they are placed on the queue, you should not allow multiple applications to access the same queue.

Alternatively, you can get messages using the direct rather than sequential method. Each message on a queue has an MQ message descriptor (MQMD). The message descriptor, among other things, has a MSGID field and a CORRELID field. These fields can be used to uniquely identify a message, and consequently, can be used with MQGET to get a specific message.

Generally, when messages are put to a queue, MQSeries creates a default unique value for the MSGID field which includes a binary time stamp. Alternatively, applications can devise their own MSGID and CORRELID values. This allows other applications that anticipate such values to retrieve specific messages in a specific order. For example, an application may prefix the 24-character MSGID field with the characters `REQ` followed by a 3-digit sequence number. An application that processes the messages can then use MQGET and specify a MSGID of `REQ001`, then `REQ002` and so forth. To avoid interleaving, a second application that processes the same queue might look for messages with a different prefix.

One thing to remember when designing your applications is that the direct method of processing queues has considerable overhead. The sequential approach is significantly more efficient.

There is also the case where one application is putting messages to a queue, while another is getting messages from the same queue. Messages are not available until they are committed using CICS SYNCPOINT. However, the local lock table of the syncpointing application is not merged with the global lock table until a subsequent MQI call is issued. That is why it is recommended with MQSeries for VSE that MQ/VSE CICS applications follow a SYNCPOINT with an MQCMIT and SYNCPOINT ROLLBACK with an MQBACK.

Once the lock tables are merged, the messages are available to other applications. Consequently, if an application attempts to get a message before messages are committed, the MQGET call returns MQRC_NO_MSG_AVAILABLE. To avoid this, applications can use the MQGMO_WAIT option with the MQGET call. For example:

```
App1 (Issues MQGET with wait)
   App2 MQPUT Message 1
   App2 MQPUT Message 2
   App2 MQPUT Message 3
   App2 SYNCPOINT
   App2 MQCMIT
App1 (MQGET returns Message 1)
```

Because the lock tables keep track of ranges of messages that are no longer available, in a worse case scenario, of interleaving 1 for 1, the lock table can fill up unless your application SYNCPOINTs at the frequency of the lock value. For example, if you have global and local lock values of 200 for a queue, you should commit your units of work at least after every 200 updates. A value of 200 means MQSeries can keep track of 200 ranges. In the case of interleaving 1 for 1, the table may look like this:

```
0001 - 0001
0003 - 0003
0005 - 0005
0007 - 0007
```

etc.

Each range identifies a range of message QSNs that are no longer available. To avoid a lock table filling up, you must commit your work at a frequency less than or equal to the lock value. If you do not have multiple applications processing the same queue, this concern is alleviated, and the lock value can be quite small.

8.5 Application design and reply-to queues

Reply-to queues can serve a variety of functions. They are a means for applications to identify where a message response should be sent if one is needed. This decision is up to the application. A likely scenario is where a client application sends a “request” message and waits for a “reply” from a server application.

Reply-to queues are also used by MQSeries when special “report” messages are generated by the system. MQSeries for VSE supports several report messages, including:

```
MQRO_COA
MQRO_COA_WITH_DATA
MQRO_COA_WITH_FULL_DATA
MQRO_COD
MQRO_COD_WITH_DATA
MQRO_COD_WITH_FULL_DATA
MQRO_EXPIRATION
MQRO_EXPIRATION_WITH_DATA
MQRO_EXPIRATION_WITH_FULL_DATA
```

Each of these is described in the following sections.

In addition, MQSeries for VSE uses the reply-to queue in the MQMD of PCF command message as a destination for PCF reply messages. A special case of this occurs with the MQSeries Command utility (MQPMQSC). The MQSeries Command utility uses the system command reply queue specified in the queue manager's communication settings as a reply-to queue for MQSC commands generated by the command utility.

You can write an application that generates your own PCF commands and specify your own reply-to queue in the MQMD of messages sent to the system command queue. A special case of PCF commands is the Escape PCF. The Escape PCF allows you to specify MQSC-type commands (that is, verb-based commands) as a text field in the Escape PCF message. Consequently, you can write your own equivalent of the MQSeries Command utility by generating Escape PCF messages, and using the MQMD ReplyToQ field to designate a queue for responses. Responses to Escape PCF commands are Escape PCF responses. The format of both are described in the *MQSeries for VSE System Management Guide*, GC34-5364.

8.5.1 Applications and report messages

Reply-to queues can be very useful to application programs interested in keeping track of the arrival and delivery of messages. The arrival of a message occurs when a message is first put to a queue; the delivery of a message occurs when it is read.

Application programs can request reply acknowledgment of arrival and delivery by setting certain flags in parameters passed to MQPUT and MQPUT1 calls. The same settings apply for both MQI calls.

When application programs, or the receiver MCA, are putting messages to a local or remote queue, they can request an MQSeries *report* message when the message is initially written or when it is read. Such report messages are written to a reply-to queue, which is just another local queue.

The request for a report message is made when the message is put to a queue by setting the appropriate flags in the Message Descriptor (MD). The MD is a parameter to the MQPUT and MQPUT1 calls. It is also an output parameter to MQGET calls.

The MD parameter is described in the *MQSeries Application Programming Reference*, SC33-1673. There are four fields in the MD of immediate relevance:

- Report
- MsgType
- ReplyToQ
- ReplyToQMgr

The Report field enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also (for both reports and replies) how the message and correlation identifiers in the report or reply message are to be set.

For reply messages, there are several options that can be specified with the Report field. These are described in Table 8-5.

Table 8-5 Message description report options for reply-to requests

Report Option	Description
MQRO_COA	Confirm-on-arrival reports required. Message data from the original message is not included with the report message.
MQRO_COA_WITH_DATA	Confirm-on-arrival reports with data required. This is the same as MQRO_COA except that the first 100 bytes of the application message data from the original message is included in the report message.
MQRO_COA_WITH_FULL_DATA	Confirm-on-arrival reports with full data required. This is the same as MQRO_COA except that all of the application message data from the original message is included in the report message.
MQRO_COD	Confirm-on-delivery reports required. Message data from the original message is not included with the report message.
MQRO_COD_WITH_DATA	Confirm-on-delivery reports with data required. This is the same as MQRO_COD except that the first 100 bytes of the application message data from the original message is included in the report message.
MQRO_COD_WITH_FULL_DATA	Confirm-on-delivery reports with full data required. This is the same as MQRO_COD except that all of the application message data from the original message is included in the report message.

The MsgType field indicates the type of message. For reply messages requests, the MsgType should include MQMT_REQUEST, which means the message requires a reply. When an application gets a reply message, the MsgType is MQMT_REPLY.

The ReplyToQ field is the name of the message queue to which the reply message should be sent. The name is the local name of a queue that is defined on the queue manager identified by ReplyToQMgr. For confirm-on-arrival requests, the reply is sent when the object message (as opposed to the reply message) is first put to a queue. For confirm-on-delivery requests, the reply is sent when the object message is read.

If the ReplyToQMgr field is blank, the local queue manager looks up the ReplyToQ name in its own queue definitions. If a local definition of a remote queue exists with this name, the ReplyToQ value in the transmitted message is replaced by the value of the RemoteQName attribute and the ReplyToQMgr value is replaced with the RemoteQMgrName from the definition of the remote queue. If a local definition of a remote queue does not exist, ReplyToQ is unchanged and ReplyToQMgr defaults to the local queue manager.

The options you select in the Report field determine the nature of the reply message. Obviously, the presence or absence of WITH_DATA affects the data portion of the reply message. Otherwise, the values in the message descriptor tell you the message is a reply message.

For application design, you may consider using individual queues dedicated to receiving reply messages.

8.5.2 Applications and message expiry

Applications can place messages on a queue with an expiry value. This is done by setting the Expiry field in the message's MQMD to a value other than MQEI_UNLIMITED. The Expiry value specifies an interval in tenths of a second. If a message with an Expiry value is not retrieved within that expiry interval it is deemed expired, and is not returned to applications issuing MQGET calls. MQSeries has its own processes to flag expired messages as logically deleted, independent of application programs.

MQSeries determines whether a message has expired by the local time of the queue manager. When a message is placed on a local queue of the queue manager, the local time is stored with the message. When an attempt is made to retrieve the message, MQSeries checks this time with the current local time and the Expiry value. If the message has expired, it is not returned and, depending on the type of MQGET call, MQSeries continues to try to find a suitable message to return the application.

This approach means that queue managers across an MQSeries network do not need to be configured to exactly the same time or time zone. For example, if a message is placed on a transmission queue at 13:45:10 with an Expiry of 30 seconds (300), and is subsequently transmitted to a queue manager with a local time of 12:43:25, the message will not expire until 12:43:55.

Note, however, that the time between the message being placed on the transmission queue and the time the Sender MCA retrieves the message to send it is deducted from the Expiry value. For example, if the message is placed on the transmission queue at 13:45:10, and is retrieved by the Sender MCA at 13:45:12, the Expiry value in the MQMD received by the remote queue manager will be reduced to 28 seconds (280).

MQSeries allows you to request an expiry report message in the event that a message expires. This is done by setting the Report field in the message's MQMD to one of the following:

```
MQRO_EXPIRATION
MQRO_EXPIRATION_WITH_DATA
MQRO_EXPIRATION_WITH_FULL_DATA
```

MQSeries will not generate expiry report messages for messages that have an Expiry value of MQEI_UNLIMITED, because these will never expire.

As with COA and COD report messages, you can request that none, some, or all of the data of the expiring message is reproduced in the expiry message. Once again, report messages, when generated, are sent to the ReplyToQ specified in the expiring message's MQMD.

8.6 MQSeries–CICS bridge

The MQSeries–CICS bridge is a component of MQSeries for VSE that allows direct access from MQSeries applications to applications on your CICS system. In bridge applications there are no MQSeries calls within the CICS application (the bridge enables implicit MQI support). This means that you can reengineer legacy applications that were controlled by 3270-connected terminals to be controlled by MQSeries messages, without having to rewrite, recompile, or relink them.

The bridge enables an application that is not running in a CICS environment to run a program or transaction on CICS and get a response back. This non-CICS application can be run from any environment that has access to an MQSeries network that encompasses MQSeries for VSE.

For the bridge to be available to remote applications, the MQSeries–CICS bridge monitor transaction, CKBR, must be running in the CICS partition that hosts MQSeries for VSE.

The bridge monitor is a long-running transaction that waits for bridge request messages to arrive on a special queue designated for such requests. The default name for this queue is SYSTEM.CICS.BRIDGE.QUEUE, however, you can specify the name of a different queue when you start the CKBR transaction.

Request messages indicate whether the bridge should start a program or a transaction, and specify in the ReplyToQ field of the message's MQMD where the bridge should send reply messages. An application that uses the bridge will generate a request message and then wait for and process the reply. In the case of a pseudo-conversation, this may be an iterative process.

In the following sections, we step through some program examples using the bridge.

8.6.1 Bridge data structures

The use of data structures by bridge programs differs depending on whether the program is bridging to a DPL program or a 3270 transaction.

A DPL program is a CICS program that uses only the DPL subset of CICS commands. Essentially, this is a non-3270 program that can be called via CICS LINK. A DPL program can simply run, perform its function and return, or be passed data in the CICS commarea, perform its function and return data in the commarea.

Using the bridge, DPL programs can also be called multiple times within a unit of work, and syncpointed or rolled back collectively.

The simplest case of bridging to a DPL program is to build a request message that just identifies the name of the program that the bridge should run.

MQMD	ProgName

In this case, the bridge monitor will get the request message, and start the identified CICS program, with no commarea. The Format in the request message's MQMD can be set to MQFMT_STRING.

To pass data to the program in the commarea, you can append additional data to the request message following the program name. The program name must be 8 characters, padded with spaces if necessary.

MQMD	ProgName	CommareaData

In this case, the bridge monitor will LINK to the program and pass the appended message data in the commarea. The length of the commarea is equivalent to the length of the appended message data. The bridge always attempts to perform code page conversion if it is necessary, so if the Format of the MQMD is MQFMT_STRING, your appended message data will be treated as a string.

If you want to include the execution of a program within a single unit of work, or if your appended data needs special conversion, prepend the request message with a CICS Information Header (MQCIH) data structure. The MQCIH data structure is defined in the C language header file, CMQC.H and the COBOL copybooks CMQCIHL and CMQCIHV. For example:

MQMD	MQCIH	ProgName

The MQCIH contains a UOWControl field which indicates whether the execution of a program is part of a unit of work. For example, the UOWControl field can be set to indicate that the execution of a program is the first, in the middle, or the last for the current unit of work. It can also be used to request syncpoint or rollback.

To pass data to a program that runs in a unit or work, simply append the data to the end of the request message.

MQMD	MQCIH	ProgName	CommareaData

Once again, the bridge monitor will LINK to the program and pass the appended message data in the commarea. The program is free to change this data, which ultimately will be sent back to requester in a bridge reply message.

The MQCIH structure also contains a Format field that can be used if you need special conversion of the commarea data. However, all request messages that prepend an MQCIH structure should set the Format in the message's MQMD to MQFMT_ CICS.

Bridge reply messages to bridge requests have the same format as the request. So if you request a program without commarea data, the reply will contain only the program name. If you request a program with commarea data, the reply will consist of the program name followed by the commarea data (as and if modified by the program). Requests with the MQCIH structure receive replies with the MQCIH structure.

If the program ends abnormally, a message is returned to the reply queue with the following structure, whether or not the inbound message preceding the failure contained an MQCIH:

MQMD	MQCIH	MQI* message

`MQI* message` represents an error message that indicates the error type. The value of field MQCIH.Format is set to MQFMT_STRING, so that the message can be properly converted if the final destination uses a different CCSID and encoding. The MQCIH also contains other fields that you can use to diagnose the problem.

Bridging to 3270 transactions

Bridging to 3270 transactions is different in some respects from bridging to DPL programs. Firstly, request messages always have a prepended MQCIH structure. Also, the transaction name is a field in the MQCIH structure rather than a separate data element of the message (like the program name in DPL requests).

The simplest case of bridging to a 3270 transaction is to build a request message that contains only the MQCIH structure, which includes the TransactionId field that names the transaction the bridge should start.

MQMD	MQCIH

Typically, 3270 transactions are interactive, that is, they issue calls like CICS SEND and RECEIVE. Consequently, requests to run 3270 transaction normally include additional data structures called vectors.

Inbound vectors are data structures you might include in your request messages. Outbound vectors are structures you might receive in a reply message. Your bridge application, which generates requests and processes reply messages, needs to understand the interactive operation of the transaction you are running. For example, if the transaction issues a CICS RETRIEVE in its initialization, your request message must include a RETRIEVE vector. If the transaction issues a CICS RECEIVE, you must provide a RECEIVE vector. Similarly, if the transaction issues a CICS send, your application must understand a SEND vector to process the reply.

Vectors have a standard 16-byte header as follows:

```
typedef struct {
   signed long int brmq_vector_length;
   char            brmq_vector_descriptor[4];
   char            brmq_vector_type[4];
   char            brmq_vector_version[4];
} brmq_vector_header;
```

The vector header is followed by a vector structure relative to the vector type. For example, a RETRIEVE vector has the following definition:

```
typedef struct {
   char            filler__001[16];
   char            brmq_rt_rtransid[4];
   char            brmq_rt_rtermid[4];
   char            brmq_rt_queue[8];
   signed long int brmq_rt_data_len;
} brmq_retrieve;
```

Note that the first 16 bytes of the RETRIEVE vector are reserved for the vector header. Vectors are described in the *CICS External Interfaces Guide*, SC33-1669. Vector definitions are provided in CICS header files and copybooks as follows:

```
DFHBRMQO.C for COBOL
DFHBRMQH.H for C
DFHBRMQL.P for PL/I
```

A request message with a vector has the following format:

MQMD	MQCIH	BRMQ structure

Some vectors require additional data appended to the BRMQ structure. A RETRIEVE vector is an example. If the brmq_rt_data_len value is greater than 0, then the vector should be appended with data up to this length. For example:

MQMD	MQCIH	BRMQ structure	data

This principle is true for all vectors that expect additional data. You can also include multiple vectors and vector data in a single request message.

MQMD	MQCIH	BRMQ structure	data	BRMQ structure	data

If the transaction you are running issues a CICS command for which you have not provided an appropriate vector, the bridge will send a reply message requesting the needed vector information. Your application can then send another request message to satisfy this request.In the case of Basic Mapping Support (BMS) calls like SEND MAP and RECEIVE MAP, the data appended to the vector is called an Application Data Structure (ADS). ADS has a long and a short form that describes the BMS data stream. By using ADS definitions, your application can generate and process BMS data streams without knowledge of the BMS map definition.

ADS structures are described in the *CICS External Interfaces Guide*, SC33-1669.

8.6.2 Bridge programming

Writing a bridge application basically involves putting a request message on the bridge request queue to run a program or transaction, and then waiting for a reply.

For DPL programs, the reply is generally modified data in the commarea. For 3270 transactions, the data is more complex. For example, it may include SEND or SEND MAP data for your application to process, and it may require data appropriate for a RECEIVE or RECEIVE MAP. In both cases, the interaction between your application and a program or transaction via the bridge may be an iterative process.

DPL bridge programming

In this section, we step through an application that runs a DPL program that passes commarea data, and receives modified data on return. The application runs as an MQ client.

Firstly, the application needs to connect to the queue manager.

```
memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
MQCONN(QMName, &hcon, &cc, &rc);
```

Next, we need to open the bridge request queue and a reply queue. We place our request to run a program on the request queue, and then wait for a reply message from the bridge.

```
memset(od.ObjectQMgrName, ' ', MQ_Q_MGR_NAME_LENGTH);
memcpy(od.ObjectName, BRIDGE_Q, strlen(BRIDGE_Q));
O_opts = MQOO_OUTPUT;
MQOPEN(hcon, &od, O_opts, &hobj_b, &cc, &rc);

memset(od.ObjectQMgrName, ' ', MQ_Q_MGR_NAME_LENGTH);
memset(od.ObjectName, ' ', MQ_Q_NAME_LENGTH);
memcpy(od.ObjectName, REPLY_Q, strlen(REPLY_Q));
O_opts = MQOO_INPUT_SHARED;
MQOPEN(hcon, &od, O_opts, &hobj_r, &cc, &Reason);
```

Because we are going to put a message to the request queue, we open the bridge request queue for output. Because we are going to get a message from the reply queue, we open it for input.

We can now prepare and put our request message to the request queue. The program we want to run in CICS is called MQBRTES0, and we will pass 8 bytes in the commarea.

```
reqmsg = (PMQVOID)&msgbuf;
strcpy(reqmsg, "MQBRTES0DATA IN ");
blen = strlen(reqmsg);
memset(md.ReplyToQ, 0, 48);
memcpy(md.ReplyToQ, REPLY_Q, strlen(REPLY_Q));
memcpy(md.CorrelId,
          MQCI_NEW_SESSION,
          MQ_CORREL_ID_LENGTH);
memcpy(md.Format,   MQFMT_STRING, MQ_FORMAT_LENGTH);
MQPUT(hcon, hobj_b, &md, &pmo, blen, reqmsg, &cc, &rc);
```

We must remember to set the ReplyTo queue field in the MQMD to the name of our reply queue. This is where we expect to receive a reply message from the bridge. We also need to set the MQMD's CorrelId to MQCI_NEW_SESSION. This tells the bridge that we are starting a new unit of work. Because the program name and the commarea data are alphanumeric, we can set the Format to MQFMT_STRING.

At this point we have put a request on the request queue. However, the bridge cannot process the request until it is committed.

```
MQCMIT(hcon, &cc, &rc);
```

The bridge can now get the request from the request queue. It can then extract the program name, MQBRTES0, and LINK to it, passing the appended data in the commarea.

The application can now prepare for and get the reply.

```
repmsg = (PMQVOID)&msgbuf;
mlen = blen;
gmo.WaitInterval = 10000;
gmo.Options = MQGMO_WAIT | MQGMO_CONVERT;
memcpy(md.CorrelId, md.MsgId, MQ_CORREL_ID_LENGTH);
MQGET(hcon, hobj_r, &md, &gmo, mlen, repmsg, &dlen,
      &cc, &rc);
```

Because it may take a second or more for the bridge to start the program and for it to run, we need to tell the MQGET call to wait. In this case, we set WaitInterval to 10 seconds, which should be more than enough for the program to run and the bridge to send a reply. We also need to request data conversion because our client runs on an ASCII machine.

Lastly, before we issue the MQGET, we need to set the CorrelId to the MsgId. The MsgId, because we have not reset it, has a unique 24-byte value set by the MQPUT call. The bridge remembers this value and uses it to set the MsgId and CorrelId of its reply message. This way, our application can get the correct reply message.

Once the MQGET returns successfully, we can examine the reply message and process the commarea data.

Finally, we can close the request and reply queues and disconnect from the queue manager.

```
C_opts = MQCO_NONE;
MQCLOSE(hcon, &hobj_r, C_opts, &cc, &rc);

C_options = MQCO_NONE;
MQCLOSE(hcon, &hobj_b, C_opts, &cc, &rc);

MQDISC(&hcon, &cc, &rc);
```

A full listing of this example is available in “DPL bridge sample” on page 286.

3270 bridge programming

In this section, we step through an application that runs a 3270 transaction. For simplicity, we will run the CEMT transaction, which uses CICS CONVERSE calls. The application will run as an MQ client.

As usual, the application first needs to connect to the queue manager.

```
memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
MQCONN(QMName, &hcon, &cc, &rc);
```

Next, we need to open the bridge request queue and a reply queue. We place our request to run CEMT on the request queue, and then wait for a reply message from the bridge.

```
memset(od.ObjectQMgrName, ' ', MQ_Q_MGR_NAME_LENGTH);
memcpy(od.ObjectName, BRIDGE_Q, strlen(BRIDGE_Q));
O_opts = MQOO_OUTPUT;
MQOPEN(hcon, &od, O_opts, &hobj_b, &cc, &rc);

memset(od.ObjectQMgrName, ' ', MQ_Q_MGR_NAME_LENGTH);
memset(od.ObjectName, ' ', MQ_Q_NAME_LENGTH);
memcpy(od.ObjectName, REPLY_Q, strlen(REPLY_Q));
O_opts = MQOO_INPUT_SHARED;
MQOPEN(hcon, &od, O_opts, &hobj_r, &cc, &Reason);
```

Because we are going to put a message to the request queue, we open the bridge request queue for output. Because we are going to get a message from the reply queue, we open it for input. We can now prepare and put our request message to the request queue. This is where things differ significantly from bridging to a DPL program.

From experience (or experimentation) we know that the CEMT transaction issues a CICS RECEIVE to receive input parameters from the terminal. In our case we want to run CEMT with the parameters `I TASK`. That is:

```
CEMT I TASK
```

CEMT will then issue a CICS CONVERSE to send a task list display to the terminal, and get a response from the terminal. We could provide the receive and converse data in two request messages, but for simplicity, we will provide everything in a single request message.

Our request message has the following structure:

```
struct tagCEMT
{
    MQCIH                cih;
    union
    {
      brmq_vector_header  vh;
      brmq_receive        rv;
    } rec;
    MQCHAR               Data[12];
    union
    {
      brmq_vector_header  vh;
      brmq_converse       cv;
    } con;
} CEMT;
```

Our CEMT request begins with an MQCIH structure, which is required by all 3270-type bridge requests. It is then followed by a receive vector to satisfy the CEMT transaction's initial CICS RECEIVE. The 12-byte data area following the receive vector contains data for the receive, in this case: `CEMT I TASK`. This data will be passed to the CEMT transaction by the bridge, as though it had been received from a terminal.

At this point the CEMT transaction issues a CICS CONVERSE call to send the task list display and wait for a terminal response. Anticipating this, our CEMT request has a converse vector, which we will use to respond with PF3, and terminate the CEMT transaction.

So first, we need to set up the MQMD.

```
memset(md.ReplyToQ, ' ', 48);
memcpy(md.ReplyToQ, REPLY_Q, strlen(REPLY_Q));
memcpy(md.CorrelId, MQCI_NEW_SESSION,
       MQ_CORREL_ID_LENGTH);
memcpy(md.Format,   MQFMT_CICS, MQ_FORMAT_LENGTH);
```

As with all requests, we must remember to set the ReplyTo queue field in the MQMD to the name of our reply queue. This is where we expect to receive a reply message from the bridge. We also need to set the MQMD's CorrelId to MQCI_NEW_SESSION. This tells the bridge that we are starting a new unit of work. Because requests to bridge to 3270 transactions always start with an MQCIH structure, we must set the Format to MQFMT_CICS. This will allow the bridge to convert the request to the VSE queue manager's local code page.

Next, we set up the MQCIH.

```
memcpy(&CEMT.cih, &cih, sizeof(MQCIH));
CEMT.cih.LinkType           = MQCLT_TRANSACTION;
CEMT.cih.ConversationalTask = MQCCT_NO;
CEMT.cih.UOWControl         = MQCUOWC_ONLY;
CEMT.cih.Encoding           = MQENC_NATIVE;
CEMT.cih.CodedCharSetId     = 850;
memcpy(CEMT.cih.Format, "CSQCBDCI", MQ_FORMAT_LENGTH);
memcpy(CEMT.cih.TransactionId, "CEMT", strlen("CEMT"));
```

First, we copy default values into the structure. Then we set the LinkType to MQCLT_TRANSACTION to indicate that we are bridging to a transaction rather than a DPL program. We set ConversationalTask to MQCCT_NO because we are going to provide everything the CEMT transaction will need in a single request message. For similar reasons, we set UOWControl to MQCUOWC_ONLY. Because our application is running on an ASCII machine, we need to let the queue manager know the client's integer encoding and CCSID. We also need to set the Format to CSQCBDCI, which names a special data conversion exit that will allow the queue manager to convert the MQCIH and appended vectors. The CSQCBDCI exit is provided with MQSeries for VSE. Lastly, we identify the transaction we want to run, in this case, CEMT.

We can now set up the receive vector header.

```
CEMT.rec.vh.brmq_vector_length = sizeof(brmq_receive)+12;
memcpy(CEMT.rec.vh.brmq_vector_descriptor, "0402", 4);
memcpy(CEMT.rec.vh.brmq_vector_type,       "I   ", 4);
memcpy(CEMT.rec.vh.brmq_vector_version,    "0000", 4);
```

The vector length includes the size of the vector plus appended data relevant to the vector. In our case, we append an additional 12 bytes, which will contain input parameters for CEMT. Each vector has its own unique descriptor. For a receive, the descriptor is "0402". The vector type is "I" for input, and the version is "0000", which is currently the only supported value (even though the *CICS External Interfaces Guide* says this field should be set to X'00000000').

Following the header is the receive vector and its data.

```
memcpy(CEMT.rec.rv.brmq_re_transmit_send_areas,"Y   ", 4);
memcpy(CEMT.rec.rv.brmq_re_buffer_indicator,   "N   ", 4);
memcpy(CEMT.rec.rv.brmq_re_aid, ENTER, 4);
CEMT.rec.rv.brmq_re_data_len = 12;

memcpy(CEMT.Data, "CEMT I TASK ", 12);
```

In the receive vector, we need to set transmit_send_areas to YES to ensure CEMT's send areas are preserved (otherwise we won't end up with the task list output). We set buffer_indicator to NO because the CICS RECEIVE does not use the BUFFER option. We simulate the Enter key by setting the attention identifier to ENTER (which we have defined as X'7D404040'). The terminal data we are simulating is 12 bytes, that is `CEMT I TASK`.

The converse vector header is similar to the receive vector header, except that it does not have appended data (because we are only simulating PF3), and the unique descriptor for a CICS CONVERSE is "0406".

```
CEMT.con.vh.brmq_vector_length = sizeof(brmq_receive);
memcpy(CEMT.con.vh.brmq_vector_descriptor, "0406", 4);
memcpy(CEMT.con.vh.brmq_vector_type,       "I   ", 4);
memcpy(CEMT.con.vh.brmq_vector_version,    "0000", 4);
```

In the converse vector we set the transmit_send_areas to YES again so that we don't lose our output, and set the attention identifier to PF3. When CEMT receives the PF3 it will terminate, and the bridge will send our output (the task list) in a reply message. There is no appended data, so we set the data length to 0.

```
memcpy(CEMT.con.cv.brmq_co_transmit_send_areas,"Y   ", 4);
memcpy(CEMT.con.cv.brmq_co_aid, PF3, 4);
CEMT.con.cv.brmq_co_data_len = 0;
```

We can now put the request to the bridge request queue.

```
reqmsg = (PMQCHAR)&CEMT;
blen = sizeof(struct tagCEMT);
MQPUT(hcon, hobj_b, &md, &pmo, blen, reqmsg, &cc, &rc);
```

Once again, the bridge cannot process the request until it is committed.

```
MQCMIT(hcon, &cc, &rc);
```

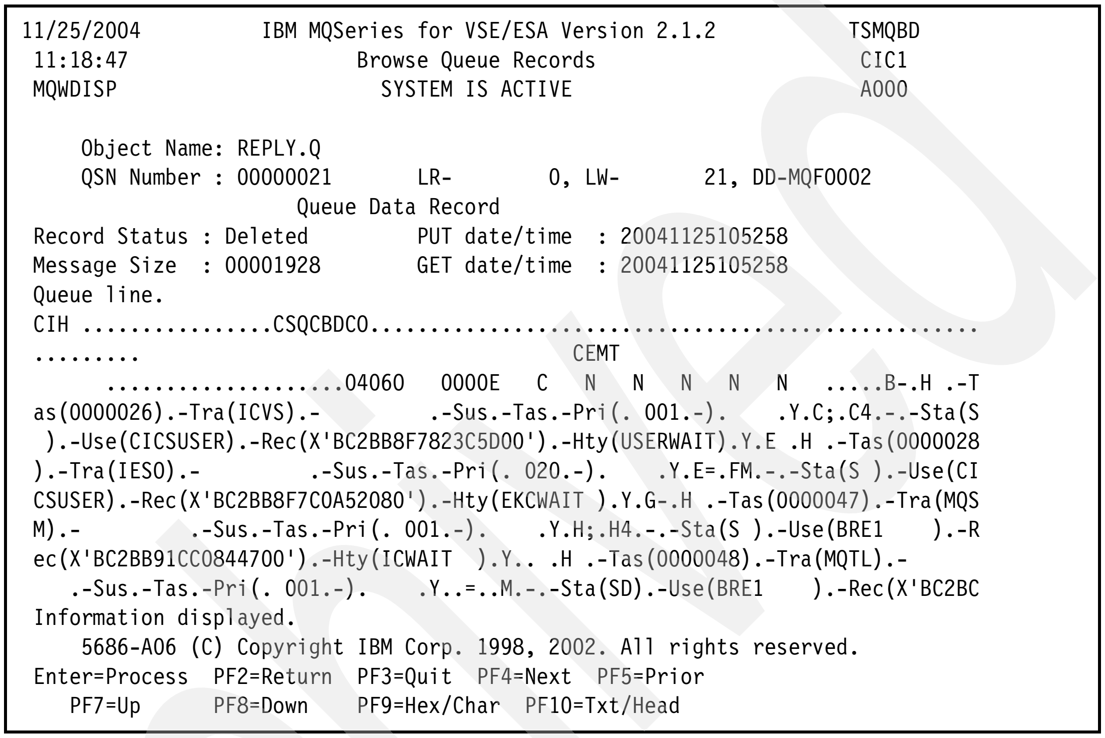
The application can now prepare for and get the reply.

```
mlen = 1000;
gmo.WaitInterval = 10000;
gmo.Options = MQGMO_WAIT |
              MQGMO_CONVERT |
              MQGMO_ACCEPT_TRUNCATED_MSG;
memcpy(md.CorrelId, md.MsgId, MQ_CORREL_ID_LENGTH);

MQGET(hcon, hobj_r, &md, &gmo, mlen, &repmsg, &dlen,
      &cc, &rc);
```

We have reserved a 1000-byte buffer for the reply message. We set the WaitInterval to 10 seconds, which should be more than enough for the transaction to run and the bridge to send a reply. We also need to request data conversion because our client runs on an ASCII machine. Because our buffer is only 1000 bytes and the reply message may be larger, for this example we accept truncation.

As with the DPL example, before we issue the MQGET, we need to set the CorrelId to the MsgId. The MsgId, because we have not reset it, has a unique 24-byte value set by the MQPUT call. The bridge remembers this value and uses it to set the MsgId and CorrelId of its reply message. This way, our application can get the correct reply message.

Once the MQGET returns successfully, we can examine the reply message and process the task list data. The following screen shot displays the first 790 bytes of the reply message.



```
11/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
 11:18:47                 Browse Queue Records                      CIC1
 MQWDISP                    SYSTEM IS ACTIVE                        A000

     Object Name: REPLY.Q
     QSN Number : 00000021       LR-        0, LW-       21, DD-MQF0002
                      Queue Data Record
 Record Status : Deleted        PUT date/time  : 20041125105258
 Message Size  : 00001928       GET date/time  : 20041125105258
 Queue line.
 CIH ................CSQCBDCO................................................
 .........                                    CEMT
       ....................04060   0000E   C   N   N   N   N   N   .....B-.H .-T
 as(0000026).-Tra(ICVS).-        .-Sus.-Tas.-Pri(. 001.-).    .Y.C;.C4.-.-Sta(S
  ).-Use(CICSUSER).-Rec(X'BC2BB8F7823C5D00').-Hty(USERWAIT).Y.E .H .-Tas(0000028
 ).-Tra(IESO).-         .-Sus.-Tas.-Pri(. 020.-).    .Y.E=.FM.-.-Sta(S ).-Use(CI
 CSUSER).-Rec(X'BC2BB8F7C0A52080').-Hty(EKCWAIT ).Y.G-.H .-Tas(0000047).-Tra(MQS
 M).-        .-Sus.-Tas.-Pri(. 001.-).    .Y.H;.H4.-.-Sta(S ).-Use(BRE1    ).-R
 ec(X'BC2BB91CC0844700').-Hty(ICWAIT  ).Y.. .H .-Tas(0000048).-Tra(MQTL).-
    .-Sus.-Tas.-Pri(. 001.-).    .Y..=..M.-.-Sta(SD).-Use(BRE1    ).-Rec(X'BC2BC
 Information displayed.
     5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
 Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
    PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure 8-4 MQSeries–CICS bridge sample reply message

Finally, we can close the request and reply queues and disconnect from the queue manager.

```
C_opts = MQCO_NONE;
MQCLOSE(hcon, &hobj_r, C_opts, &cc, &rc);

C_options = MQCO_NONE;
MQCLOSE(hcon, &hobj_b, C_opts, &cc, &rc);

MQDISC(&hcon, &cc, &rc);
```

A full listing of this example is available in "3270 bridge sample" on page 290.

9

Security

There are two approaches to security for an MQSeries for VSE environment: you can use CICS security, or MQSeries security.

CICS security differs depending on whether you are running MQSeries in CICS/VSE or CICS TS. Security in CICS/VSE allows you to implement transaction and resource security (for example, FCT, DCT, PPT). CICS TS does not have its own security. Instead, security is handled by the Basic Security Manager (BSM), or by an External Security Manager (ESM). When CICS security is implemented, application programs may receive the NOTAUTH response from CICS command calls.

MQSeries security is an installable feature that requires an external security manager. MQSeries will authenticate connections to the queue manager, access to queues and authorization for commands. MQSeries does this by using the SAF interface to an external security manager. MQSeries security is transparent to applications, except that when it is active, applications may receive an MQRC_NOT_AUTHORIZED return code from MQI calls.

The MQSeries security feature is documented in the *MQSeries for VSE System Management Guide*, GC34-5364. This guide also provides an example of setting up security with an ESM. Consequently, there is not much more that can be added here. Instead, this chapter focuses on CICS security.

When you write your own applications to connect to MQSeries, you use the MQSeries MQI and the CICS API. The MQSeries MQI is a set of linkable objects that also use the standard CICS API. Therefore, as far as security is concerned, there are no considerations that could apply to normal applications and not to MQSeries applications.

However, the most important problem is not to protect CICS applications from people in your enterprise who may (generally) be well identified, but from the outside world, especially in client/server mode. This means that such security should be processed at the TCP/IP level (firewall, exits, and so forth) or at the application level. Therefore, for the moment, unless you entirely trust your security system, it would be wise to use MQSeries for VSE only in an intranet environment.

You may need to implement security services that are not currently in use on your production system. Our goal is to provide a general overview on how you might implement basic security. Security is handled differently for the CICS Transaction Server (CICS TS) and CICS/VSE 2.3.

9.1 Security in CICS/VSE V2.3

When you use CICS security, users need to identify themselves to CICS. This is achieved either by the logon transaction CESN (or the old CSSN) or from the Interactive User Interface (IUI) logon panel. At this point, CICS internally creates a control block called SNTTE that is associated with a terminal. This control block contains the following information, essentially retrieved from the Sign On table (or from the VSE control file if logged via the IUI):

- National Language
- Operator Class and priority
- Operator Identifier
- Operator Name
- Resource security keys
- Transaction security keys
- Terminal time-out
- Userid

If terminal operators do not log on, they may only run transactions that are not protected.

9.1.1 Transaction security

The transaction security is enabled via transaction security keys. When you define a transaction either with RDO or by assembling a Program Control Table (PCT), you can assign a security characteristic (class) to this transaction. It is called a Transaction Key. It is represented by a number in range from 1 to 64 for the TRANSEC parameter. If no value is specified for TRANSEC, 1 is used by default.

To be able to use this transaction, a user must be given authorization for the TRANSEC value. This is done via the parameter *SCTYKEY* of the user SNT entry, or if you are using the IUI, by the Maintain User Profiles dialog. Up to 64 security keys are possible for the same user. If users do not log on explicitly, they get a default profile with `SCTYKEY=1`.

9.1.2 Resource security

The Resource Security Level (RSL) is a second level of CICS security that extends the security concept to resources, such as files, transient data destinations, application programs, and temporary storage. However, using RSL requires a very good understanding of applications; otherwise, programs may receive a NOTAUTH abend condition code when they are trying to access resources.

If you want to use this second security level, you need to:

1. Define your transaction with parameter `RSLC=YES` in the PCT entry, or in the equivalent field if you use CEDA.
2. Specify an RSL key in the range of 1 to 24 when you define the resources to be accessed by this transaction. You achieve this with the RSL parameter in entries of the following tables:
   - FCT
   - JCT
   - DCT
   - TST
   - PPT (or equivalent with CEDA)
   - PCT (or equivalent with CEDA)

   If you do not want to protect a particular resource, and you set the RSLC field to `YES` for transactions, you must define this resource with `RSL=PUBLIC`.

If a resource used by a transaction has no explicit RSL key, it gets a default value of 0. In such a case, if this transaction has the RSLC field set to YES, access to this resource is denied.

3. To assign users with one or multiple resource security keys (up to 24), use the RSLKEY parameter of DFHSNT entries (or by setting equivalent fields if using the IUI dialogs). To be authorized to access a resource, the key of this resource must be specified for a user.

For more details on CICS/VSE security, please read *CICS for VSE/ESA 2.3 Resource Definition (Macro)* SC33-0709 or *CICS for VSE/ESA 2.3 Resource Definition (Online)*, SC33-0708.

9.2 Security in CICS/Transaction Server

CICS TS does not provide any security of its own. Instead, it has code for calling security routines through a special interface called RACROUTE.

When such macros are executed, control passes to the System Authorization Facility (SAF). SAF checks whether it can handle the request by itself, or if a security manager is needed to do checking against certain security files, databases, or tables.

VSE has a security manager with limited functions. It is called the Basic Security Manager (BSM). If you need more functions, you need to contact a software vendor to get an External Security Manager (ESM).

9.2.1 Basic Security Manager (BSM)

The BSM is standard and is always started if no ESM is specified with ASI procedures. BSM needs a security server started in a partition at IPL time (FB by default). This server is designed to access the security database.

As far as CICS is concerned, the transaction security items are no longer specified when you define transactions with CEDA (or DFHCSDUP). Instead, you must create, assemble, and link-edit a table into IJSYSRS.SYSLIB. The name of this phase is DTSECTXN. It can also be created through IUI dialogs. When BSM starts, it loads this phase in storage.

BSM is also used for batch applications to verify resource authorizations against files or VSE items (library, sublibrary, and members). It also uses a table for this purpose called DTSECTAB, loaded in storage as well. In CICS TS, the function of DTSECTAB has changed. Batch USER entries are not part of this table anymore, but located, as well as for CICS users, in the VSE control file.

Logon authorization

The scenario for logging on to CICS is now as follows:

1. CICS users try to log on by typing their name and password on the Good Morning panel (which may be an IUI logon panel.)
2. CICS issues a RACROUTE call to ask the security manager to check this information. That is, control is passed to SAF.
3. Because SAF cannot check the validity from either DTSECTAB or DTSECTXN, it wakes up the security server and passes the needed data.
4. The security server retrieves the user entry from the VSE control file, and passes back this record to SAF.
5. SAF returns to CICS with the user record information.

6. CICS is now able to build a profile control block for this user (or denies the logon if the user is unknown).

Transaction authorization checking

When users enter a transaction name on their terminal, the following happens:

1. CICS issues a RACROUTE macro to SAF.
2. SAF is able to retrieve the transaction security definition from DTSECTXN, checks if the access key matches one of the user's, and passes the result back to CICS.
3. CICS either accepts or refuses to execute the transaction.

9.2.2 External Security Manager (ESM)

There is no more security provided for CICS TS than we have just depicted with BSM. Therefore, if you want to have the equivalent of the CICS/VSE second level security, you must obtain an External Security Manager from a vendor.

The logic is similar: at different points of its code, CICS TS uses RACROUTE macros to connect to SAF. If SAF cannot handle the request, it passes control to the ESM.

Describing ESMs is beyond the scope of this document. Therefore, please refer to the appropriate documentation from software vendors.

9.3 Security in TCP/IP

As we said previously, TCP/IP is the most critical area as far as security is concerned because any MQSeries client can access the server queues. A common solution is to filter incoming packets, based on IP addresses, and port numbers (generally 1414 for MQSeries). Typically, this is achieved by installing filters, firewalls, or proxies.

Another solution is to check the connection validity through a TCP/IP user exit. The same exit may be used for different tasks (for example, verifying access authorization to datasets). In our particular case, the exit may be used to:

1. Verify the remote user IP address and reject the connection if the association IP_address and Port_number are not in a predefined list. To make this possible, specify in your TCP/IP configuration:

   ```
   SET SECURITY_IP=YES
   ```

2. Verify the hardware address (MAC address) of a host on your LAN. To make this possible, specify in your TCP/IP configuration:

   ```
   SET SECURITY_ARP=YES
   ```

In both cases, you need to specify the security user exit name as follows:

```
DEFINE SECURITY,DRIVER=phase_name
```

And also enable the exit:

```
SET SECURITY=ON
```

For more detailed information about TCP/IP security and how to code a user exit, please refer to *TCP/IP for VSE/ESA User's Guide*, SC33-6601. This kind of checking does not ensure that clients are the individuals you believe them to be.

Another approach could be at the application level. For example, you may create a message layout that encompasses user IDs and passwords before issuing MQPUTs. However, this does not prevent a client from reading any queue. The effort needed to implement such a solution is considerable compared to the poor results you might expect. Therefore, we do not recommend this. Instead, if you want to authenticate remote clients, consider using security channel exits, discussed in the next section.

Even if you do not implement MQSeries security, or use a TCP/IP security exit, it does not necessarily mean that it is *easy* for any user to access any queue. Keep in mind that for a user to access a queue, many conditions must be met. For example:

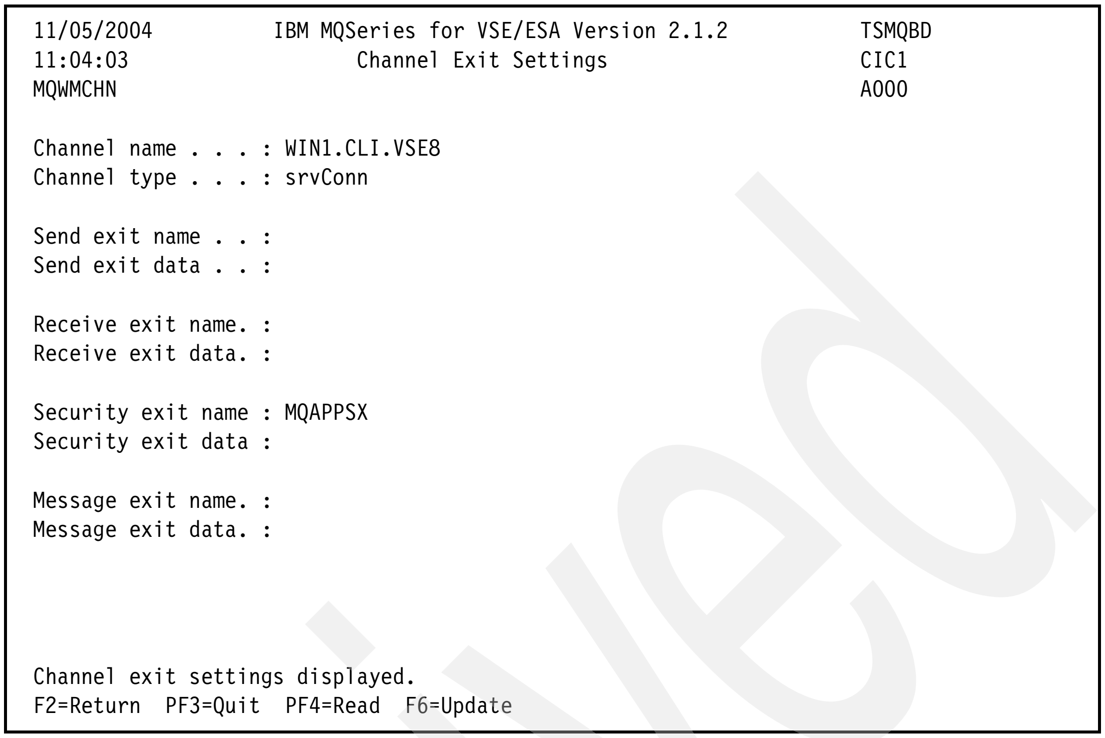
- An MQSeries client product has to be installed.
- The VSE host IP address is known.
- The port number used for the MQSeries queue manager is known.
- The VSE queue manager name is known.
- A client channel name as defined by the VSE system administrator is known.
- Local queue names are known.

In other words, only someone who knows your VSE environment and MQSeries definitions can easily penetrate your system.

9.4 Channel security exits

MQSeries also supports channel exits, including security exits. Security exits are executed when a channel is first connected. This includes client channels. You may consider using security exits for both queue-manager-to-queue-manager channels and client channels.

Channel exits are described in the *MQSeries for VSE System Management Guide*, GC34-5364. They form part of the channel definition accessible using MQMT option 1.3 and **PF11**.



```
11/05/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
11:04:03                  Channel Exit Settings                    CIC1
MQWMCHN                                                            A000

Channel name . . . : WIN1.CLI.VSE8
Channel type . . . : srvConn

Send exit name . . :
Send exit data . . :

Receive exit name. :
Receive exit data. :

Security exit name : MQAPPSX
Security exit data :

Message exit name. :
Message exit data. :




Channel exit settings displayed.
F2=Return  PF3=Quit  PF4=Read  F6=Update
```

Figure 9-1   Channel security exit definition

The security exit name in the channel exit settings identifies an executable program that is defined to your CICS system. The phase must be defined in your CICS CSD. MQSeries for VSE provides a sample channel exit, MQPCHNX.Z, in the installation sublibrary. You can write your own exit programs to meet your specific needs.

Security exits, as already mentioned, are called when a channel is first activated. For queue-manager-to-queue-manager channels, the Receiver MCA always initiates a security data flow between itself and the sender MCA if a security exit is defined. If a receiver security exit is not defined, the sender MCA initiates a security data flow, once again, if an exit is defined. For the purpose of security exits, client programs act as senders.

The security data flow includes data generated by the exit program, prepended with MQSeries header information. The data is passed by MQSeries to the security exit on the corresponding end of the channel. The security data exchange continues until both ends of the channel are satisfied. Channel operation then continues normally. Security exits can also terminate the channel.

Channel exits, including security exits, are started using CICS LINK, and are passed a set of parameters in the COMMAREA. These include:

- Channel Exit Parameters (MQCXP)
- Channel Definition (MQCD)
- Data Length (MQLONG)
- Agent Buffer Length (MQLONG)
- Agent Buffer (MQPTR)
- Exit Buffer Length (MQLONG)
- Exit Buffer (MQPTR)

In each case, MQSeries for VSE passes the address of these parameters. The MQCXP and MQCD data structures are described in the *MQSeries Intercommunication* manual, SC33-1872. These data structures contain information about the channel and the conditions under which MQSeries called the exit. They also include output fields that the exit can set to control channel operation (for example, to terminate or accept the channel).

The agent buffer contains the data flow, if any, from the remote exit. As you can see, security exits can be set up to exchange information such as user ID and password.

In the case of clients, the server (started by the Receiver MCA when a client connection is detected) can initiate a security data flow. If the client program uses an MQCONN MQI call to connect to the queue manager, it cannot specify a security exit. The client can pass a user ID and password using the environment variables MQ_USER_ID and MQ_PASSWORD. These will be passed to the security exit on the server if one is defined.

If the client is running on Windows XP, the Windows user ID is passed, but the MQ_PASSWORD variable value is not passed. In such a case, the security exit on the server might look up the user from a stored list and determine the password itself. MQSeries can then use the user ID and password provided by the security exit for authentication purposes. To do this, the security exit must put the user ID and password values in the RemoteUserIdentifier and RemotePassword fields of the MQCD data structure.

Alternatively, the client can use the MQCONNX MQI call, which allows the client program to pass an MQCNO data structure to the MQI call. The MQCNO data structure points to an MQCD data structure which in turn can identify a security exit program for the client.

If you plan to use Java clients, the MQSeries Classes for Java allow client programs to use the MQEnvironment and MQSecurityExit classes. The MQSeries Classes for Java are documented in *MQSeries Using Java*, SC34-5456.

9.5 SSL-enabled channels

Another security measure is to use SSL-enabled channels. SSL stands for Secure Sockets Layer. SSL provides a means of encrypting data sent across a TCP/IP network. A channel on MQSeries for VSE is considered SSL-enabled if its definition includes an SSL cipher specification. In addition, for SSL to be initialized and therefore available to MQSeries, the SSL Parameters (keyring sublibrary and member) in the Global Systems Definitions must be configured correctly when MQSeries is started.

The SSL cipher specification, and associated SSL parameters, are configurable from MQMT option 1.3 and **PF10** as part of the channel definition.

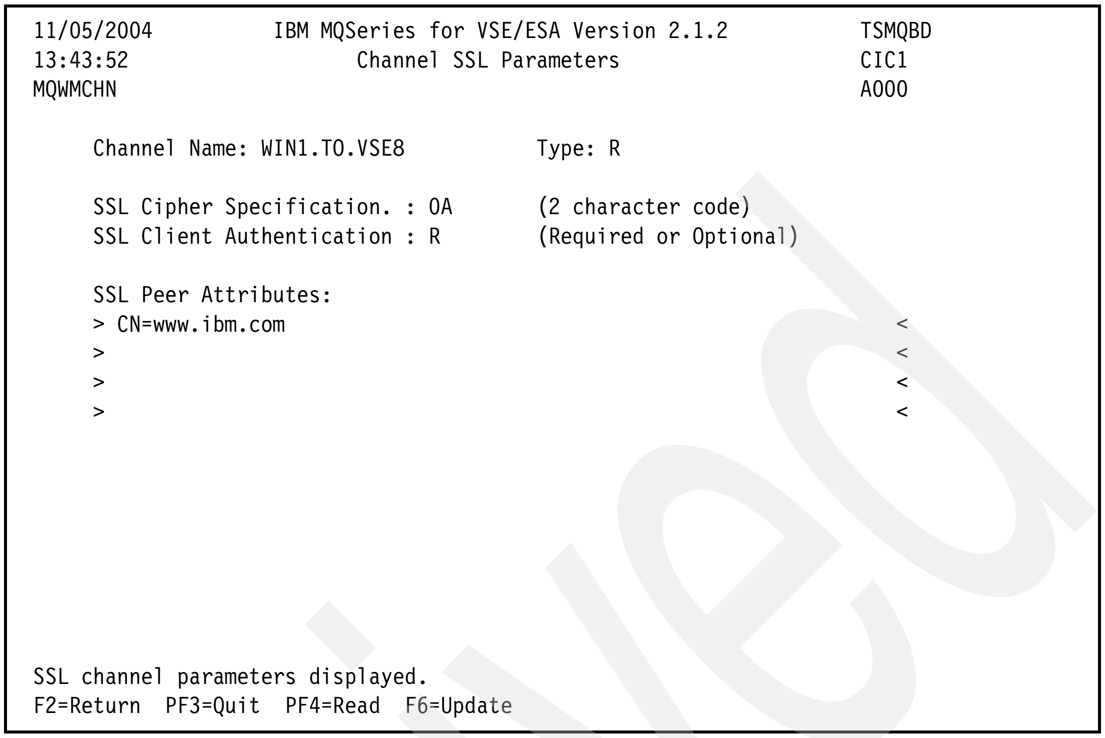

```
 11/05/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSMQBD
 13:43:52                 Channel SSL Parameters                    CIC1
 MQWMCHN                                                            A000

      Channel Name: WIN1.TO.VSE8           Type: R

      SSL Cipher Specification. : 0A       (2 character code)
      SSL Client Authentication : R        (Required or Optional)

      SSL Peer Attributes:
      > CN=www.ibm.com                                                  <
      >                                                                 <
      >                                                                 <
      >                                                                 <









 SSL channel parameters displayed.
 F2=Return  PF3=Quit  PF4=Read  F6=Update
```

Figure 9-2 Channel SSL parameters

If the SSL Cipher Specification is left blank, the channel is not considered SSL-enabled. Valid cipher specifications are listed in relevant SSL for VSE documentation. In the above example, "0A" is the SSL for VSE code for RSA1024_3DESCBC_SHA, or triple DES. When data is passed over the channel, it will be encrypted using triple DES algorithms.

When a channel is SSL-enabled, the corresponding channel on the remote system must also be SSL-enabled. It must also identify the same cipher specification.

The SSL Peer Attributes parameter allows you to demand that the X.509 certificate of the remote system contains specific details. The certificate is exchanged during channel initialization. This means a channel connection can be rejected by MQSeries if expected details are not passed by the remote system. The SSL Peer Attribute and SSL-enabled channels in general are described in the *MQSeries for VSE System Management Guide*, GC34-5364.

A comprehensive description of how to set up SSL-enabled channels is provided in 7.1, “SSL-enabled channels” on page 126.

10

Performance

There are many factors that influence the performance of MQ applications within the VSE environment. The MQ application can be viewed as part of a stack of software components, each having a bearing on the overall performance. For example:

- The application
- The MQ subsystem
- The CICS subsystem
- The file system (VSAM)
- The network (LU 6.2 or TCP/IP)
- The operating system (VSE)

This chapter only covers in detail those areas that are most relevant to MQSeries for VSE/ESA.

In addition, the overall performance is influenced by the users' expectations. For example, a user that compares the MQSeries for VSE/ESA data transfer rate across a network with the data transfer rate achieved by a specific file transfer application, such as FTP, will be disappointed. This is because they are very different applications performing very different functions. MQSeries provides a broad range of services that are not available in a simple file transfer program.

The expectations must be set at the correct level.

The performance of any computer system is governed by three general resources:

- Processor
- Memory
- I/O

If, for example, an application program has very large storage requirements because it has specified large buffer allocations, this may impact other programs attempting to run simultaneously. Similarly, if an application program has processor intensive computations, this will also impact other programs.

This chapter documents some features where the performance of MQSeries for VSE/ESA is influenced by these resources. It does not cover performance areas specific to the application or the operating system.

10.1 MQSeries performance

A good golden rule for MQSeries for VSE/ESA is that the number of queue and channel definitions should be kept to a minimum. Each unused queue and channel definition requires processing and storage resources during MQ initialization and operation.

Do not define unused test queues or test channels in a production system. Keep queue and channel definitions to a minimum.

In addition, the global system, queue, and channel definitions each specify parameters that may consume resources.

10.1.1 MQSeries parameters

The following MQSeries queue manager parameters can affect the overall performance of your system:

- System Wait Interval

  This is the wake-up frequency of MQSM, the MQ housekeeping transaction.

  An appropriate value can optimize processor and I/O resources. If this value is set too low, for example, a value of 1, the MQSM transaction is scheduled once every second. This may consume a large amount of processor and I/O resources. The default value of 30 is usually sufficient, but if you have a busy CICS system that only uses MQ applications infrequently, it may be worth experimenting to increase this value.

- Max. Recovery Tasks

  Always specify a value of zero unless dual queues are in operation. Dual queues are not recommended as they have very high I/O and processor usage.

  A good design may be much more efficient. Do not forget, messages are written into normal VSAM KSDS clusters, and therefore, you can take advantage of standard recovery functions of CICS (dynamic back out or emergency restart). Dual queues may then be avoided.

- Maximum Concurrent Queues

  The queue manager allocates various internal control blocks when MQSeries is started. Generally, these are from “above the line” storage if it is available. Because storage in CICS is a resource that needs careful management, MQSeries offers configuration parameters to control storage usage to a degree. The queue manager's *Maximum Concurrent Queues* parameter is an example.

  MQSeries allocates storage to keep track of open queues based on the *Maximum Concurrent Queues* parameter. When choosing a value for this parameter, try to estimate the number of concurrent tasks that will require queue access and set the parameter appropriately. Remember that MQSeries itself requires its own queue access for logging to the system queue. The following MQSeries features also require queue access:

  – PCF Command Server
  – Instrumentation Events
  – MQSeries–CICS Bridge

  If you are running MQSeries batch programs, do not forget to include these in your estimate.

- Maximum Number of Tasks

  The *Maximum Number of Tasks* parameter affects the amount of storage allocated by the queue manager to keep track of active application connections. Once again, you should estimate the upper limit for concurrent applications and set this parameter appropriately. Each concurrent task can have only one active connection, because MQSeries will return a warning to applications that attempt to connect more than once.

Certain MQSeries features use queue manager connections, so these need to be included in your estimate. These features are the same as those listed above for the *Maximum Concurrent Queues* parameter.

- Maximum Message Size

  The queue manager and the MCAs each allocate buffers depending on this value. This will waste storage if a value is specified that is larger than the largest actual MQ message.

  Message throughput is also a consideration when dealing with performance. Setting a small message size and writing applications that distribute small messages may save storage, but will increase the volume of messages transmitted across your MQSeries network.

  You need to design your application to determine optimal message sizes and then set the Maximum Message Size appropriately for each queue. Remember that the value specified in the queue manager definition is only used to validate the upper limit when defining queues.

- Maximum Global Locks

  The Maximum Global Locks parameter affects the amount of storage allocated by MQSeries to keep track of concurrent queue accesses. This storage is allocated once for each queue, but it is only allocated when a queue is opened for the first time after MQSeries is started. It then remains allocated until MQSeries is shut down.

  The storage is used to keep track of random accesses of messages in the queue. In effect, MQSeries keeps track of message ranges that have been read from the queue. MQSeries uses one global lock entry for each such range. Since it is possible to read every other message (worst case scenario), this parameter must be set to a value large enough to handle ranges created by such queue access. However, there is an upper limit of 1000 for this parameter.

  Generally, queues are accessed sequentially, but because concurrent tasks can access the same queue, ranges of read and unread messages can be generated as units of work are committed or rolled back.

  If you are planning to process a queue sequentially, you can set this parameter as low as 100 or even 50.

  Remember that the value specified in the queue manager definition is only used to validate the upper limit when defining queues.

- Maximum Local Locks

  The Maximum Local Locks parameter is the same as the Maximum Global Locks parameter, except that it is allocated once for each active open. For example, if Q1 is opened by two applications, there will be one global lock table and two local lock tables. Unlike the Global Lock table, the Local Lock table is released once the queue is closed.

  As with the Maximum Global Locks parameter, if you are planning to process a queue sequentially, you can set this parameter as low as 100 or even 50. If you plan to access the queue randomly, you might want to use a value as high as 1000.

  Remember that the value specified in the queue manager definition is only used to validate the upper limit when defining queues.

- Total Number of Queues

  This is not a queue manager parameter. Instead, it is the total number of queues you have defined to your queue manager. Each queue definition requires an amount of storage, which is allocated by the queue manager when the system is started.

  In a production system, you should only define those queues you plan to use, and design your applications to use as few queues as possible.

- Total Number of Channels

  This is not a queue manager parameter. Instead, it is the total number of channels you have defined to your queue manager. Each channel definition requires an amount of storage, which is allocated by the queue manager when the system is started.

  In a production system, you should only define those channels you plan to use.

The following MQSeries channel parameters can affect the overall performance of your channels:

- LU 62 retry and delay

  When an LU 6.2 connection is in an *Acquired* state, a session has to be allocated. If none is free, then the sender MCA loops the Allocation Retry times every Delay Fast seconds. Then it loops indefinitely every Delay Slow seconds until it finds a free session.

  If this happens too often, that is, if these time intervals are too small, performance may be degraded. If you have specified only a mono-session for this connection (because you know the sender MCA is the only application that uses it), you may set all relevant fields to zero.

  Note: If your connection is not in an Acquired state, there is no code in MQSeries to acquire it (as CEMT does, for example). Ensure connections are acquired to avoid unnecessary retries.

- Get retry number and delay

  The Number of Retries and Delay Time govern the termination logic of the MQ/VSE sender MCA. When a transmit queue is empty, MQ releases resources after a DISCINT (Disconnect Interval). A simple formula for calculating a DISCINT value is:

  DISCINT = (retry number + 1) * retry delay

  This should be kept as small as possible to allow the freeing of CICS resources and avoid unnecessary I/O.

  It can be weighed against the overhead of starting a new transaction when a new message arrives on the transmission queue.

  Note that when a message is put to a transmission queue the delay is canceled. So, for example, setting retry number to 3 with delay of 15 is the same as having a retry number of 0 and delay of 60. In the latter case we have not called the queue manager every 15 seconds to try to get a message. Consequently, you can set the retry number to 0 providing you set the delay appropriately.

- Max Messages per Batch

  This parameter offers the possibility of tuning for your specific system. It is more efficient for the network and reduces the frequency of syncpointing if you have a higher value for his parameter.

  Changing to any value > 1 must be weighed against the implications of recovery. For example, if your Max Messages per Batch is `1000`, and a failure occurs, all transferred messages must be reprocessed by MQSeries.

- Max Transmission Size

  A maximum of 32000 for LU 6.2 connections and a maximum of 65535 for TCP/IP connections. MQ uses a transmission header of up to 476 bytes in size.

  Use a value approximately the size of your maximum message data size plus 476. This may not be possible if the MQ messages are larger than the maximum transmission size allowed for your chosen protocol. In such a case, you should use the maximum transmission size. Do not use channels with large transmission sizes for small message transfer.

- Max Message Size

  Try to ensure this is not greater than the largest message to be sent or received. Otherwise, you waste storage for buffers.

10.1.2 MQSeries and the system log queue

Some performance considerations can be taken in regard to the MQSeries system log queue. For example:

- Minimize negotiation messages when connecting to other platforms. To do this, set all the channel values (max message size, batch size, and so forth) to the same value on each partner.
- Try and ensure that messages are transmitted at the same time or in bursts. Of course, it depends on applications, but this would reduce the number of times that negotiations occur between partners. This can also be done by extending the RETRY values for sender channels on MQ/VSE and the DISCINT values for sender channels on partners. This must be done with great caution and on a trial and error basis. Large values for the DISCINT and RETRY values hold CICS resources for the duration of the transmission.
- Ensure that all applications correctly issue MQCLOSE and MQDISC. Each time an MQ application terminates without using either of these statements, then the MQ housekeeping transaction (MQSM) must perform the close and disconnect for the application. This will also produce one or more messages on the system log queue that must be processed by the MQ message handler.

Another way to reduce MQSeries overhead logging to the system log queue is to restrict the types of messages that are sent to the log. This can be done using the queue manager's log and trace settings accessible via MQMT option 1.1, and **PF10**.

For example, you may choose to suppress informational and warning messages and have only error and critical messages sent to the log.

Tuning a system is almost always a matter of compromises.

10.1.3 MQSeries and the event queues

Generating instrumentation event messages, which are written to the event queues, can create considerable overhead. You should not activate instrumentation events for your queue manager, or its channels and queues, unless the information is important and you plan to process the event messages.

Processing event messages is normally handled by an administrative task that acts on event messages as they are generated. The administrative task can be local or remote to the queue manager. That is, the event queues can be remote. Even if event messages are processed on a remote system, the overhead to produce them remains.

Only activate instrumentation events if they are needed.

10.2 CICS performance

MQSeries for VSE/ESA and applications that use it operate as transactions and programs in a CICS/VSE or CICS TS environment. As such, they rely on the tuning and optimization of that CICS system. See either the *CICS for VSE/ESA Performance Guide*, SC33-0703 or *CICS Transaction Server for VSE/ESA Performance Guide*, SC33-1667 for information about optimizing your CICS system.

In most cases, all advice about tuning a CICS environment also applies to MQSeries for VSE. You should especially look at performance reports (CICS statistics or reports created by vendor software) in order to discover and eliminate potential bottlenecks.

10.2.1 MQSeries impact on CICS

The impact that MQSeries has on your CICS system can be measured in one sense by the transactions and programs that run during normal MQSeries operation. This section describes the common transactions and programs that you can expect to see running under CICS while your MQSeries system is active.

During normal operation, one MQ transaction is always started (and can be in a suspended or active state) as shown in Table 10-1.

Table 10-1 MQ transaction always started

Transaction	Program Name	Function
MQSM	MQPSCHK	MQ system housekeeping and monitoring for terminated MQ applications

A few others may also be started during normal operation as shown in Table 10-2.

Table 10-2 Other MQ transactions

Transaction	Program Name	Function
MQER	MQPERR	MQ message formatter for SYSTEM.LOG. Only active when there are messages to process.
MQTL	MQPTCPLN	TCP/IP listener program. Always active if your MQSeries system is expected to accept TCP/IP connections from remote sender MCAs in client programs.
MQ02	MQPAIP2	The generic MQ trigger program. Active only after a trigger event. This may be the MQ sender MCA program MQPSEND.
MQ01	MQPRECV/ MQPTCPSV	MQ receiver MCA. Active only when processing inbound MQ requests from a partner MCA or MQ client. The MQPTCPSV (server) program is active in the case of an MQ client.
MQBI	MQBICITK	MQSeries batch interface. The interface is an optional service to support access to MQSeries objects by batch programs.
MQBX	MQBICIRH	MQSeries batch partner transaction. One instance of MQBX is started for each active batch program connection. These transaction can only be started if the batch interface is active.

Transaction	Program Name	Function
MQCS	MQPCSRV	The MQ Command Server. This transaction, if active, monitors the system command queue for PCF command messages.
MQCX	MQPCPRO	The MQ Command Processor. One instance of this transaction is started for each active PCF command.
MQCN	MQPCNSL	The MQ Console Requester. One instance of this transaction is started for each system message sent to the VSE console that requires an operator response.
MQGX	MQPXGET	The MQ Syncpoint Get program. One instance of this transaction is started for each application that issues an MQGET outside its current logical unit of work (that is, using the MQGMO_NO_SYNCPOINT option).
MQPX	MQPXPUT	The MQ Syncpoint Put program. One instance of this transaction is started for each application that issues an MQPUT outside its current logical unit of work (that is, using the MQPMO_NO_SYNCPOINT option).
MQXP	MQPEXPR	The MQ Expire Message program. One instance of this transaction is started for each queue message that is deemed to have expired.
MQIE	MQPIEVT	The MQ Event Processor. One instance of this transaction is started for each instrumentation event generated by the queue manager.
MQRG	MQPVSAM	The MQ VSAM Reorganization program. This transaction is started once for each scheduled automatic reorganization. Only one instance of this transaction can run at a time.
CKBR	CSQCBR00	The MQ CICS Bridge. This is a long-running transaction that provide CICS bridge services.
CKBP	CSQCBP00	The MQ CICS Bridge process. One instance of this transaction is started for each active CICS bridge application.

Note: You are advised not to define your own transactions beginning with "MQ". This is to avoid conflict between customer and MQSeries transaction names. Remember that future enhancements to MQSeries may result in new MQ transactions, so even if a name is not currently used, it may be used in a future release of MQSeries for VSE.

MQ routines invoked by customer MQ applications include those shown in Table 10-3.

Table 10-3 MQ routines invoked by customer MQ applications

Program name	Function
MQPAIP1	High-level MQ queue manager functions.
MQPQUE1	Low-level MQ queue manager I/O functions.

This is not an exhaustive list and does not include the transactions and programs used during MQ initialization and termination or the MQMT master terminal transaction.

10.2.2 CICS data tables

When MQSeries is started, the configuration file is heavily accessed. From your CICS statistics, you may notice that it is typically used in a direct read mode with a very few updates and a few browses. The following is taken from a sample CICS statistics report:

```
*************************************** FILE STATISTICS  ************************************
 FILE      GET   GET UPD   BROWSE    ADD   UPDATE   DELETE  VSAM EXCP REQUESTS  WAIT-ON-STRING
 NAME     REQSTS  REQSTS   REQSTS  REQSTS  REQSTS   REQSTS     DATA      INDEX   TOTAL HIGHEST

DFHCSD        0        0        0       0       0        0        0                  0       0
FILEA         0        0        0       0       0        0        0                  0       0
IESTRFL       1        0        0       0       0        0        1          2       0       0
IESPRB      202        4        0       0       2        0      206          4       0       0
IESCNTL       5        0        0       0       0        0        5          6       0       0
IESROUT       0        0        0       0       0        0        1          1       0       0
INWFILE       0        0        0       0       0        0        0          0       0       0

MQFCNFG    2982        5       49       0       5        0      982       1748       0       0
MQFI001     109       66      109      50      66        0      148         30       0       0
MQFO001       0        0        0       0       0        0        0                  0       0
MQFI002      21        2        2       0       2        0        8          2       0       0
MQFO002      10        2       26       0       2        0       11          2       0       0
MQFI003       6        1        1       0       1        0        5          6       0       0
MQFO003      10        1        1       0       1        0        3          6       0       0
MQFLOG        5        1     1939       1       1        0      980        126       0       0
MQFERR        4        1        0       0       1        0        2          1       0       0
MQFMON       48        1        1       0       1        0       18          6       0       0
```

Figure 10-1 File statistics

In addition, the configuration file is typically a very small file compared to other MQSeries VSAM clusters. Therefore, it is a good candidate to be eligible as a CICS data table. If, based on your own statistics, you feel the MQSeries configuration file should be a CICS data table, you need to modify the entry for MQFCNFG in your DFHFCT (or CSD for CICS TS). The data table, if used, must be specified as type of CICSTABLE and not USERTABLE.

A CICS data table is the facility of having a KSDS VSAM cluster totally, or partially, in storage. With data tables, VSAM is not used to retrieve records (a hashing method is used instead). Application programs do not need to be modified. Integrity functions (dynamic back out and emergency restart) are still operational for updates, because in such cases, the normal VSAM path is used. Of course, CICS data tables are useful whenever accesses are mainly for read.

Unlike CICS 2.3, CICS TS allows you to exploit CICS data tables in browse mode. Since MQSeries does browse the configuration file, the use of data tables is even more significant under CICS TS.

10.3 VSAM performance

When you define VSAM clusters for message queues using IDCAMS, do not forget to add 736 bytes to the user data message for the MQSeries header information. This formula is relevant when specifying the average and maximum VSAM record lengths.

If the VSAM maximum record size is less than maximum message length for a queue associated with a file, MQSeries will split relevant messages into two or more VSAM records. This is time and resource consuming and should be avoided if possible. We recommend that you always specify a maximum record size equal to or larger than your maximum message size (including the header).

Do not underestimate the average record size. It is used to calculate the primary and secondary allocation sizes if you specify the allocation in records. If the average is too small, this may lead to a large number of extents and control intervals.

We recommend you submit an IDCAMS LISTCAT job periodically to verify the number of secondary allocations for your MQSeries clusters. If you notice a high number of allocations, increase your average record size, or your primary and secondary allocation values.

For MQSeries VSAM clusters, you do not need to specify free space for CIs or CAs for two reasons:

- There is no initial loading.
- Unless you have multiple queues in a VSAM cluster, CI or CA splits will never occur since keys are dynamically created based on an ascending sequence. Once again, we do not recommend multiple queues per VSAM cluster.

All considerations for tuning VSAM clusters equally apply to MQSeries files, especially for the number of buffers for data and indexes. When your MQSeries system is operational and has reached the number of daily messages you would normally expect, check your CICS file statistics. You should check:

- The number of WAITs on the buffers.
- The hit read ratio.
- The number of WAITs on strings. If there are any, increase the value in the DFHFCT for the relevant file.
- The number of Data EXCP versus Index EXCP. If the number of Index EXCP is too high, increase the number of index buffers.
- Try to use LSR pools. However, this is acceptable only if your CPU is not saturated and paging activity is low. Spread your files to different LSR pools (the response time should be better and the CPU activity should be lower).
- Do not use the same CI sizes for indexes and data.
- Define very active queues on separate volumes.

Refer also to the *VSE/VSAM User's Guid*e, SC33-6535, Chapter 5, for information about optimizing VSAM file performance.

10.4 Network performance

Tuning the MQSeries application from a network point of view is the same as tuning any other CICS application. This is also true for applications using VTAM, which is relevant to MQSeries APPC (LU 6.2) connections. Many documents exist for tuning VTAM that describe all parameters related to performance issues (pacing, buffering, sessions, log mode, and so forth). Therefore, we do not want to repeat such documentation in this book.

In this section, we focus on TCP/IP for VSE environments, which may be preferred for MQSeries cross-platform communication.

10.4.1 TCP/IP performance

MQSeries uses the TCP/IP protocol for data transmission. Therefore, every kind of TCP/IP tuning automatically has an effect on MQSeries and the resources it uses. There are five areas of immediate importance:

- Inbound traffic
- Outbound traffic
- Slow connections (for example, WAN connections based on 64-kbit or 128-kbit ISDN telephone lines)
- Storage allocations
- General items

Inbound traffic

Use large SET WINDOW sizes (<= 65535). TCP/IP considers the data item a contiguous byte data stream. The receiver propagates a WINDOW size; the sender can fill up the WINDOW buffer before having to wait for acknowledgments.

Define the TCP/IP MAX_SEGMENT size at its maximum value of 32684 bytes. This also reduces network traffic.

Outbound traffic

Be sure you have defined an appropriate MTU size for your network connection. This limits the number of bytes the IP layer can put onto the network with a single IP frame. Generally, larger values are better. However, the MTU value chosen must match your network characteristics. The network participant with the lowest MTU size will in effect limit the MTU. If you have gateways with MTU values smaller than yours, they will have to segment the data into chunks of only 572 bytes. At the receiver, additional buffer reassembly effort is necessary, affecting network traffic, CPU usage, retransmissions, and I/O path lengths.

Slow connections

TCP/IP dynamically tries to determine whether it should expect a data acknowledgment at a certain time. If it “feels” an ACK being delayed, it assumes the worst case (data loss) and retransmits the data. It uses the value defined by RETRANSMIT as a starting point for its recalculation.

It turns out that the retransmission algorithm has its limitations with slow network connections, eventually flooding the network with superfluous retransmissions. In this case, you can try to set RETRANSMIT to a higher start value, or you can set it to something you consider appropriate and set `FIXED_RETRANS=ON` to have TCP/IP use the fixed retransmission time-out value and suspend its dynamic adaption to it.

Storage allocations

Depending on the number of servers you have defined, and the number of applications running in socket mode (for example, MQSeries), you must have enough space in your virtual storage to accommodate all buffer requirements.

In addition, because TCP/IP may handle unsupported TP devices, it uses private CCW translation routines. That is, before issuing I/Os, it has to fix permanently, in real storage, buffers, control blocks, and CCW chains. Therefore, it may need a large real storage for those PFIXs. We were able to run a few figures with only 200 K, but in more complex environments, we had to specify up to 900 K. We recommend you start with 500 K and see whether this is enough (otherwise, you receive console messages in cases of PFIX shortages). You specify this size with the SETPFIX JCL command in your TCP/IP startup job:

```
// SETPFIX LIMIT=500K
```

Specifying a value that is too high is harmless because the amount is not “reserved” from the real storage. You can monitor usage using the MAP REAL operator command.

General considerations

Be sure you have the appropriate settings, for example, REDISPATCH, DISPATCH_TIME, or ALL_BOUND. It is best not to alter the supplied defaults.

If you are extensively using Telnet 3270 servers, define two TCP/IP partitions, one specially dedicated to TN3270 with a priority higher than CICS and one for other applications (for example, MQSeries). This should provide users with regular terminal response times.

Finally, you should ensure your TCP/IP partner host’s TCP/IP definitions match your own as best as possible, be it workstation, mid-range, or mainframe. If you are operating inbound most of the time, the remote sender needs to send the data efficiently in order to allow you to receive data efficiently. This also applies to the other options such as MTU.

11

Problem determination

This chapter assumes that you have correctly installed MQSeries for VSE as described in Chapter 1, “Installation” on page 1.

This chapter covers common problems that customers have encountered while installing and running MQSeries.

11.1 Installation pitfalls

Some common pitfalls occur after MQSeries for VSE has been installed from the MSHP install tape.

11.1.1 VSE release level and software stack

MQSeries for VSE runs as a subsystem within a CICS/VSE or CICS TS environment. Because of this, it relies on a software stack for its communications and application support functions. For example, the COBOL, C, and PL/I language support is performed by Language Environment/VSE, the intersystem communications support is performed by VTAM or TCP/IP, and the file system support is performed by CICS and VSAM.

Note: Because of the dependence of MQSeries for VSE on the software stack, it is extremely important that the stack be at the correct release and service level.

Refer to 1.1, "MQSeries prerequisite software levels" on page 2 for details.

11.1.2 Upgrading from MQSeries 1.4

There are certain areas to be aware of when migrating from MQSeries 1.4 to MQSeries 2.1. For example:

- V2.1 of MQSeries for VSE has a series of new and updated console messages. One message that existing users of MQSeries for VSE/ESA V1.4 find alarming is as follows:

  ```
  MQI0035I - MQSeries for VSE/ESA licensed support for 0007 clients
  ```

  This message is for documentation purposes only. The system does not perform any checking on the actual number of clients attached.

- A new configuration file (MQFCNFG) is required for MQSeries for VSE.

  The MQFCNFG file used with MQSeries for VSE/ESA V1.4 is not compatible with V2.1 and a new version of this file must be created. This can be done using the supplied JCL (MQJCNFG.Z) and then populated either manually or using the MQJMIGR1.Z and MQJMIGR2.Z sample JCL files to migrate your existing V1.4 MQFCNFG file. This is documented in 1.7, "Migrating from MQSeries 1.4 to MQSeries 2.1" on page 21.

11.1.3 Prerequisite CICS groups

To run successfully, MQSeries requires several CICS groups to be installed. MQSeries itself is defined in a CICS group called MQM. This contains the required definitions for the MQSeries programs, transactions, mapsets, and so on. In addition to this, there are several others that are required:

- V2.1 of MQSeries for VSE uses the EXEC CICS WRITE OPERATOR command to put messages to the VSE console. This requires that the CICS DFHEXEC group is installed.
- Language Environment, COBOL, and C runtime support. The CICS group name required for this is CEE, but the definitions are split between several files, one file for each component. This is covered in more detail in "Language Environment/VSE CSD definitions" on page 15.
- If the Language Environment debug tool is required, a further group, EQA, must be installed. This is found in EQACCSD.A.

11.1.4 TCP/IP modules and libraries

CSI International customers with TCP/IP should note that, starting September 1999, CSI International ships the Language Environment socket interface module, $EDCTCPV. This is the interface that MQSeries uses for all its TCP/IP calls. The module is shipped in their default library PRD2.TCPIP. This library must be placed in the LIBDEF statement before the PRD1.BASE library.

11.2 Application program problems

If you are experiencing problems in your application programs, you should read *MQSeries for VSE System Management Guide*, GC34-5364, Chapter 6. This provides excellent first steps in problem analysis.

The problems that occur can be broadly categorized as follows:

- Application problems
- MQSeries problems
- CICS problems
- Communication problems

To make things more complicated, some of these may actually be dependent on each other. For example, an application may issue an MQI call that may issue CICS commands for communication functions.

Simplify the problem by breaking it down. Do not attempt to resolve a complex application problem in one large chunk. It is best to view these large and complex problems as a series of smaller problems.

For example, if you had an application that receives messages from a remote queue manager onto a triggered local queue where the triggered application reads the local queue and responds back to a second remote queue manager, the problem may be broken down as follows:

1. Inbound message receipt
2. Triggering of a local application
3. Local application execution
4. Outbound message transmission

Each of these steps should be tested and verified in isolation. When each step is confirmed, steps can be tested in combination. For example, test steps 1 and 2, followed by testing steps 1, 2, and 3, and so on. This way, your testing is more manageable.

There are also various means of debugging your applications programs. The following sections deal with some of these options.

11.2.1 Language Environment source language debugger

The Language Environment debug tool is one of the easiest ways to debug code in either C, COBOL, or PL/I. The user can step through the code line by line, set breakpoints, examine and modify data areas, and use the application's own variable names. For more information about its capabilities, refer to the *LE V1R4 Debugging Guide and Run-Time Messages*, SC33-6681.

Before attempting to start using the Language Environment debug tool, ensure that you have installed the appropriate CICS groups documented in 11.1.3, "Prerequisite CICS groups" on page 196 and read the information in *Debug Tool for VSE/ESA Installation and Customization Guide*, SC26-8798.

All debug tools make use of large amounts of virtual storage. We ran our CICS TS system in partition F6 by modifying the ALLOC.PROC file in IJSYSRS.SYSLIB with the following lines:

```
ALLOC F6=40M     CICS
SIZE F6=5M
```

This allowed us sufficient memory resource to run our development and test systems and then compile and debug the TTPTST2 supplied program with the Language Environment debug tool.

See the *VSE/ESA Guide to System Functions*, SC33-6611 for more information about allocating space for VSE partitions.

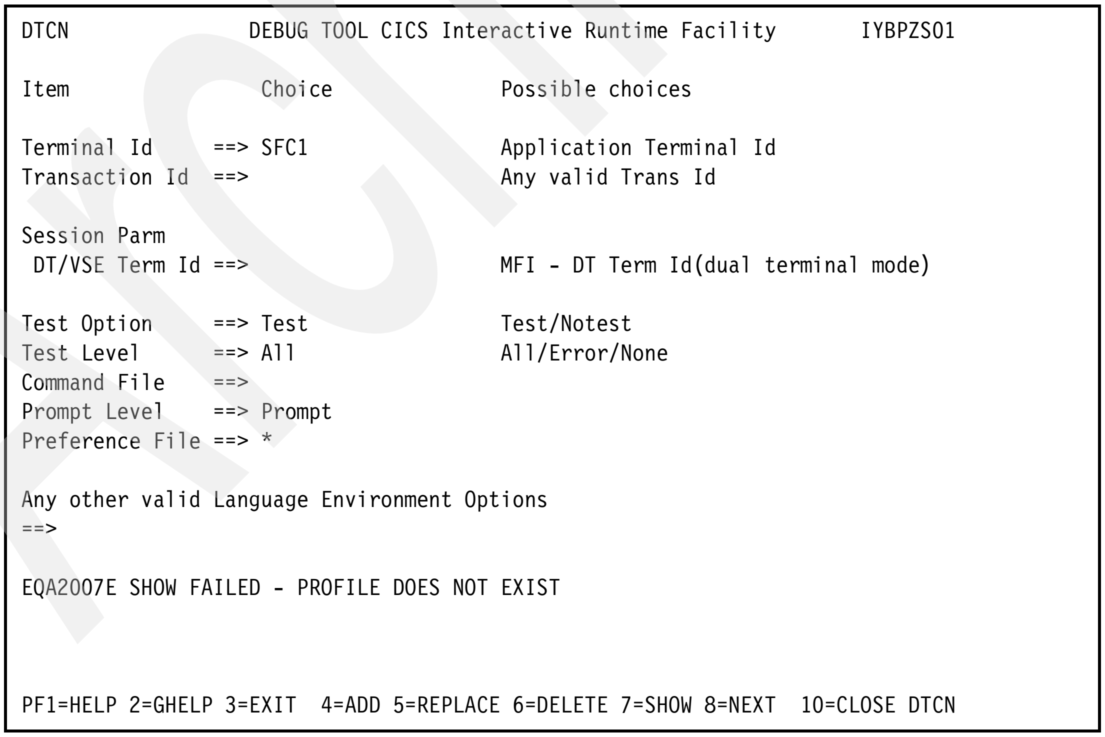
A production system could be tailored to have less storage requirements.

To build an application (in this example, a COBOL program) that can be executed using the debug tool, you should modify the compile JCL to include the DEBUG options for the compiler. For example, the sample JCL in “JCL to build a CICS COBOL program” on page 298 could be modified to use the following options:

```
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM=’’
  PROCESS LIB,APOST,NOADV,RENT,NOXREF,BUF(4096),NODYNAM,TEST(ALL,SYM)
  PROCESS NOSEQ,TRUNC(OPT),DATA(31)
```

To start the Language Environment debug tool under CICS or TS, use the DTCN transaction.

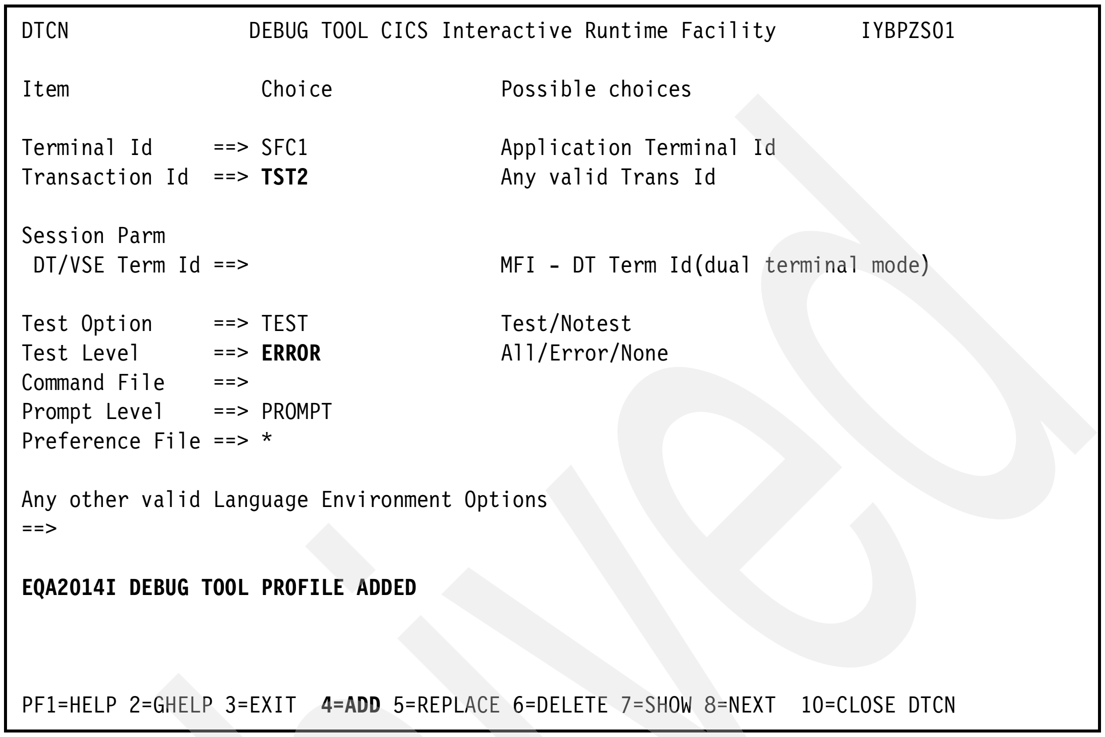
The initial screen display will appear as illustrated in Figure 11-1.



```
DTCN              DEBUG TOOL CICS Interactive Runtime Facility       IYBPZS01

Item               Choice               Possible choices

Terminal Id     ==> SFC1                Application Terminal Id
Transaction Id  ==>                     Any valid Trans Id

Session Parm
 DT/VSE Term Id ==>                     MFI - DT Term Id(dual terminal mode)

Test Option     ==> Test                Test/Notest
Test Level      ==> All                 All/Error/None
Command File    ==>
Prompt Level    ==> Prompt
Preference File ==> *

Any other valid Language Environment Options
==>

EQA2007E SHOW FAILED - PROFILE DOES NOT EXIST



PF1=HELP 2=GHELP 3=EXIT  4=ADD 5=REPLACE 6=DELETE 7=SHOW 8=NEXT  10=CLOSE DTCN
```

Figure 11-1 Language Environment DEBUG TOOL DTCN screen

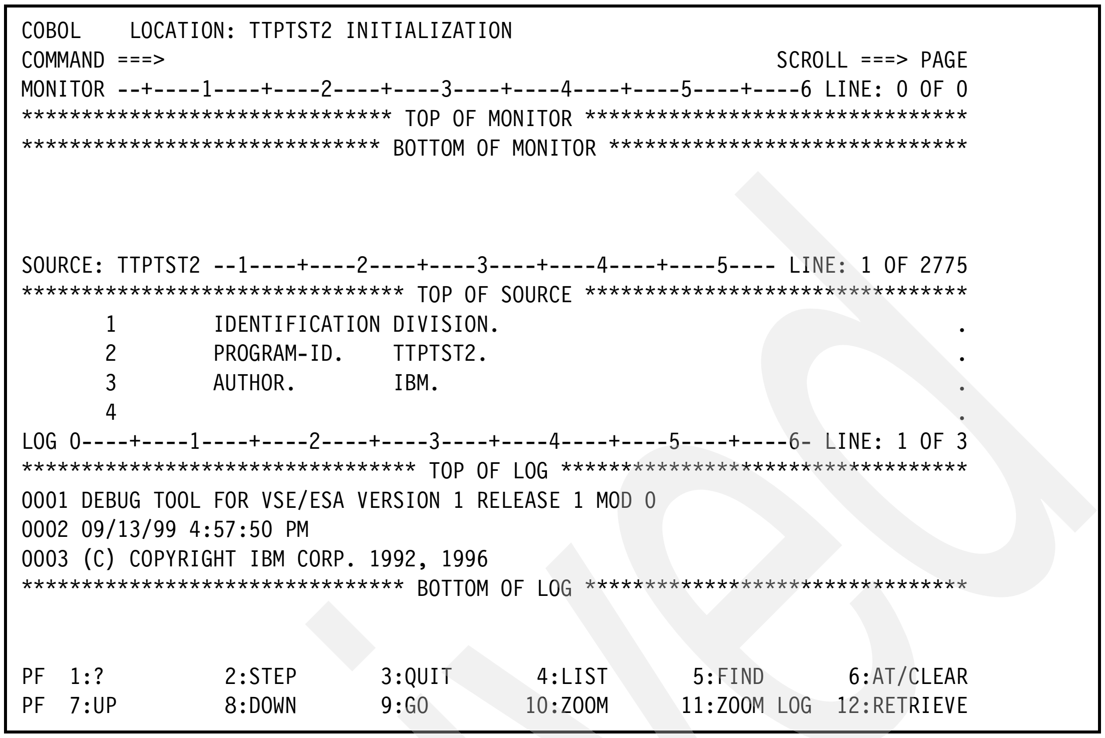
Then add the name of the transaction that is to be debugged using the **PF4** key to the list that DTCN maintains. This is illustrated in Figure 11-2.



```
DTCN              DEBUG TOOL CICS Interactive Runtime Facility       IYBPZS01

Item                Choice              Possible choices

Terminal Id     ==> SFC1                Application Terminal Id
Transaction Id  ==> TST2                Any valid Trans Id

Session Parm
 DT/VSE Term Id ==>                     MFI - DT Term Id(dual terminal mode)

Test Option     ==> TEST                Test/Notest
Test Level      ==> ERROR               All/Error/None
Command File    ==>
Prompt Level    ==> PROMPT
Preference File ==> *

Any other valid Language Environment Options
==>

EQA2014I DEBUG TOOL PROFILE ADDED



PF1=HELP 2=GHELP 3=EXIT  4=ADD 5=REPLACE 6=DELETE 7=SHOW 8=NEXT  10=CLOSE DTCN
```

Figure 11-2 Specifying the transaction name to Language Environment DEBUG TOOL DTCN

When the transaction name is invoked, the Language Environment debug tool will take control and display the source of the program. The example in Figure 11-3 on page 200 shows the screen displayed when the sample transaction TST2, which has been compiled using debug options, is invoked.

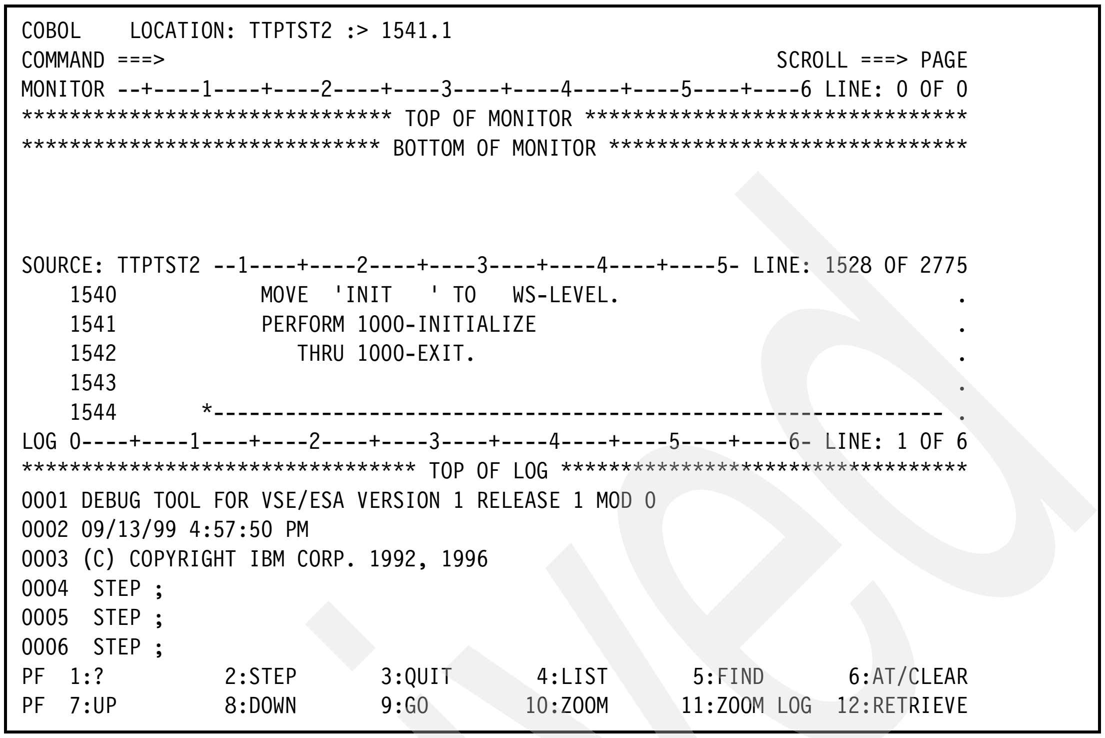


```
COBOL    LOCATION: TTPTST2 INITIALIZATION
COMMAND ===>                                                   SCROLL ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 0 OF 0
******************************* TOP OF MONITOR ********************************
****************************** BOTTOM OF MONITOR ******************************



SOURCE: TTPTST2 --1----+----2----+----3----+----4----+----5---- LINE: 1 OF 2775
********************************* TOP OF SOURCE ********************************
      1       IDENTIFICATION DIVISION.                                        .
      2       PROGRAM-ID.    TTPTST2.                                         .
      3       AUTHOR.        IBM.                                             .
      4                                                                       .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 3
********************************* TOP OF LOG **********************************
0001 DEBUG TOOL FOR VSE/ESA VERSION 1 RELEASE 1 MOD 0
0002 09/13/99 4:57:50 PM
0003 (C) COPYRIGHT IBM CORP. 1992, 1996
******************************** BOTTOM OF LOG ********************************


PF  1:?         2:STEP      3:QUIT      4:LIST      5:FIND      6:AT/CLEAR
PF  7:UP        8:DOWN      9:GO       10:ZOOM     11:ZOOM LOG 12:RETRIEVE
```

Figure 11-3 Initial Language Environment DEBUG TOOL screen for TST2 transaction

After the program is invoked, the user can then step through the code line-by-line using the **PF2** key.

If there are called subroutines, or other CICS routines that are linked from the main code, these also need to be compiled with the debug compiler option.



```
COBOL    LOCATION: TTPTST2 :> 1541.1
COMMAND ===>                                                   SCROLL ===> PAGE
MONITOR --+----1----+----2----+----3----+----4----+----5----+----6 LINE: 0 OF 0
******************************* TOP OF MONITOR ********************************
****************************** BOTTOM OF MONITOR ******************************



SOURCE: TTPTST2 --1----+----2----+----3----+----4----+----5- LINE: 1528 OF 2775
    1540            MOVE  'INIT   ' TO   WS-LEVEL.                             .
    1541            PERFORM 1000-INITIALIZE                                    .
    1542               THRU 1000-EXIT.                                         .
    1543                                                                       .
    1544       *-------------------------------------------------------------- .
LOG 0----+----1----+----2----+----3----+----4----+----5----+----6- LINE: 1 OF 6
********************************* TOP OF LOG **********************************
0001 DEBUG TOOL FOR VSE/ESA VERSION 1 RELEASE 1 MOD 0
0002 09/13/99 4:57:50 PM
0003 (C) COPYRIGHT IBM CORP. 1992, 1996
0004  STEP ;
0005  STEP ;
0006  STEP ;
PF  1:?         2:STEP      3:QUIT      4:LIST      5:FIND      6:AT/CLEAR
PF  7:UP        8:DOWN      9:GO       10:ZOOM     11:ZOOM LOG  12:RETRIEVE
```

Figure 11-4 Step through screen for Language Environment DEBUG TOOL DTCN

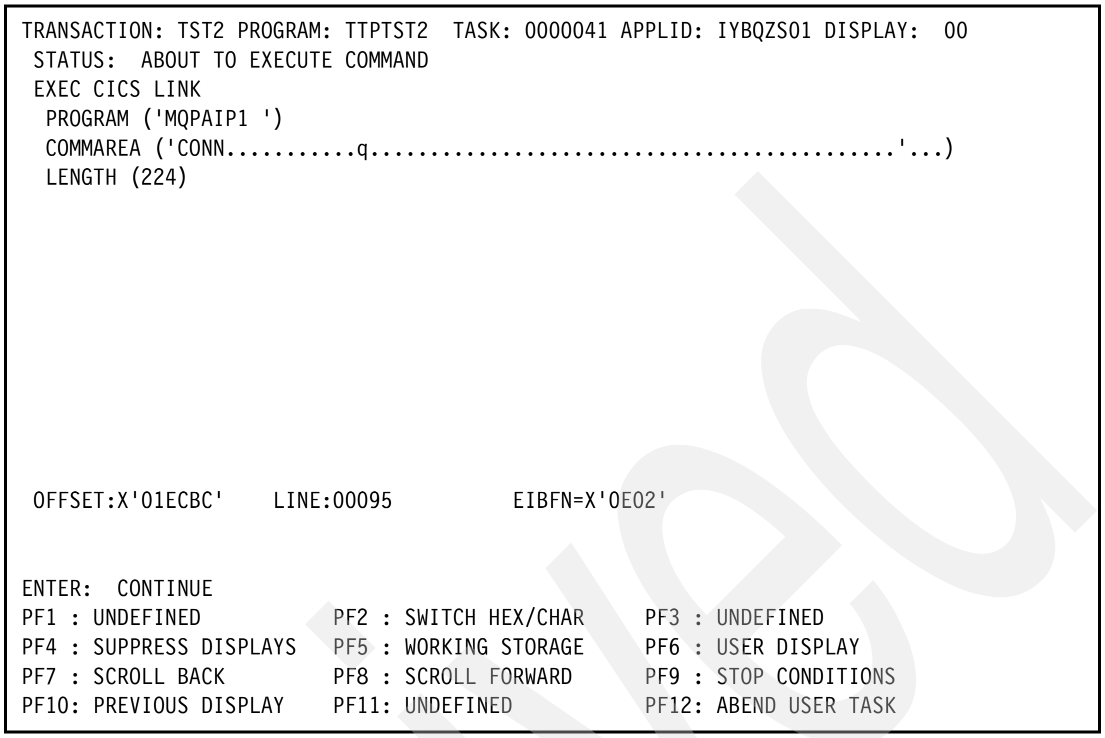
The Language Environment debug tool provides the lowest level of granularity for accessing source code when debugging an application.

11.2.2 CICS Execution Diagnostic Facility (CEDF)

As with any CICS transaction, a customer's application can be invoked using CEDF. CEDF allows you to see each CICS command before and after its execution together with the CICS EIB fields and the program's working storage. CEDF is fully documented in *CICS for VSE/ESA CICS-Supplied Transactions*, SC33-0710, or *CICS TS CICS-Supplied Transactions*, SC33-1655.

CEDF is started like any other transaction. The next transaction that is started after CEDF is executed under CEDF control. CEDF pauses before the program begins execution and at every EXEC CICS call.

Figure 11-5 on page 202 illustrates an example of an CEDF screen.



```
TRANSACTION: TST2 PROGRAM: TTPTST2  TASK: 0000041 APPLID: IYBQZS01 DISPLAY:  00
 STATUS:  ABOUT TO EXECUTE COMMAND
 EXEC CICS LINK
  PROGRAM ('MQPAIP1 ')
  COMMAREA ('CONN..........q.........................................'...)
  LENGTH (224)









 OFFSET:X'01ECBC'    LINE:00095         EIBFN=X'0E02'


ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR     PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE     PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED           PF12: ABEND USER TASK
```

Figure 11-5 CEDF screen during program execution

In this example, the supplied test transaction TST2 has been started under the control of CEDF. We have stepped through the first EXEC CICS commands and we can see control about to be passed to one of the MQSeries queue manager routines, MQPAIP1.

If we continued this debugging session, we would see the sequence of EXEC CICS commands as they are executed. This would assist us in isolating any CICS problems that may occur in our test transaction.

11.2.3 CICS auxiliary trace

For some application problems, it may be worth testing using the CICS trace facility. This gives the lowest level of granularity for tracing. All of the CICS functions from the application and MQSeries are detailed as follows:

- CICS V2.3 uses the CEMT SET AUXTRACE ON OPEN command to start tracing and the DFHTUP utility for formatting the trace output. See “CICS trace format JCL” on page 302.
- CICS TS V1.1 uses CEMT SET AUX STA or the screen-based CETR transaction. See “JCL to format CICS TS V1.1 traces” on page 302.

The formatted output is very different between these two versions.

This type of trace output is usually only required when requested by IBM support personnel. Remember to set all MQ/VSE trace flags to `Y` before starting the trace (refer to 2.1.2, “Log and Trace settings” on page 36). However, it is still useful for determining the results of your application’s EXEC CICS calls.

11.2.4 The MQI monitor

The MQI monitor allows a user to trace MQI calls and their return code values as they are executed by MQSeries and application programs. The details of how to toggle the MQI monitor on and off are in the *MQSeries for VSE System Management Guide*, GC34-5364, Chapter 6, and 3.1, “Start/stop queues” on page 60.

In addition, there are some restrictions that users should be aware of:

- The monitor traces MQI calls for all MQSeries applications. It is not possible to switch on the monitoring for a single application. Therefore, it is better to only run one application while using the MQI monitor.
- The MQI monitor does not trace the results of the MQDISC command.

Table 11-1 describes the contents of an MQI monitor message.

Table 11-1 MQI monitor message description

MQ API Monitor record field name	Length	Dec	Hex.	Range of values
MQ-MON-TRAN-ID	4	0	X’00’	
MQ-MON-TERM-ID	4	4	X’04’	
MQ-MON-CICS-TASKN	4	8	X’08’	
MQ-MON-SYSTEM-NUM	2	24	X’18’	“01”
MQ-MON-FUNCTION	4	40	X’28’	CONN, MCCO, OPEN, PUT, INQ, SET, GET, CLOS
MQ-MON-START-ABSTIME	4	48	X’30’	
MQ-MON-END-ABSTIME	4	56	X’38’	
MQ-MON-CCODE	4	64	X’40’	
MQ-MON-RCODE	4	68	X’44’	
MQ-MON-QM-NAME	48	84	X’54’	
MQ-MON-Q-NAME	48	132	X’84’	
MQ-MON-RESOLVED-Q	48	180	X’B4’	

11.3 Common problems

Once a system has been installed, and the initial problems encountered during development and testing have been resolved, most customers run their production systems without problems. Even so, problems can occur on a system that has been stable for some time. If problems begin to occur in a system that has previously been running correctly, you should look for possible changes that may have caused the problem. To this end, you should consider the following:

- Service incorrectly applied

  A common problem that can affect the entire operation of your MQSeries system is incorrectly applied service. When you apply an MQSeries PTF, there are sometimes additional instructions in the PTF cover letter and also as comments in the PTF JCL. These should be read and carried out carefully.

Typically, these might include one of the following:

- Rerun MQJSETUP.Z and the MQSU transaction

  Sometimes PTFs include a change to the MQSeries configuration file. This is shipped in part SYSIN.Z. The MQJSETUP.Z job reloads this file, along with any updates. Once loaded, the MQSU transaction applies the new version of the file to the MQSeries configuration file. If you do not follow these instructions, your configuration can be out of step, causing unpredictable results. You should check that the number of records loaded as displayed upon completion of the MQSU transaction matches the number loaded as printed in the MQJSETUP job output.

- Modify your CSD using MQJCSD.Z or MQJCSD24.Z

  Sometimes a PTF will involve changes to the CICS CSD. This may be new transactions, programs or files, or changes to existing entries. If you do not apply the modifications provided in MQJCSD.Z (or MQJCSD24.Z for CICS TS), you may get unpredictable results.

- Modify your DCT using MQCICDCT.Z

  Rarely, a PTF may involve modifications to MQSeries DCT entries or new entries. These will be provided in part MQCICDCT.Z. When instructed, you must rebuild your CICS DCT to incorporate these changes, or risk unpredictable results.

► Application failure to use MQCLOSE or MQDISC

When writing your MQ applications, you must remember to explicitly close open queues using the MQCLOSE MQI call, and remember to disconnect from the queue manager using the MQDISC MQI call.

An application that issues an MQCONN to connect to the queue manager, should always issue an MQDISC to disconnect when the application ends or no longer needs access to MQSeries. Similarly, an application that opens a queue using MQOPEN, should always use MQCLOSE when the queue is no longer needed.

If an application does not follow these principles, MQSeries will detect that connection and queue handles have been left open when the task has ended, and issue warning messages to the system log. It also means MQSeries resources are held unnecessarily until MQSeries identifies the obsolete handles.

► Changes or increase in workload or work patterns

Has the workload on the system increased in some way or the workload on a partner system that is transmitting to the MQSeries for VSE system? This is not always apparent on the partner systems, but you may look for increases in network traffic.

Has the pattern of work changed or the type of data? New data types may have been introduced or the times at which the system is operational may have changed. For example, new night shift work patterns may cause increases in evening or over night workloads.

Increased workloads may require increased disk, processor, or storage resources.

► New hardware installed

New hardware may cause changes in the timing of certain application programs. If an application system relies on a sequence of processing that has changed, this could cause problems.

► New or updated software installed

One of the most common reasons that application systems begin to fail is that changes are made to the installed software.

Fixes to IBM and third party software may have been installed. These may have come from official install media or downloaded from Web sites.

Important: Never apply software from an Internet site directly to your live production system. Always apply to a test system first.

Ensure you have taken adequate backups before installation.

11.4 Known problems

Once you have done your own investigations and established that it is not an easily recognized application problem, check with your IBM support branch. The problem may be known and there may be corrective service available for the problem.

The following is a list of problems that at the time of writing this redbook are known to exist:

- AFCL, AFCP, or AJCN abend during MQSeries execution

  These abends are all symptoms of configuration problems with the CICS system journal. MQSeries for VSE relies on the CICS system journal being correctly defined using buffer values that are suitable for the largest VSAM record that MQSeries will access.

  This is explained in more detail in 1.5.4, “Journal Control Table (JCT) definitions” on page 18 and also in the *CICS for VSE/ESA 2.3 Resource Definition (Macro)*, SC33-0709, and *CICS for VSE/ESA 2.3 Customization Guide*, SC33-0707, or equivalent manuals for CICS TS.

- TP names in channel definitions

  The MQSeries for VSE sender channel definitions created by MQMT option 1.3 require a reference to a TP Name field. This field is a reference to either a transaction program profile in the communications setup of the partner (SNA Server, CM2, Communications Manager, or their equivalent) or a transaction definition (PCT entry) on a CICS partner system where the MQSeries partner uses the MQSeries CICS mover.

  This TP name field must be precisely defined. In the case where the partner resides on a Windows XP, OS/2, or UNIX-based system, it may be case sensitive. In addition, the profile definition on the partner must reference a valid program name in a valid directory.

  This type of problem usually requires a communications trace to see the SNA sense data flowing between the systems.

  Ensure that the same character case is used on each partner for the TP name. (mq01/MQ01/amqcrs6a/AMQCRS6A).

- Stopping and starting TCP/IP while MQSeries is active

  If you shut down TCP/IP while MQSeries is active, the TCP/IP Listener transaction (MQTL) will abnormally terminate, along with any active TCP/IP channels. This is because these use TCP/IP socket services that are stopped when TCP/IP is shut down.

  Similarly, if TCP/IP is not active when MQSeries is started, the TCP/IP Listener program will not start. This means TCP/IP channels will not be unavailable until TCP/IP is started and the Listener program is subsequently restarted.

- MQSeries and CICS Storage Protection

  CICS TS provides the option to run with the storage protection feature active. This is done by setting the CICS SIT parameter, STGPROT, to YES (that is, `STGPROT=YES`).

MQSeries programs, defined in the CSD, can have an ExecKey of either CICS or USER. When storage protection is active, some MQSeries programs must run with `ExecKey=CICS`, otherwise they will abend.

Consequently, if you plan to run your CICS partition with storage protection enabled, you must ensure the appropriate MQSeries programs are defined with ExecKey=CICS. For a definitive list of programs that need ExecKey=CICS, review the MQJCSD24.Z installation JCL. MQJCSD24.Z contains CSD definitions for all MQSeries programs. Make sure you have the latest MQSeries maintenance applied to your system to ensure you have the latest ExecKey requirements.

11.5 Assistance with problems

Whenever you encounter a problem that you have fully researched but cannot resolve, it is worth seeking assistance from other sources. Possible sources include IBM support centers, Web sites, user groups, and so on.

Ensure that you have carefully researched the problem.

Check with documentation to ensure that:

- There are no documented restrictions that apply to your platform.

  For example, certain fields and options are not supported on the MQSeries for VSE platform.
- The MQSeries product works the way in which your application assumes.

  Some users assume certain behavior from the MQI that does not conform to the documentation.

When contacting IBM support centers, ensure that you have adequate documentation for the problem. For example:

- Details of the exact error messages and symptoms

  Check the VSE console and the SYSTEM.LOG for messages.

  If required, look on partner systems for messages, for example, in the partner MQSeries amqerrlog files.
- Application return codes

  If the application received an error return code, record it. Perhaps these could be obtained using the MQI monitor.
- A short and accurate description of the application and how it works
- All of the relevant local and remote queues definitions and channel definitions for MQSeries for VSE and, where necessary, the partner systems

  If required, these can be printed on MQSeries for VSE with the MQPUTIL program using the PRINT CONFIG option.

Be aware that problems occur on all software systems. The problem may be within a customer's application or within the system software.

Complete and accurate diagnostic information is essential for resolving problems.

A

Distributed queuing examples

In Chapter 6, “Distributed queuing” on page 101, we examined distributed queuing in terms of the MQSeries objects involved and the interactions between these objects. In this appendix, we provide specific examples of distributed queuing including actual definitions for channels and queues. Our examples include:

- VSE to VSE
- VSE to OS/390
- OS/390 to VSE
- VSE to Windows XP
- Windows XP to VSE
- Windows XP Client to VSE
- VM/CMS Client to VSE

In addition, we examine a simple test for each of these configurations and explain the results.

VSE to VSE distributed example

In this example, we set up distributed queuing between two VSE systems running MQSeries. Specifically, we provide an example of an MQSeries application running on VSE 2.7 putting messages on a remote queue on a VSE 2.6 system using SNA LU 6.2 channels.

The prerequisites for this example are:

- MQSeries for VSE 2.1 correctly installed on both VSE systems
- Valid VTAM connection from VSE 2.7 system to VSE 2.6 system

For the sake of convenience, we refer to the local VSE 2.7 system as VSE1 and the remote VSE 2.6 system as VSE2. The various MQSeries definitions involved in this example can be created in any order; however, we use the following order:

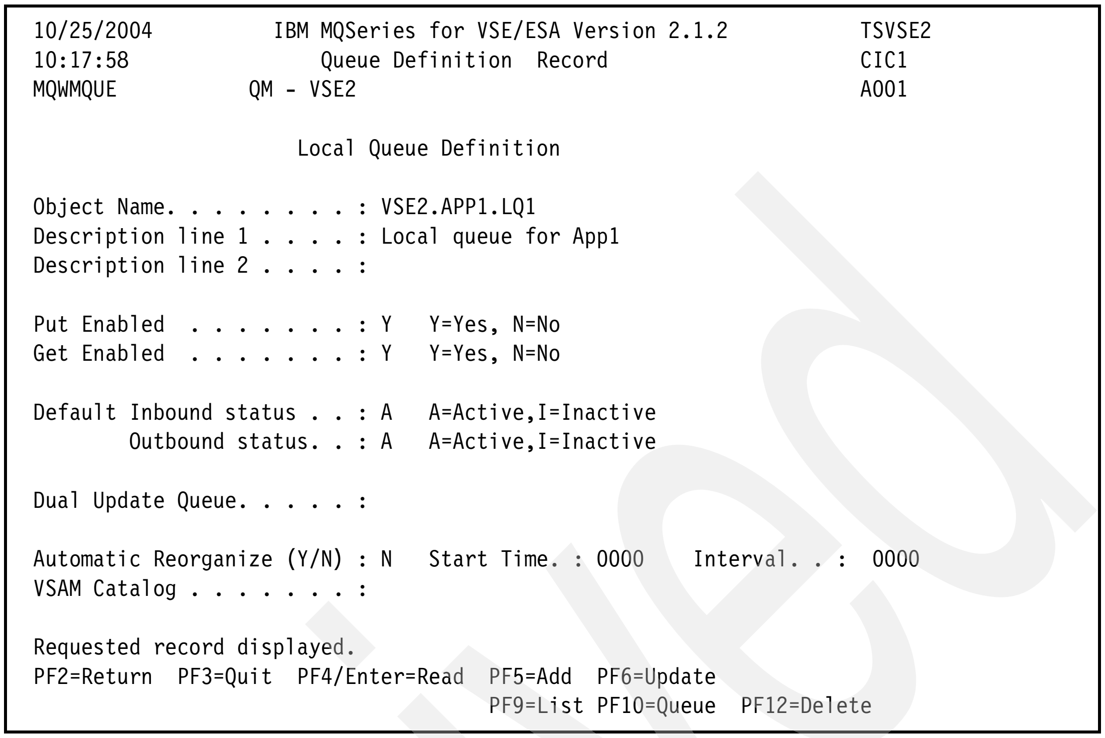
1. Local queue definition
2. Remote queue definition
3. Transmission queue definition
4. Connection definition
5. Session definition
6. Sender channel definition
7. Receiver channel definition

These are sufficient to achieve remote queuing between the two VSE systems.

VSE to VSE local queue definition

First, we define a local queue on the remote VSE2 system. This queue is identified by the remote queue definition on VSE1. In our example, when we put messages on VSE1, it is to this remote physical queue.

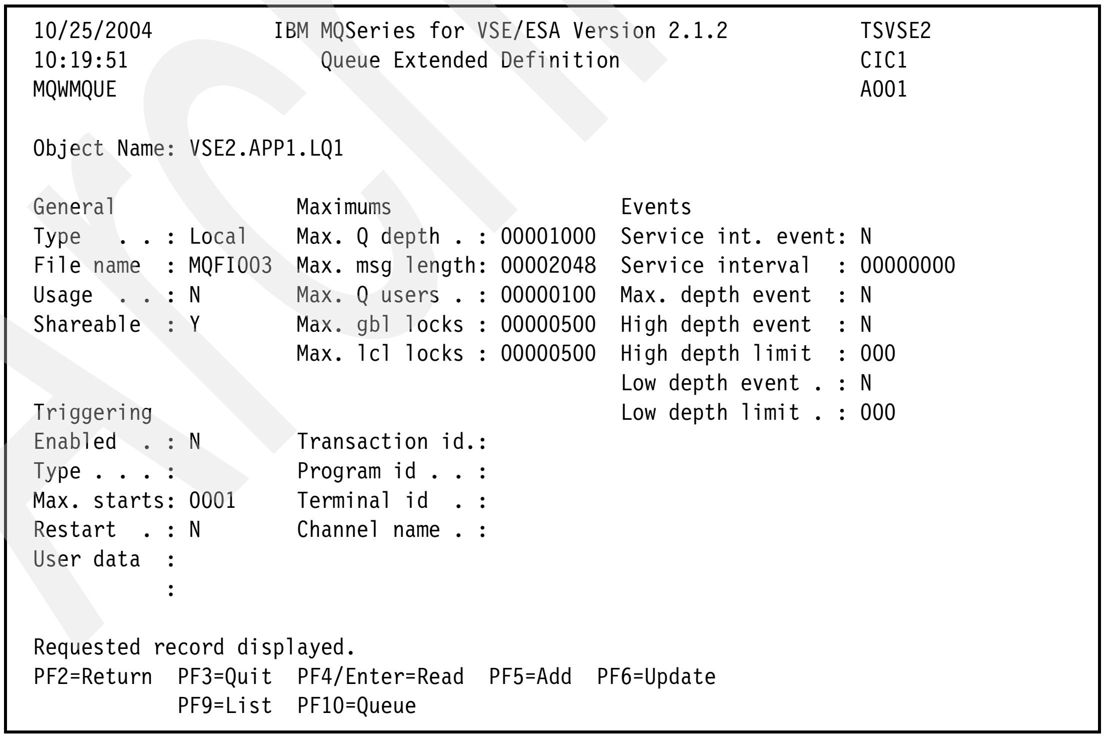
Figure A-1 on page 209 shows the local queue definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSVSE2
10:17:58               Queue Definition  Record                   CIC1
MQWMQUE           QM - VSE2                                       A001

                     Local Queue Definition

Object Name. . . . . . . . : VSE2.APP1.LQ1
Description line 1 . . . . : Local queue for App1
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-1   Local Queue Definition for VSE to VSE distributed queuing

Figure A-2 shows the local queue extended definition we used.



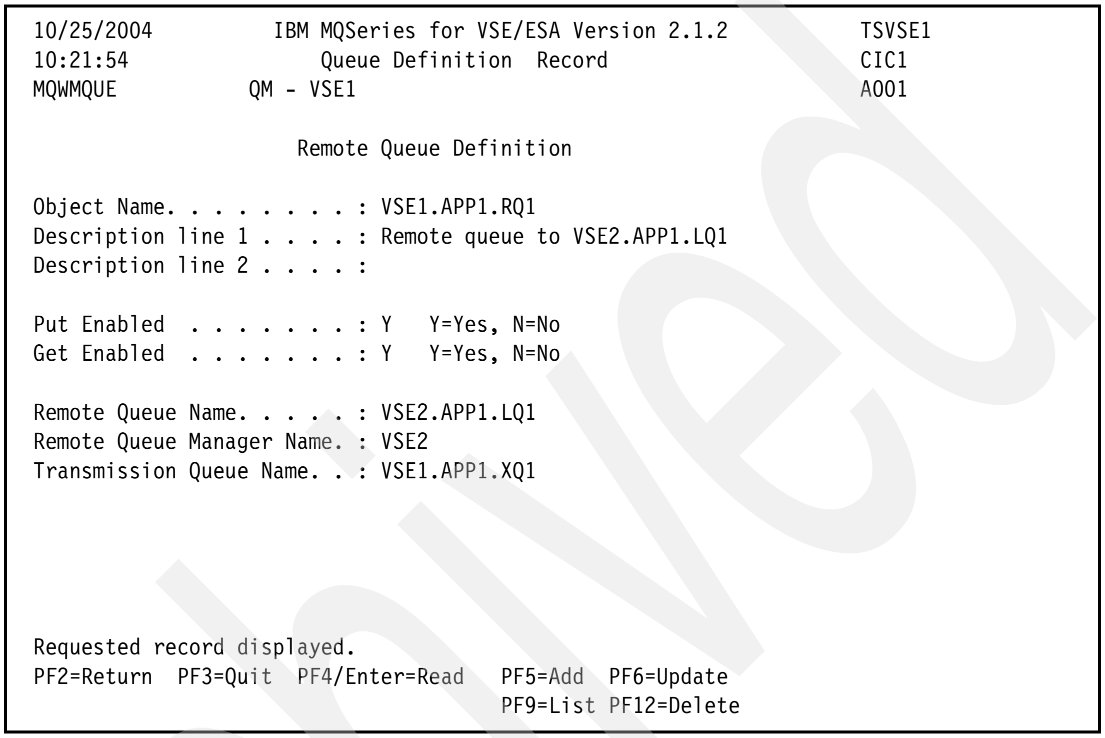
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSVSE2
10:19:51               Queue Extended Definition                  CIC1
MQWMQUE                                                           A001

Object Name: VSE2.APP1.LQ1

General               Maximums                  Events
Type   . . : Local    Max. Q depth . : 00001000  Service int. event: N
File name  : MQFI003  Max. msg length: 00002048  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000500  High depth event  : N
                      Max. lcl locks : 00000500  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : N        Transaction id.:
Type . . . :          Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-2   Local Queue Extended Definition for VSE to VSE

VSE to VSE remote queue definition

We now define a remote queue on the VSE1 system. This identifies the local queue on VSE2 that we just defined.

Figure A-3 shows the remote queue definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2           TSVSE1
10:21:54               Queue Definition  Record                     CIC1
MQWMQUE           QM - VSE1                                         A001

                      Remote Queue Definition

Object Name. . . . . . . . : VSE1.APP1.RQ1
Description line 1 . . . . : Remote queue to VSE2.APP1.LQ1
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Remote Queue Name. . . . . : VSE2.APP1.LQ1
Remote Queue Manager Name. : VSE2
Transmission Queue Name. . : VSE1.APP1.XQ1





Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read   PF5=Add  PF6=Update
                                       PF9=List PF12=Delete
```

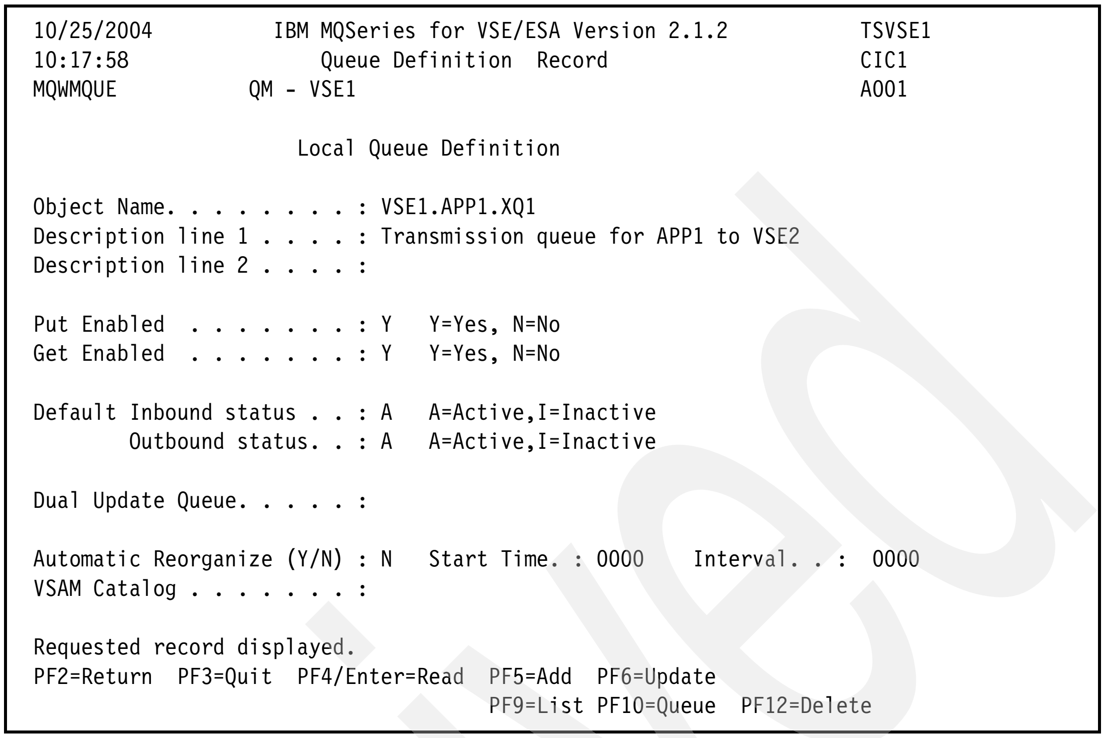
Figure A-3   Remote Queue Definition for VSE to VSE distributed queuing

Note that the remote queue definition identifies a transmission queue. Messages logically written to the remote queue are physically written to the transmission queue and are not logically deleted from the transmission queue until they are successfully received by the remote MQSeries system.

VSE to VSE transmission queue definition

The transmission queue is also defined on the local VSE1 system.

Figure A-4 on page 211 shows the transmission queue definition we used.

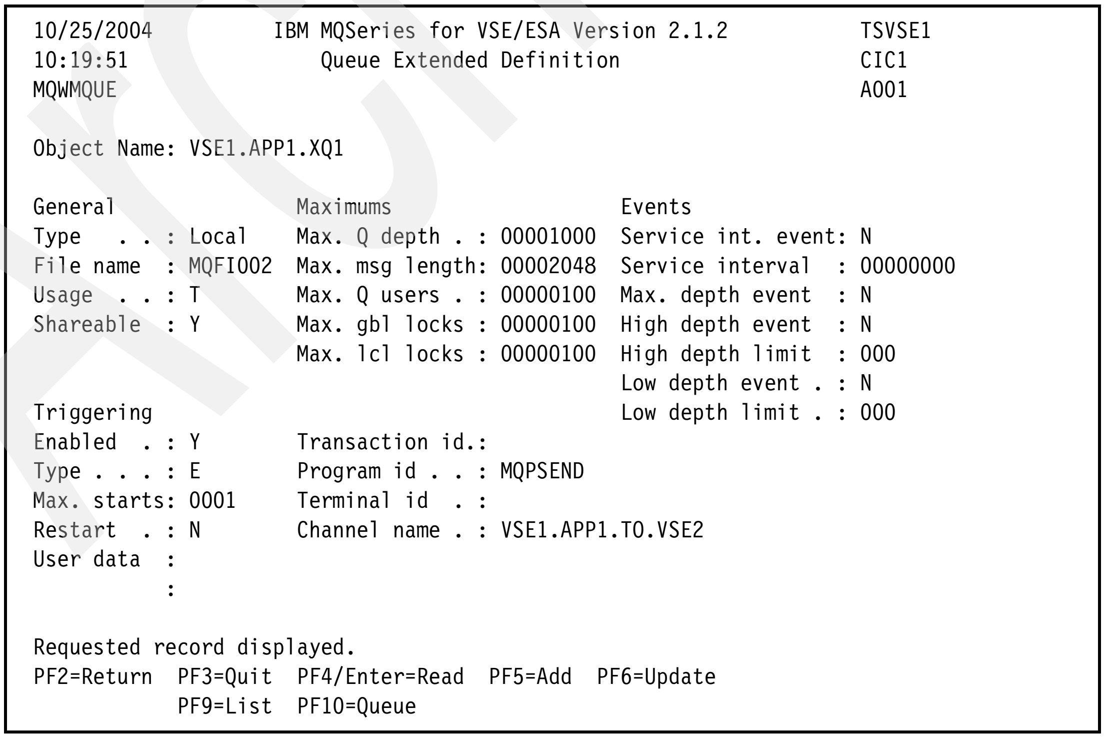


```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:17:58               Queue Definition  Record                    CIC1
MQWMQUE           QM - VSE1                                        A001

                      Local Queue Definition

Object Name. . . . . . . . : VSE1.APP1.XQ1
Description line 1 . . . . : Transmission queue for APP1 to VSE2
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-4 Transmission Queue Definition for VSE to VSE

Note that the transmission queue name matches the transmission queue identified in the remote queue definition.

Figure A-5 shows the transmission queue extended definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: VSE1.APP1.XQ1

General               Maximums                   Events
Type   . . : Local    Max. Q depth . : 00001000  Service int. event: N
File name  : MQFI002  Max. msg length: 00002048  Service interval  : 00000000
Usage  . . : T        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.:
Type . . . : E        Program id . . : MQPSEND
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . : VSE1.APP1.TO.VSE2
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-5 Transmission Queue Extended Definition for VSE to VSE

Note that the transmission queue identifies a channel name. This is the LU 6.2 sender channel that we will define later.

VSE to VSE connection definition

Having defined our queues, we now define the connection and the session definitions we need in CICS for an LU 6.2 connection between VSE1 and VSE2. These definitions are created under CICS on our VSE1 system.

Figure A-6 shows the CONNECTION definition we used.

```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0410
 CEDA  View Connection( VSE2 )
  Connection     : VSE2
  Group          : MQM
  DEscription    : LU62 CONNECTION VSE1 TO VSE2
 CONNECTION IDENTIFIERS
  Netname        : IYBPZS01
  INDsys         :
 REMOTE ATTRIBUTES
  REMOTESYSTem   :
  REMOTEName     :
  REMOTESYSNet   :
 CONNECTION PROPERTIES
  ACcessmethod   : Vtam              Vtam | IRc | INdirect
  PRotocol       : Appc              Appc | Lu61 | Exci
  Conntype       :                   Generic | Specific
  SInglesess     : No                No | Yes
  DAtastream     : User              User | 3270 | SCs | STrfield | Lms
  RECordformat   : U                 U | Vb
  Queuelimit     : No                No | 0-9999
  Maxqtime       : No                No | 0-9999
 OPERATIONAL PROPERTIES
  AUtoconnect    : Yes               No | Yes | All
  INService      : Yes               Yes | No
 SECURITY
  SEcurityname   :
  ATtachsec      : Local             Local | Identify | Verify | Persistent
                                     | Mixidpe
  BINDPassword   :                   PASSWORD NOT SPECIFIED
  BINDSecurity   : No                No | Yes
  Usedfltuser    : No                No | Yes
 RECOVERY
  PSrecovery     : Sysdefault        Sysdefault | None
```

Figure A-6 CICS CONNECTION definition for VSE to VSE

The netname in the CONNECTION definition identifies the APPLID of the CICS system running MQSeries on VSE2. The netname must be known to VTAM.

VSE to VSE session definition

We now define the session definition. Figure A-7 on page 213 shows the SESSION definition we used.

```
OBJECT CHARACTERISTICS                                     CICS RELEASE = 0410
 CEDA  View Sessions( VSE1VSE2 )
  Sessions       : VSE1VSE2
  Group          : MQM
  DEscription    : VSE1 TO VSE2
 SESSION IDENTIFIERS
  Connection     : VSE2
  SESSName       :
  NETnameq       :
  MOdename       :
 SESSION PROPERTIES
  Protocol       : Appc              Appc | Lu61 | Exci
  MAximum        : 012 , 006         0-999
  RECEIVEPfx     :
  RECEIVECount   :                   1-999
  SENDPfx        :
  SENDCount      :                   1-999
  SENDSize       : 04096             1-30720
  RECEIVESize    : 04096             1-30720
  SESSPriority   : 000               0-255
  Transaction    :
 OPERATOR DEFAULTS
  OPERId         :
  OPERPriority   : 000               0-255
  OPERRsl        : 0                                                0-24,..
  OPERSecurity   : 1                                                1-64,..
 PRESET SECURITY
  USERId         :
 OPERATIONAL PROPERTIES
  Autoconnect    : Yes               No | Yes | All
  INservice      :                   No | Yes
  Buildchain     : Yes               Yes | No
  USERArealen    : 000               0-255
  NEPclass       : 000               0-255
 RECOVERY
  RECOvoption    : Sysdefault        Sysdefault | Clearconv | Releasesess
                                     | Uncondrel | None
```

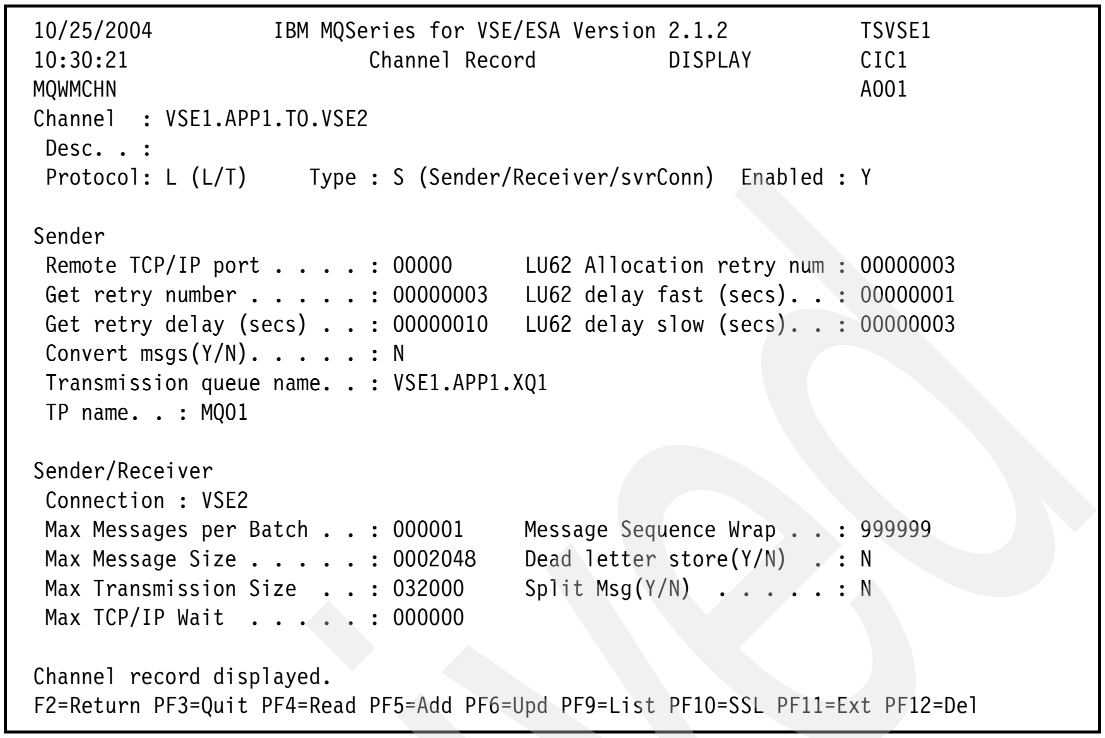
Figure A-7   CICS SESSION definition for VSE to VSE distributed queuing

Note that the connection value in the SESSION definition identifies the name we chose for the connection definition.

VSE to VSE sender channel definition

We can now define our MQSeries channels. The sender channel is defined on our local VSE1 system; the receiver channel is defined on our remote VSE2 system.

Figure A-8 on page 214 shows the LU 6.2 sender channel we used.



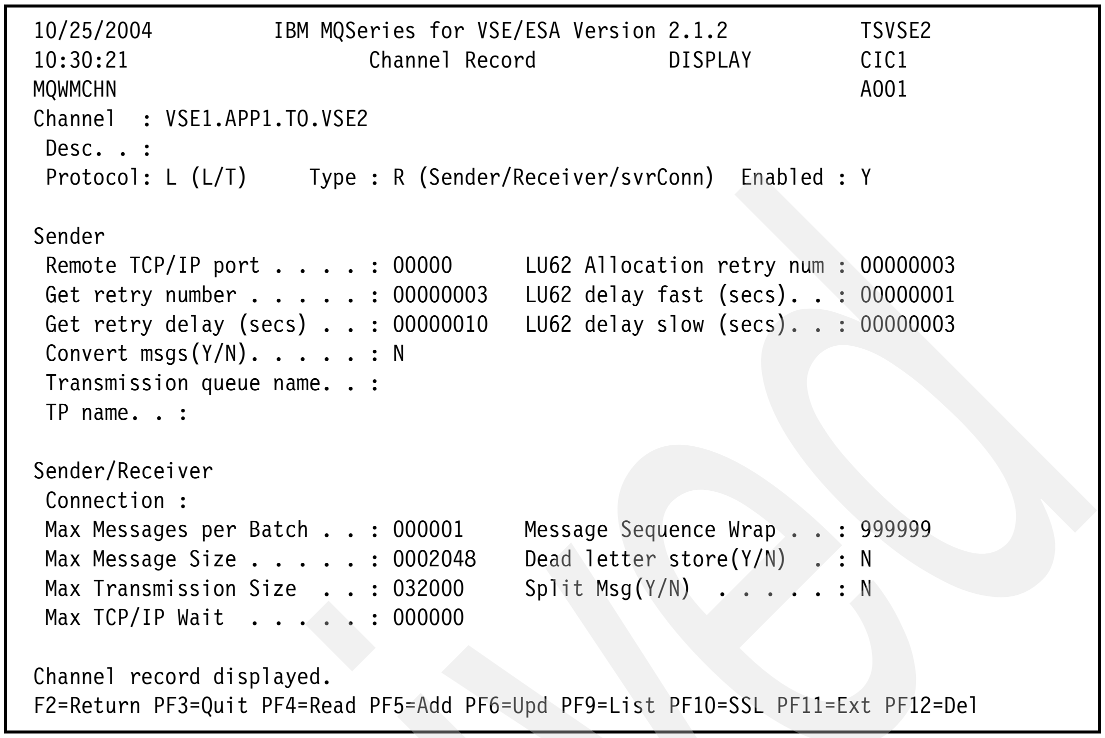
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:30:21                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A001
Channel  : VSE1.APP1.TO.VSE2
 Desc. . :
 Protocol: L (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000003
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000001
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000003
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . : VSE1.APP1.XQ1
 TP name. . : MQ01

Sender/Receiver
 Connection : VSE2
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0002048    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032000     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-8 LU 6.2 sender channel for VSE to VSE distributed queuing

Note that the partner value identifies our connection name and the transmission queue name identifies our transmission queue definition. Also note that the name of our sender channel matches the name we identified in the transmission queue definition.

VSE to VSE receiver channel definition

Finally, we define the LU 6.2 receiver channel on VSE2. Figure A-9 on page 215 shows the receiver channel definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSVSE2
10:30:21                   Channel Record          DISPLAY        CIC1
MQWMCHN                                                           A001
Channel  : VSE1.APP1.TO.VSE2
 Desc. . :
 Protocol: L (L/T)     Type : R (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000003
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000001
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000003
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0002048    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032000     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-9 LU 6.2 receiver channel for VSE to VSE distributed queuing

Note that the receiver channel name is identical to the sender channel name on VSE1. This is mandatory.

This completes our definitions.

VSE to VSE distributed test

Before we run a test program to exploit our distributed configuration, we can check that the VTAM definitions are valid by inquiring on the CICS connection. This test is carried out on the local VSE1 system. For example:

```
CEMT INQ CONN(VSE2)
```

This should show the connection in service and acquired. For example:

```
INQ CONN(VSE2)
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Con(VSE2) Net(IYBPZS01)     Ins Acq         Vta Appc
```

Assuming we do not have any VTAM problems, we can begin the distributed test.

MQSeries for VSE 2.1 provides a test program for putting and getting messages to and from a queue. The queue can be a local or remote queue; however, you cannot *get* from a remote queue. The test program is TTPTST2, which is invoked using transaction TST2 in native CICS. For example:

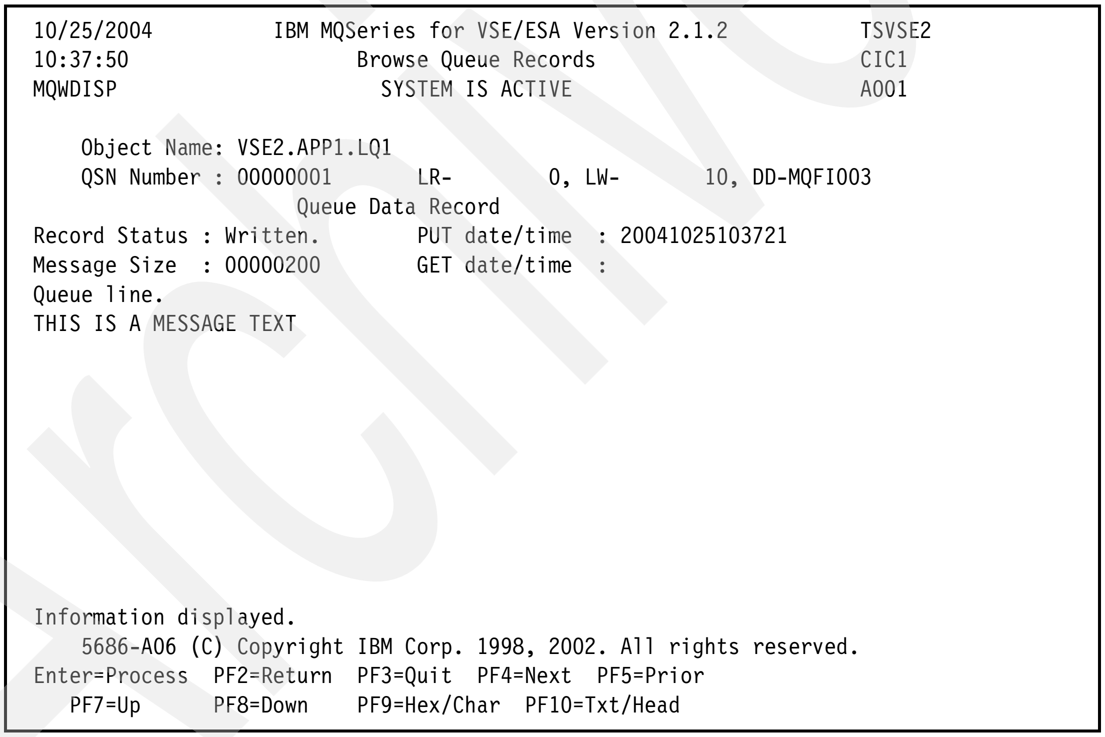
```
TST2 PUT 010 VSE1.APP1.RQ1
```

In this example, the test program attempts to put 10 messages on the remote queue VSE1.APP1.RQ1. The messages are actually written to the transmission queue identified by the remote queue definition (VSE1.APP1.XQ1). The act of putting a message on the transmission queue causes the MQPSEND program to be triggered as per the transmission queue definition.

The MQPSEND program is the sender MCA. It uses the sender channel definition to establish an LU 6.2 connection with the remote VSE2 system. Establishing the connection causes the receiver MCA to be invoked on the remote system. The sender MCA then takes messages from the transmission queue and sends them to the receiver MCA, which in turn, writes them to a local queue on VSE2. The local queue is identified in the remote queue definition. Once a batch of messages (the size of which is specified in the channel definition and negotiated when channels connect) has been successfully transferred, the messages in the batch are committed and the sender MCA then logically deletes the messages from the transmission queue. The sender MCA continues to send messages until it detects that the transmission queue is empty.

For our example, after running the TST2 transaction, we expect 10 messages to be available on the VSE2.APP1.LQ1 local queue on the VSE2 system. This is checked using the MQMT master terminal transaction on the VSE2 system.

Option 4 of the MQMT main menu lets you browse local queues. If our test is successful, we can expect the display under MQMT option 4 as shown in Figure A-10.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE2
10:37:50                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A001

    Object Name: VSE2.APP1.LQ1
    QSN Number : 00000001       LR-        0, LW-       10, DD-MQFI003
                      Queue Data Record
Record Status : Written.        PUT date/time  : 20041025103721
Message Size  : 00000200        GET date/time  :
Queue line.
THIS IS A MESSAGE TEXT










Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure A-10 Test results for VSE to VSE distributed queuing

The fact that we can see 10 messages on our local queue on VSE2 suggests our test was successful.

VSE to OS/390 distributed example

In this example, we set up distributed queuing between a VSE system and an OS/390 system. Specifically, we provide an example of an MQSeries application running on VSE 2.7 putting messages on a remote queue on an OS/390 system using TCP/IP channels.

The prerequisites for this example are:

- MQSeries for VSE 2.1.2 correctly installed on a VSE 2.7 system
- TCP/IP correctly installed on a VSE 2.7 system
- MQSeries for OS/390 correctly installed on an OS/390 system
- TCP/IP correctly installed on an OS/390 system

For the sake of convenience, we refer to the local VSE 2.7 system as VSE1 and the remote OS/390 system as MVS1. The various MQSeries definitions involved in this example can be created in any order; however, we use the following order:

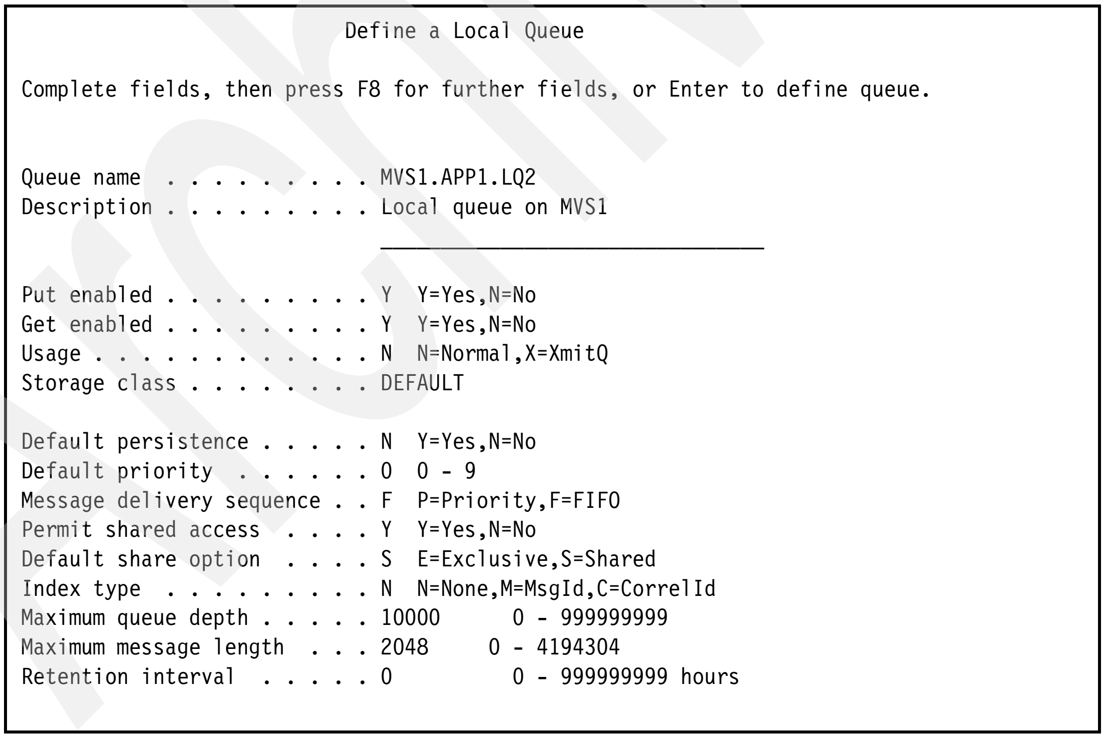
1. Local queue definition
2. Remote queue definition
3. Transmission queue definition
4. Sender channel definition
5. Receiver channel definition

These are sufficient to achieve remote queuing between the two systems.

VSE to OS/390 local queue definition

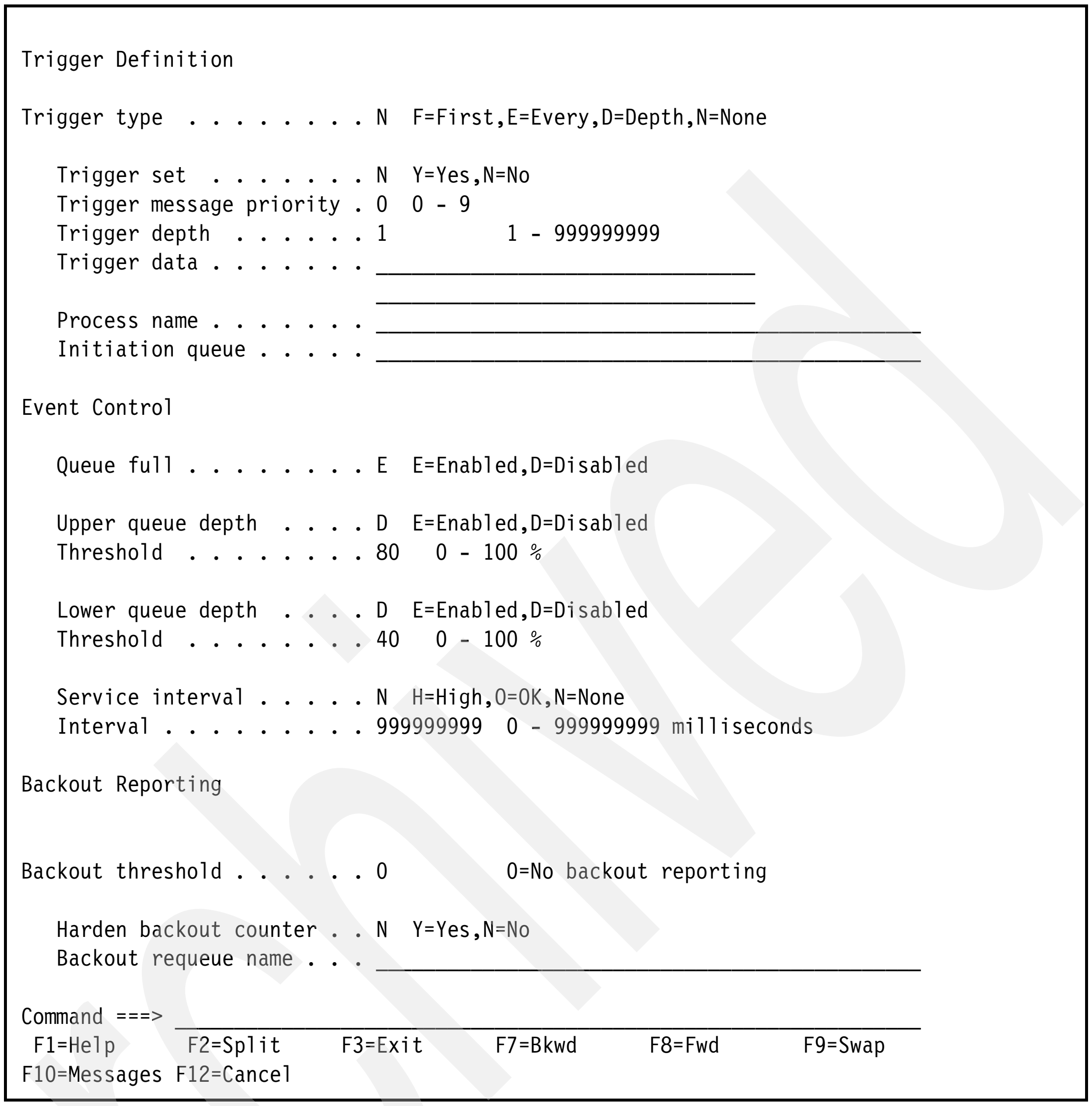
First, we define a local queue on the remote MVS1 system. This queue is identified by the remote queue definition on VSE1. In our example, when we put messages on VSE1, we ultimately expect them to arrive on this remote physical queue.

Figure A-12 on page 218 shows the local queue definition we used on MVS1.



```
                           Define a Local Queue

Complete fields, then press F8 for further fields, or Enter to define queue.


Queue name  . . . . . . . . . MVS1.APP1.LQ2
Description . . . . . . . . . Local queue on MVS1
                              ________________________________

Put enabled . . . . . . . . . Y  Y=Yes,N=No
Get enabled . . . . . . . . . Y  Y=Yes,N=No
Usage . . . . . . . . . . . . N  N=Normal,X=XmitQ
Storage class . . . . . . . . DEFAULT

Default persistence . . . . . N  Y=Yes,N=No
Default priority  . . . . . . 0  0 - 9
Message delivery sequence . . F  P=Priority,F=FIFO
Permit shared access  . . . . Y  Y=Yes,N=No
Default share option  . . . . S  E=Exclusive,S=Shared
Index type  . . . . . . . . . N  N=None,M=MsgId,C=CorrelId
Maximum queue depth . . . . . 10000      0 - 999999999
Maximum message length  . . . 2048     0 - 4194304
Retention interval  . . . . . 0          0 - 999999999 hours
```

Figure A-11 Local Queue Definition for VSE to OS/390 distributed queuing (part I)



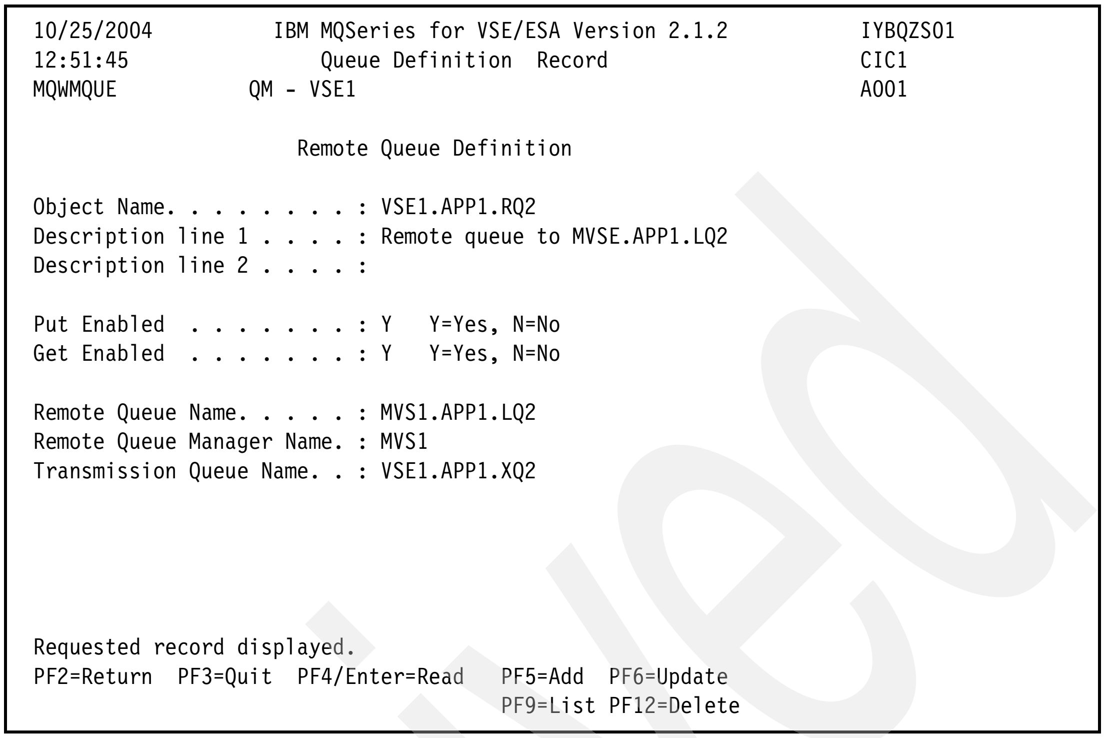
```
Trigger Definition

Trigger type  . . . . . . . . N  F=First,E=Every,D=Depth,N=None

   Trigger set  . . . . . . . N  Y=Yes,N=No
   Trigger message priority . 0  0 - 9
   Trigger depth  . . . . . . 1          1 - 999999999
   Trigger data . . . . . . . ________________________________
                              ________________________________
   Process name . . . . . . . ______________________________________________
   Initiation queue . . . . . ______________________________________________

Event Control

   Queue full . . . . . . . . E  E=Enabled,D=Disabled

   Upper queue depth  . . . . D  E=Enabled,D=Disabled
   Threshold  . . . . . . . . 80   0 - 100 %

   Lower queue depth  . . . . D  E=Enabled,D=Disabled
   Threshold  . . . . . . . . 40   0 - 100 %

   Service interval . . . . . N  H=High,O=OK,N=None
   Interval . . . . . . . . . 999999999  0 - 999999999 milliseconds

Backout Reporting


Backout threshold . . . . . . 0          0=No backout reporting

   Harden backout counter . . N  Y=Yes,N=No
   Backout requeue name . . . ______________________________________________

Command ===> _______________________________________________________________
 F1=Help      F2=Split     F3=Exit      F7=Bkwd      F8=Fwd       F9=Swap
F10=Messages F12=Cancel
```

Figure A-12 Local Queue Definition for VSE to OS/390 distributed queuing (part II)

VSE to OS/390 remote queue definition

We now define a remote queue on the VSE1 system. This identifies the local queue on MVS1 that we just defined.

Figure A-13 on page 219 shows the remote queue definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS01
12:51:45               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE1                                         A001

                     Remote Queue Definition

Object Name. . . . . . . . : VSE1.APP1.RQ2
Description line 1 . . . . : Remote queue to MVSE.APP1.LQ2
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Remote Queue Name. . . . . : MVS1.APP1.LQ2
Remote Queue Manager Name. : MVS1
Transmission Queue Name. . : VSE1.APP1.XQ2





Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read   PF5=Add  PF6=Update
                                       PF9=List PF12=Delete
```

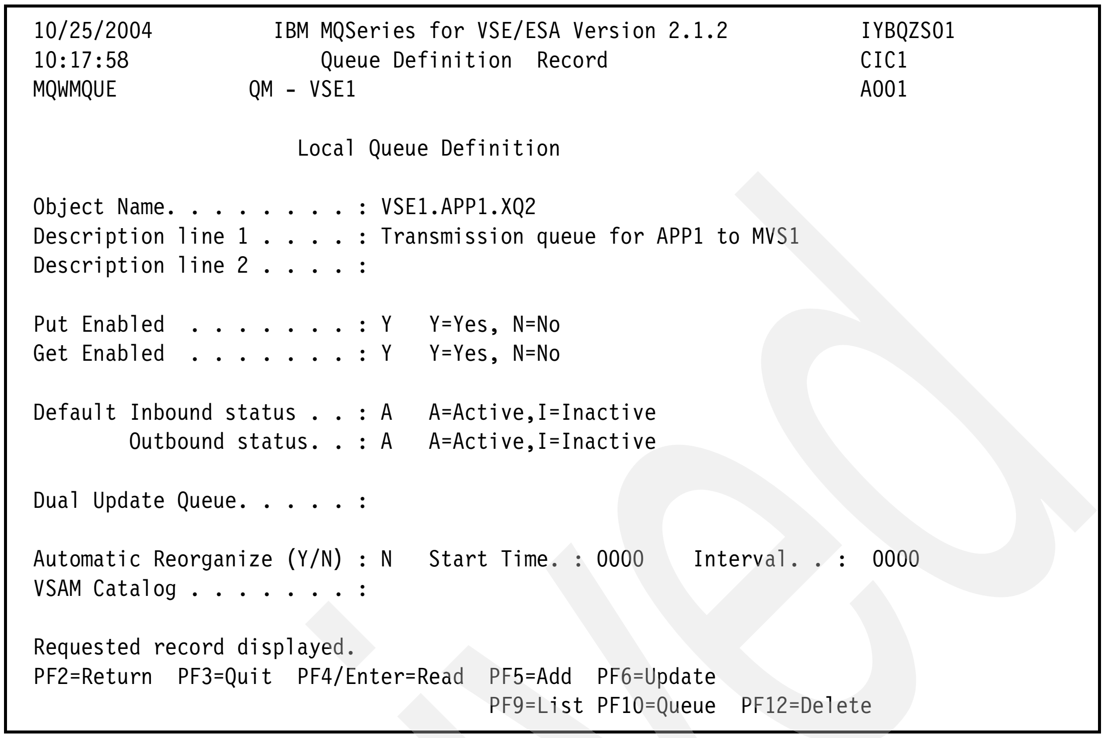
Figure A-13 Remote Queue Definition for VSE to OS/390 distributed queuing

Note that the remote queue definition identifies a transmission queue. Messages logically written to the remote queue are physically written to the transmission queue and are not cleared from the transmission queue until they are successfully received by the remote MQSeries system.

VSE to OS/390 transmission queue definition

The transmission queue is also defined on the local VSE1 system.

Figure A-14 on page 220 shows the transmission queue definition we used.

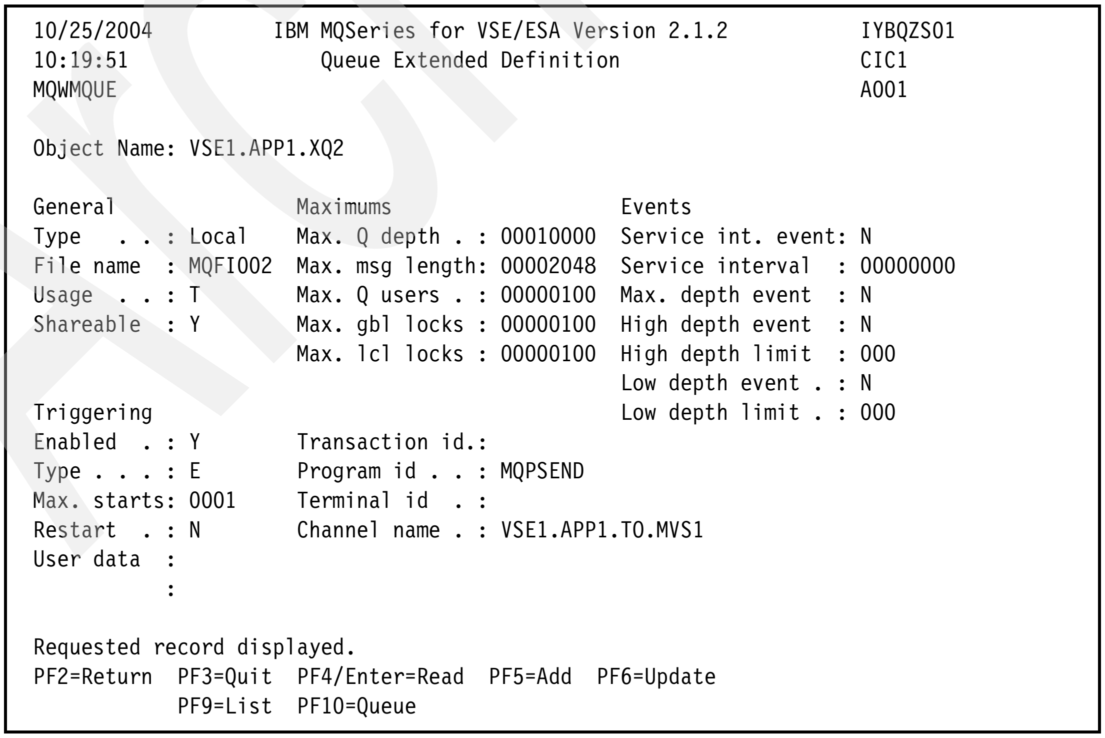


```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS01
10:17:58               Queue Definition  Record                    CIC1
MQWMQUE           QM - VSE1                                        A001

                     Local Queue Definition

Object Name. . . . . . . . : VSE1.APP1.XQ2
Description line 1 . . . . : Transmission queue for APP1 to MVS1
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-14 Transmission Queue Definition for VSE to OS/390

Note that the transmission queue name matches the transmission queue identified in the remote queue definition.

Figure A-15 shows the transmission queue extended definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS01
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: VSE1.APP1.XQ2

General               Maximums                    Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI002  Max. msg length: 00002048  Service interval  : 00000000
Usage  . . : T        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.:
Type . . . : E        Program id . . : MQPSEND
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . : VSE1.APP1.TO.MVS1
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

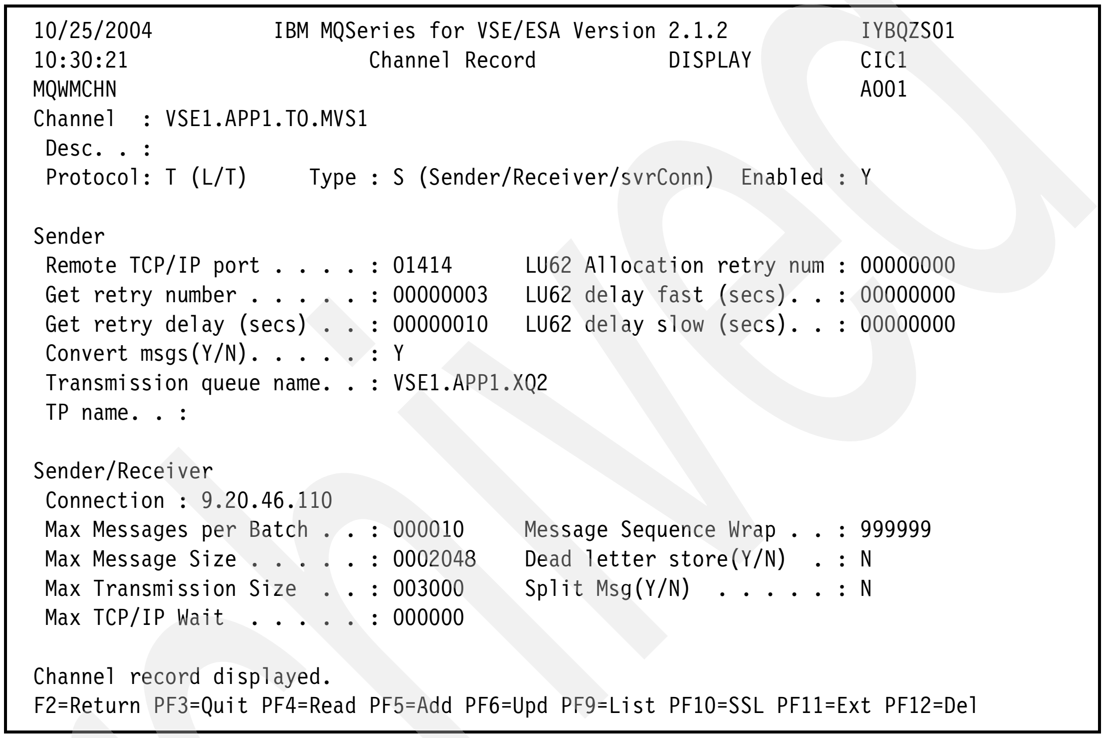
Figure A-15 Transmission Queue Extended Definition for VSE to OS/390

Note that the transmission queue identifies a channel name. This is the TCP/IP sender channel that we define next.

VSE to OS/390 sender channel definition

We now define our MQSeries channels. The sender channel is defined on our local VSE1 system; the receiver channel is defined on our remote MVS1 system.

Figure A-16 shows the TCP/IP sender channel we used.



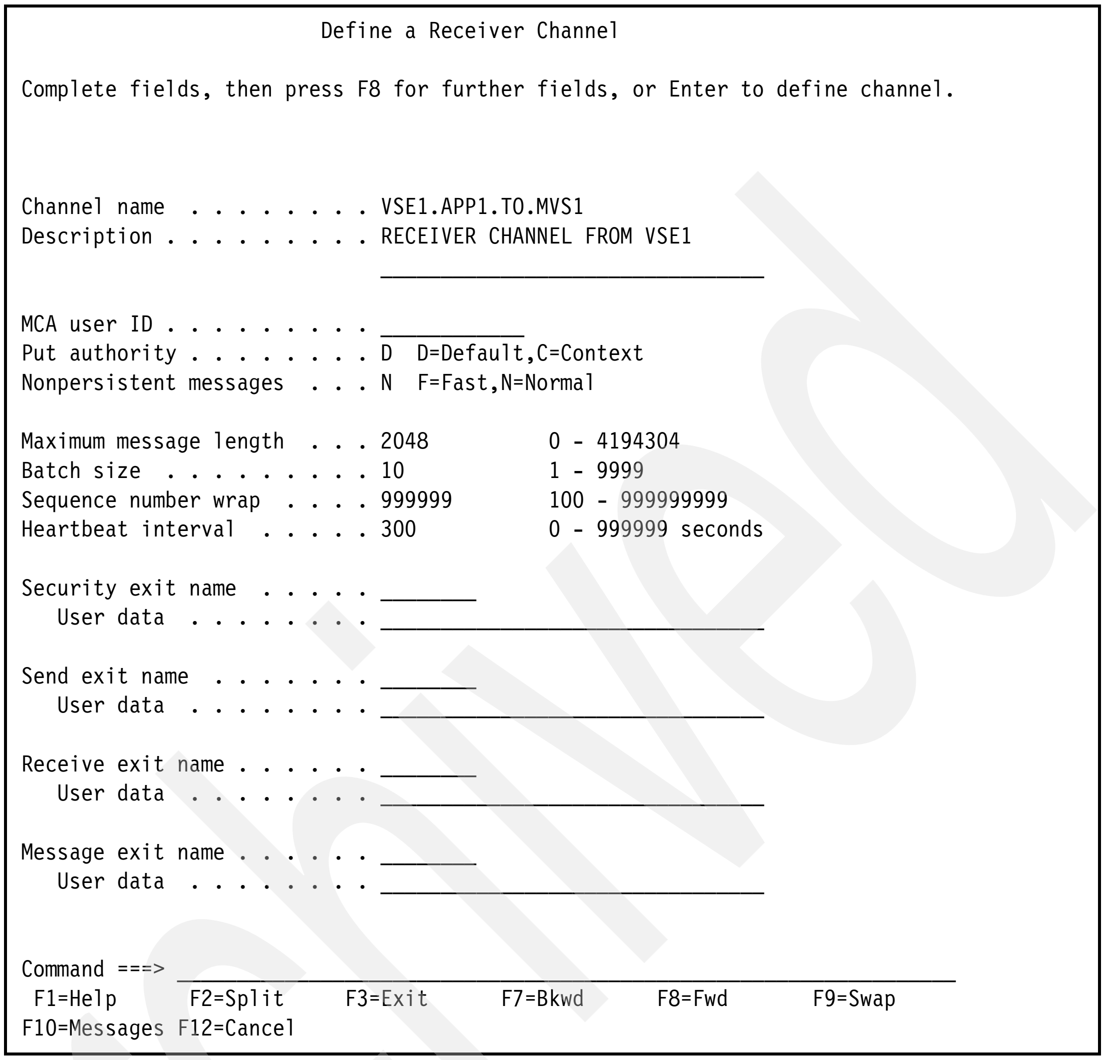
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS01
10:30:21                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A001
Channel  : VSE1.APP1.TO.MVS1
 Desc. . :
 Protocol: T (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 01414      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : Y
 Transmission queue name. . : VSE1.APP1.XQ2
 TP name. . :

Sender/Receiver
 Connection : 9.20.46.110
 Max Messages per Batch . . : 000010     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0002048    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 003000     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-16 TCP/IP sender channel for VSE to OS/390 distributed queuing

Note that the partner value identifies an IP address. This is the IP address of the remote OS/390 system. The port value identifies the TCP/IP port bound to the remote MQSeries listener program. The value 1414 is the MQSeries default; however, you can use any port number approved by your TCP/IP administrator. The transmission queue name identifies our transmission queue definition on VSE1. Also note that the name of our sender channel matches the name we identified in the transmission queue definition. The TP Name is not relevant to TCP/IP channels.

VSE to OS/390 receiver channel definition

Finally, we define the TCP/IP receiver channel on MVS1. Figure A-17 on page 222 shows the receiver channel definition we used.



```
                          Define a Receiver Channel

Complete fields, then press F8 for further fields, or Enter to define channel.



Channel name  . . . . . . . . VSE1.APP1.TO.MVS1
Description . . . . . . . . . RECEIVER CHANNEL FROM VSE1
                              ________________________________

MCA user ID . . . . . . . . . ____________
Put authority . . . . . . . . D  D=Default,C=Context
Nonpersistent messages  . . . N  F=Fast,N=Normal

Maximum message length  . . . 2048          0 - 4194304
Batch size  . . . . . . . . . 10            1 - 9999
Sequence number wrap  . . . . 999999        100 - 999999999
Heartbeat interval  . . . . . 300           0 - 999999 seconds

Security exit name  . . . . . ________
   User data  . . . . . . . . ________________________________

Send exit name  . . . . . . . ________
   User data  . . . . . . . . ________________________________

Receive exit name . . . . . . ________
   User data  . . . . . . . . ________________________________

Message exit name . . . . . . ________
   User data  . . . . . . . . ________________________________


Command ===> _________________________________________________________________
 F1=Help      F2=Split     F3=Exit      F7=Bkwd      F8=Fwd       F9=Swap
F10=Messages F12=Cancel
```

Figure A-17 TCP/IP receiver channel for VSE to OS/390 distributed queuing

Note that the receiver channel name is identical to the sender channel name on VSE1. This is mandatory. Also note that the receiver channel definition values match the values specified in the sender channel. This is necessary for the two MCAs to agree on how the connection operates.

This completes our definitions.

VSE to OS/390 distributed test

Before we run a test program to exploit our distributed configuration, we can check that we have a valid TCP/IP connection between the two hosts. This can be done using the TCP/IP **ping** program. It is important that we test the connection *from* VSE1 *to* MVS1.

On VSE, there are two ways to use the **ping** program:

- Issue a command to TCP/IP from your VSE console.
- Run the **ping** transaction in native CICS.

For our test, we issue a command from the VSE console. Our TCP/IP service is running in F5. For example:

```
msg f5
AR 0015 1I40I  READY
F5 0085 IPN300I Enter TCP/IP Command
F5-0085
85 ping 9.20.46.94
F5 0083 TCP910I  Client manager connection Established.
F5 0083 TCP910I  PING
F5 0083 TCP910I  PING Ready:
F5 0083 TCP910I  SET HOST= 9.20.46.94
F5 0083 TCP910I  009.020.046.094
F5 0083 TCP910I  PING Ready:
F5 0083 TCP910I  PING
F5 0083 TCP910I  PING 1 was successful, milliseconds:000280
F5 0083 TCP910I  PING 2 was successful, milliseconds:000180
F5 0083 TCP910I  PING 3 was successful, milliseconds:000140
F5 0083 TCP910I  PING 4 was successful, milliseconds:000160
F5 0083 TCP910I  PING 5 was successful, milliseconds:000190
F5 0083 TCP910I  PING Ready:
F5 0085 IPN300I Enter TCP/IP Command
F5-0085
85
```

At this point, we have established that we have a TCP/IP connection between the two hosts. It is also necessary to check that the MQSeries TCP/IP listener program is running on the remote host MVS1. If this program is not running, no TCP/IP channel connections are possible.

You can check the listener program's status on OS/390 via SDSF. SDSF should be an option on your ISPF primary options menu. In SDSF, use the LOG command to display the system log. From the command line, enter:

```
/+N DISPLAY DQM
```

Where *N* is the number associated with your MQSeries queue manager. You may have to page to the bottom of the system log to see the results of this display command. You should see a message similar to the following:

```
CSQM137I +5 CSQMDDQM  DISPLAY DQM COMMAND ACCEPTED
CSQX830I +5 CSQXRDQM Channel initiator active
CSQX845I +5 CSQXRDQM TCP/IP address space name is TCPIP
CSQX846I +5 CSQXRDQM TCP/IP listener started, for port number 1414
CSQX849I +5 CSQXRDQM LU6.2 listener not started
CSQX832I +5 CSQXRDQM 5 dispatchers started, 5 requested
CSQX831I +5 CSQXRDQM 8 adapter subtasks started, 8 requested
CSQX840I +5 CSQXRDQM 9 channel connections current, maximum 200
CSQX841I +5 CSQXRDQM 0 channel connections active, maximum 200
CSQX842I +5 CSQXRDQM 0 channel connections starting, 100
9 stopped, 0 retrying
CSQ9022I +5 CSQXCRPS ' DISPLAY DQM' NORMAL COMPLETION
```

If not, you can start the TCP/IP listener program from the SDSF system log with the following command:

```
/+N START LISTENER TRPTYPE(TCP) PORT(1414)
```

If you do not have the authority to run this command, speak to your MQSeries system administrator.

Assuming we do not have any TCP/IP problems and the MQSeries listener program is active, we can begin the distributed test.

MQSeries for VSE 2.1 provides a test program for putting and getting messages to and from a queue. The queue can be a local or remote queue; however, you cannot *get* from a remote queue. The test program is TTPTST2, which is invoked using transaction TST2 in native CICS. For example:

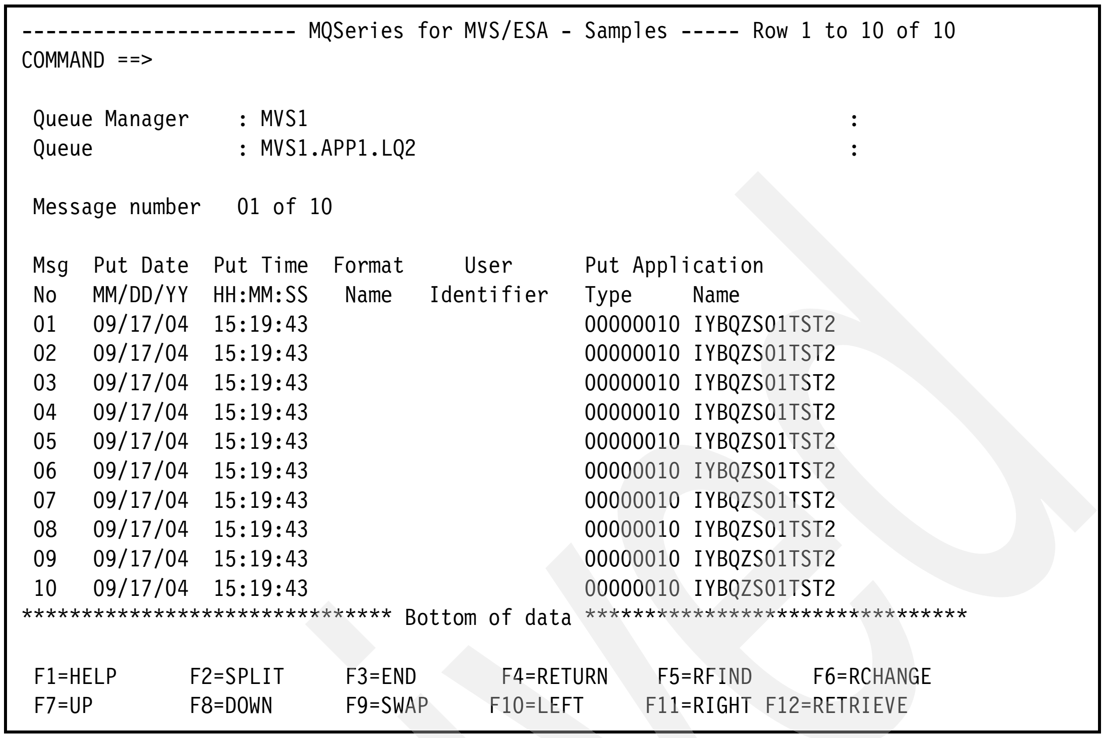
```
TST2 PUT 010 VSE1.APP1.RQ2
```

In this example, the test program attempts to put 10 messages on the remote queue VSE1.APP1.RQ2. The messages are actually written to the transmission queue identified by the remote queue definition (VSE1.APP1.XQ2). The act of putting a message on the transmission queue causes the MQPSEND program to be triggered as per the transmission queue definition.

The MQPSEND program is the sender MCA. It uses the sender channel definition to establish a TCP/IP connection with the remote MVS1 system. Establishing the connection causes the receiver MCA to be invoked on the remote system. The sender MCA then takes messages from the transmission queue and sends them to the receiver MCA, which in turn, writes them to a local queue on MVS1. The local queue is identified in the remote queue definition. Once a batch of messages (the size of which is specified in the channel definition and negotiated when channels connect) has been successfully transferred, the messages in the batch are committed and the sender MCA then logically deletes the messages from the transmission queue. The sender MCA continues to send messages until it detects that the transmission queue is empty.

For our example, after running the TST2 transaction, we expect 10 messages to be available on the MVS1.APP1.LQ2 local queue on the MVS1 system. This is checked using the ISPF MQSeries option to browse a queue.

If our test is successful we can expect the display shown in Figure A-18 on page 225.



```
---------------------- MQSeries for MVS/ESA - Samples ----- Row 1 to 10 of 10
COMMAND ==>

 Queue Manager    : MVS1                                         :
 Queue            : MVS1.APP1.LQ2                                :

 Message number   01 of 10

 Msg  Put Date  Put Time  Format     User      Put Application
 No   MM/DD/YY  HH:MM:SS   Name   Identifier   Type     Name
 01   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 02   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 03   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 04   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 05   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 06   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 07   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 08   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 09   09/17/04  15:19:43                       00000010 IYBQZS01TST2
 10   09/17/04  15:19:43                       00000010 IYBQZS01TST2
******************************* Bottom of data ********************************

 F1=HELP      F2=SPLIT     F3=END       F4=RETURN    F5=RFIND     F6=RCHANGE
 F7=UP        F8=DOWN      F9=SWAP     F10=LEFT     F11=RIGHT F12=RETRIEVE
```

Figure A-18 Test results for VSE to OS/390 distributed queuing

The fact that we can see 10 messages on our local queue on MVS1 suggests our test was successful.

OS/390 to VSE distributed example

In this example, we set up distributed queuing between an OS/390 system and a VSE 2.6 system. Specifically, we provide an example of an MQSeries application running on OS/390 putting messages on a remote queue on a VSE 2.6 system using TCP/IP channels.

The prerequisites for this example are:

- MQSeries for OS/390 correctly installed on an OS/390 system
- TCP/IP correctly installed on an OS/390 system
- MQSeries for VSE 2.1.2 correctly installed on a VSE 2.6 system
- TCP/IP correctly installed on a VSE 2.6 system

For convenience, we refer to the local OS/390 system as MVS1 and the remote VSE system as VSE2. The various MQSeries definitions involved in this example can be created in any order; however, we use the following order:

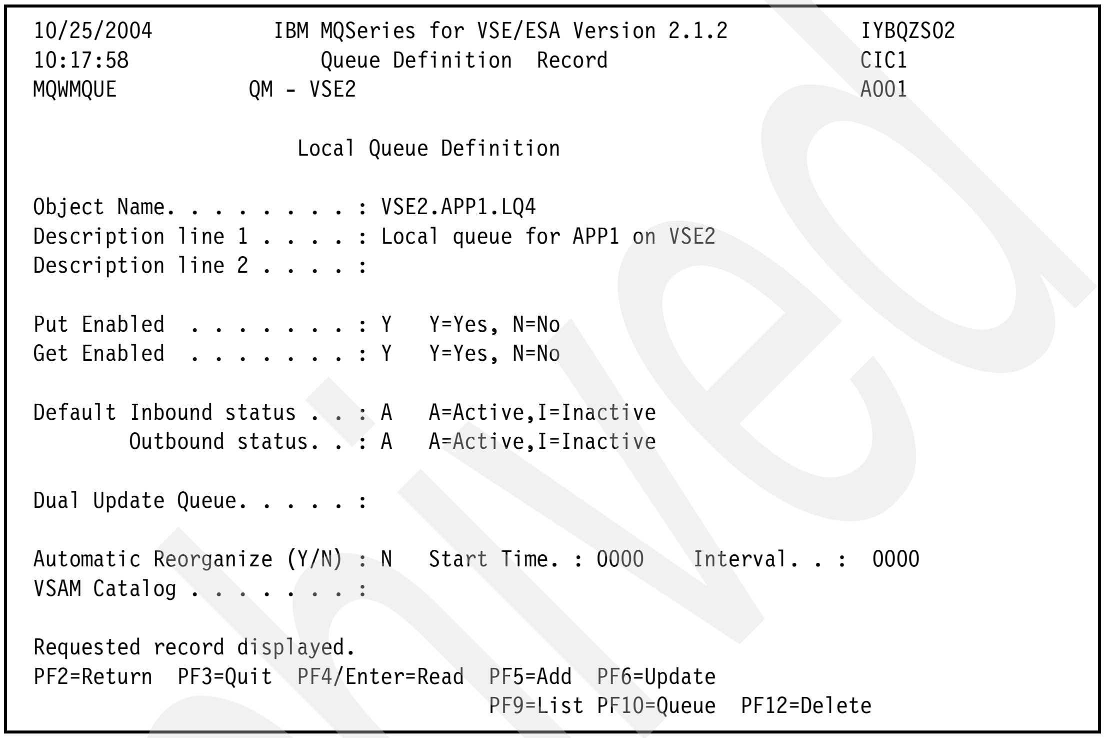
1. Local queue definition
2. Remote queue definition
3. Transmission queue definition
4. Sender channel definition
5. Receiver channel definition

These are sufficient to achieve remote queuing between the two systems.

OS/390 to VSE local queue definition

First, we define a local queue on the remote VSE2 system. This queue is identified by the remote queue definition on MVS1. In our example, when we put messages on MVS1, we ultimately expect them to arrive on this remote physical queue.

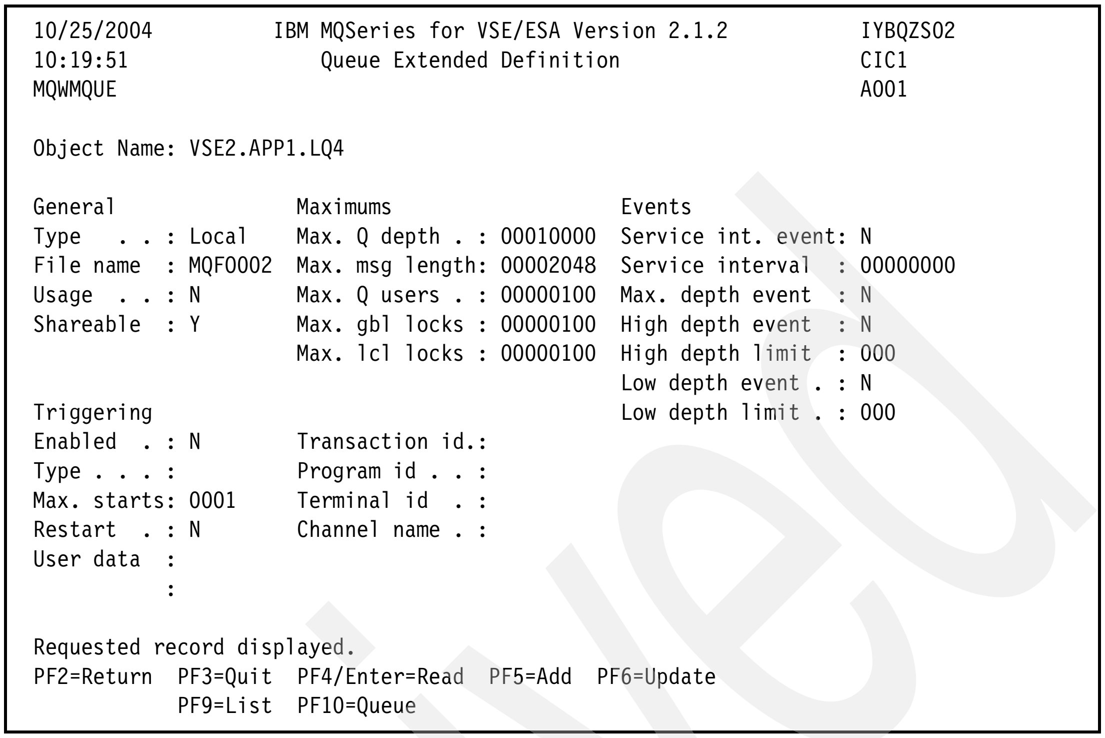
Figure A-19 shows the local queue definition we used on VSE2.



```
10/25/2004          IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS02
10:17:58                Queue Definition  Record                    CIC1
MQWMQUE           QM - VSE2                                         A001

                      Local Queue Definition

Object Name. . . . . . . . . : VSE2.APP1.LQ4
Description line 1 . . . . . : Local queue for APP1 on VSE2
Description line 2 . . . . . :

Put Enabled  . . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . . : A   A=Active,I=Inactive
        Outbound status. . . : A   A=Active,I=Inactive

Dual Update Queue. . . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-19 Local Queue Definition for OS/390 to VSE distributed queuing

Figure A-20 on page 227 shows the local queue extended definition we used on VSE2.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS02
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: VSE2.APP1.LQ4

General              Maximums                   Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFO002  Max. msg length: 00002048  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : N        Transaction id.:
Type . . . :          Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-20 Local Queue Extended Definition for OS/390 to VSE

OS/390 to VSE remote queue definition

We now define a remote queue on the MVS1 system. This identifies the local queue on VSE2 that we just defined.

Figure A-21 on page 228 shows the remote queue definition we used.

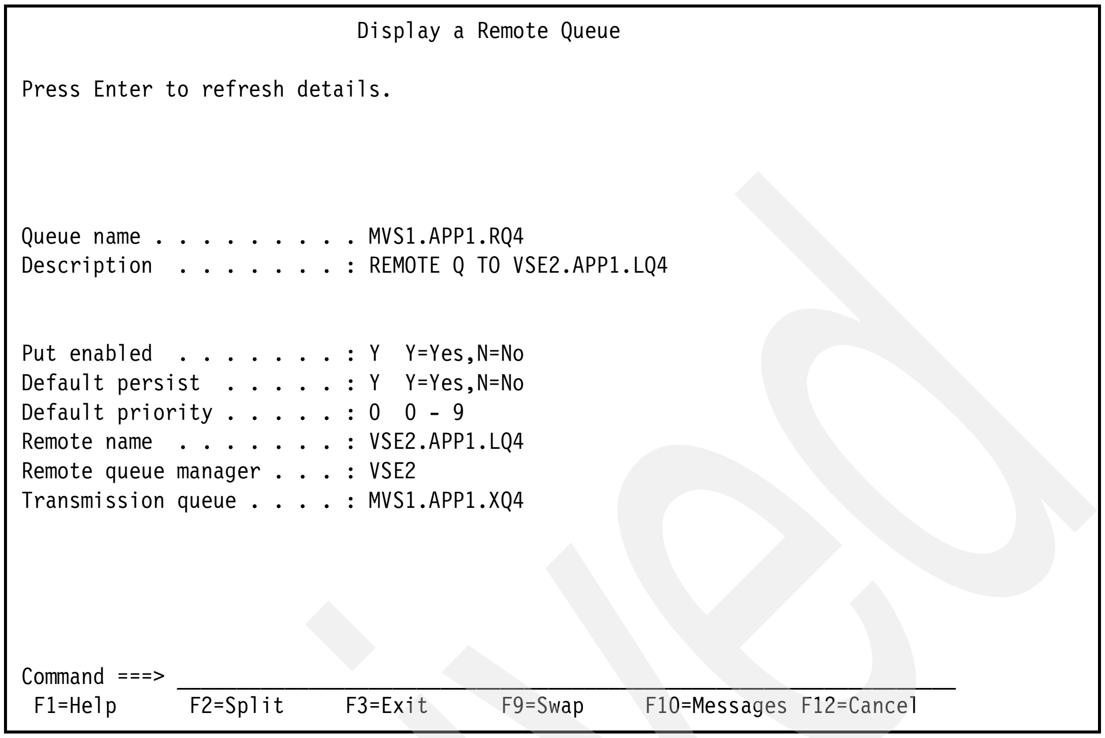
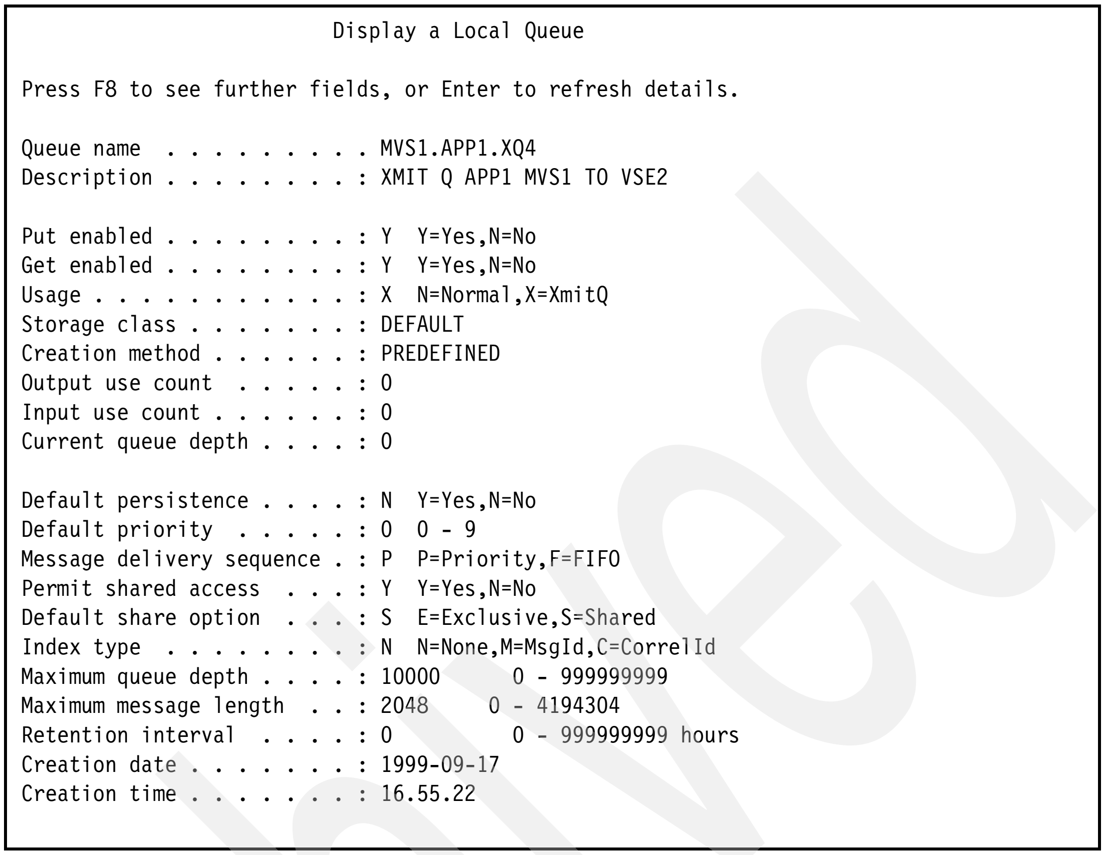

```
                              Display a Remote Queue

Press Enter to refresh details.




Queue name . . . . . . . . . MVS1.APP1.RQ4
Description  . . . . . . . : REMOTE Q TO VSE2.APP1.LQ4


Put enabled  . . . . . . . : Y  Y=Yes,N=No
Default persist  . . . . . : Y  Y=Yes,N=No
Default priority . . . . . : 0  0 - 9
Remote name  . . . . . . . : VSE2.APP1.LQ4
Remote queue manager . . . : VSE2
Transmission queue . . . . : MVS1.APP1.XQ4






Command ===> _________________________________________________________________
 F1=Help      F2=Split     F3=Exit      F9=Swap     F10=Messages F12=Cancel
```

Figure A-21 Remote queue definition for OS/390 to VSE distributed queuing

Note that the remote queue definition identifies a transmission queue. Messages logically written to the remote queue are physically written to the transmission queue and are not cleared from the transmission queue until they are successfully received by the remote MQSeries system.

OS/390 to VSE transmission queue definition

The transmission queue is also defined on the local MVS1 system. Figure A-22 on page 229 shows the transmission queue definition we used.



```
                          Display a Local Queue

Press F8 to see further fields, or Enter to refresh details.

Queue name  . . . . . . . . . MVS1.APP1.XQ4
Description . . . . . . . . : XMIT Q APP1 MVS1 TO VSE2

Put enabled . . . . . . . . : Y  Y=Yes,N=No
Get enabled . . . . . . . . : Y  Y=Yes,N=No
Usage . . . . . . . . . . . : X  N=Normal,X=XmitQ
Storage class . . . . . . . : DEFAULT
Creation method . . . . . . : PREDEFINED
Output use count  . . . . . : 0
Input use count . . . . . . : 0
Current queue depth . . . . : 0

Default persistence . . . . : N  Y=Yes,N=No
Default priority  . . . . . : 0  0 - 9
Message delivery sequence . : P  P=Priority,F=FIFO
Permit shared access  . . . : Y  Y=Yes,N=No
Default share option  . . . : S  E=Exclusive,S=Shared
Index type  . . . . . . . . : N  N=None,M=MsgId,C=CorrelId
Maximum queue depth . . . . : 10000      0 - 999999999
Maximum message length  . . : 2048     0 - 4194304
Retention interval  . . . . : 0          0 - 999999999 hours
Creation date . . . . . . . : 1999-09-17
Creation time . . . . . . . : 16.55.22
```

Figure A-22   Transmission queue definition for OS/390 to VSE (part I)

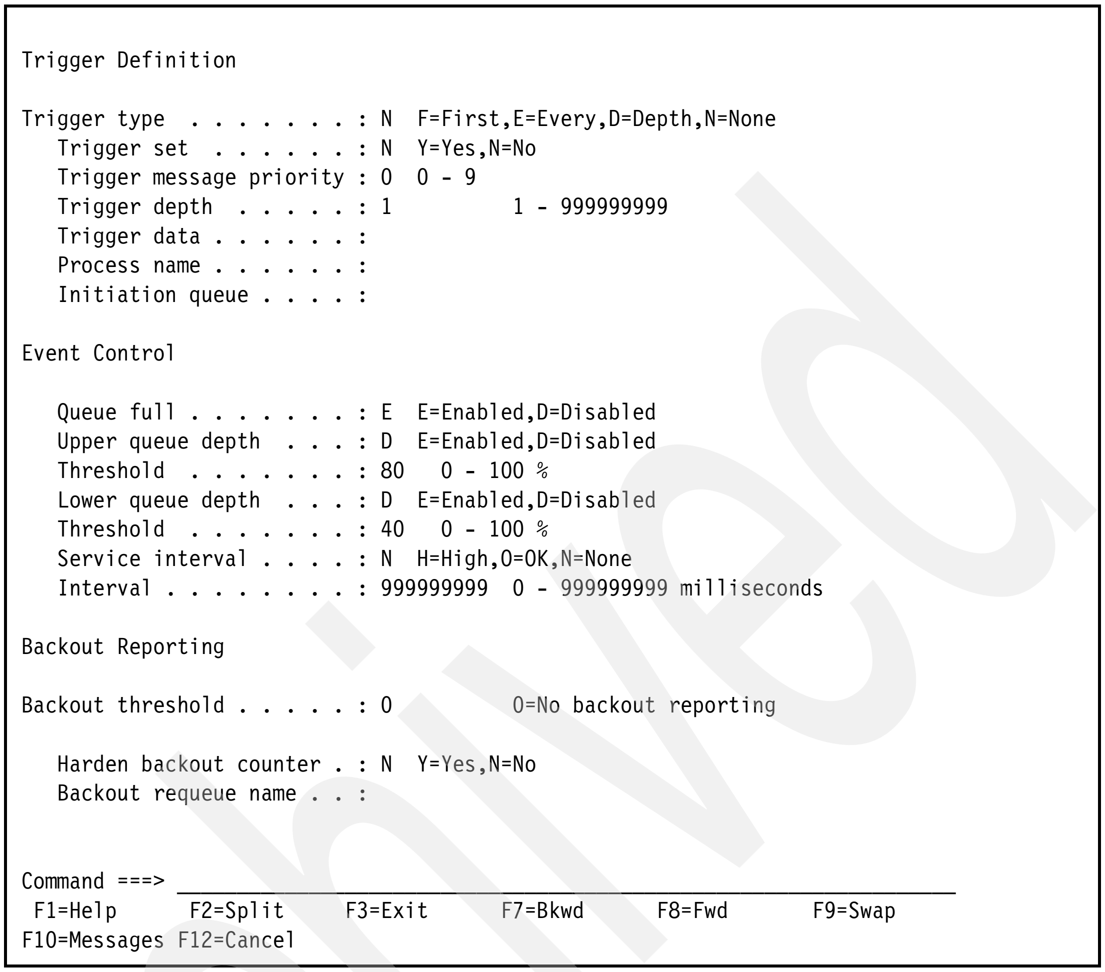

```
Trigger Definition

Trigger type  . . . . . . . : N  F=First,E=Every,D=Depth,N=None
   Trigger set  . . . . . . : N  Y=Yes,N=No
   Trigger message priority : 0  0 - 9
   Trigger depth  . . . . . : 1          1 - 999999999
   Trigger data . . . . . . :
   Process name . . . . . . :
   Initiation queue . . . . :

Event Control

   Queue full . . . . . . . : E  E=Enabled,D=Disabled
   Upper queue depth  . . . : D  E=Enabled,D=Disabled
   Threshold  . . . . . . . : 80   0 - 100 %
   Lower queue depth  . . . : D  E=Enabled,D=Disabled
   Threshold  . . . . . . . : 40   0 - 100 %
   Service interval . . . . : N  H=High,O=OK,N=None
   Interval . . . . . . . . : 999999999  0 - 999999999 milliseconds

Backout Reporting

Backout threshold . . . . . : 0          0=No backout reporting

   Harden backout counter . : N  Y=Yes,N=No
   Backout requeue name . . :


Command ===> _________________________________________________________________
 F1=Help      F2=Split     F3=Exit      F7=Bkwd      F8=Fwd       F9=Swap
F10=Messages F12=Cancel
```

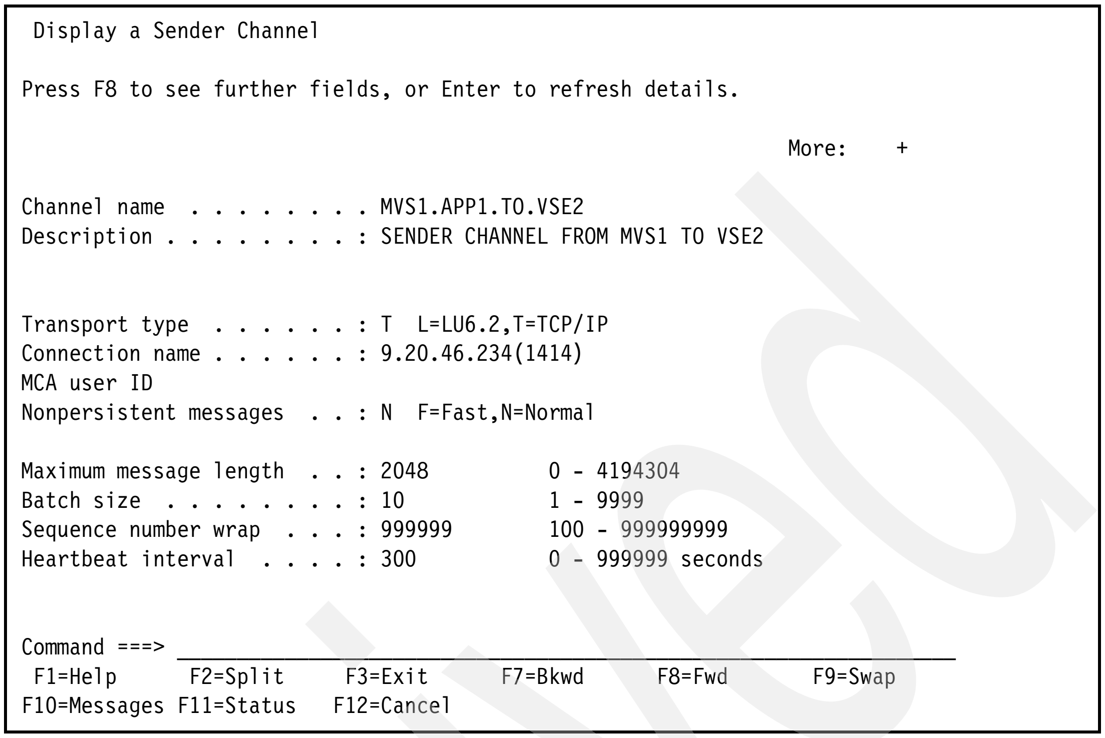
Figure A-23 Transmission queue definition for OS/390 to VSE (part II)

Note that the transmission queue name matches the transmission queue identified in the remote queue definition. The transmission queue also identifies a channel name. This is the TCP/IP sender channel that we define next.

OS/390 to VSE sender channel definition

We now define our MQSeries channels. The sender channel is defined on our local MVS1 system; the receiver channel is defined on our remote VSE2 system.

Figure A-24 on page 231 shows the TCP/IP sender channel we used.



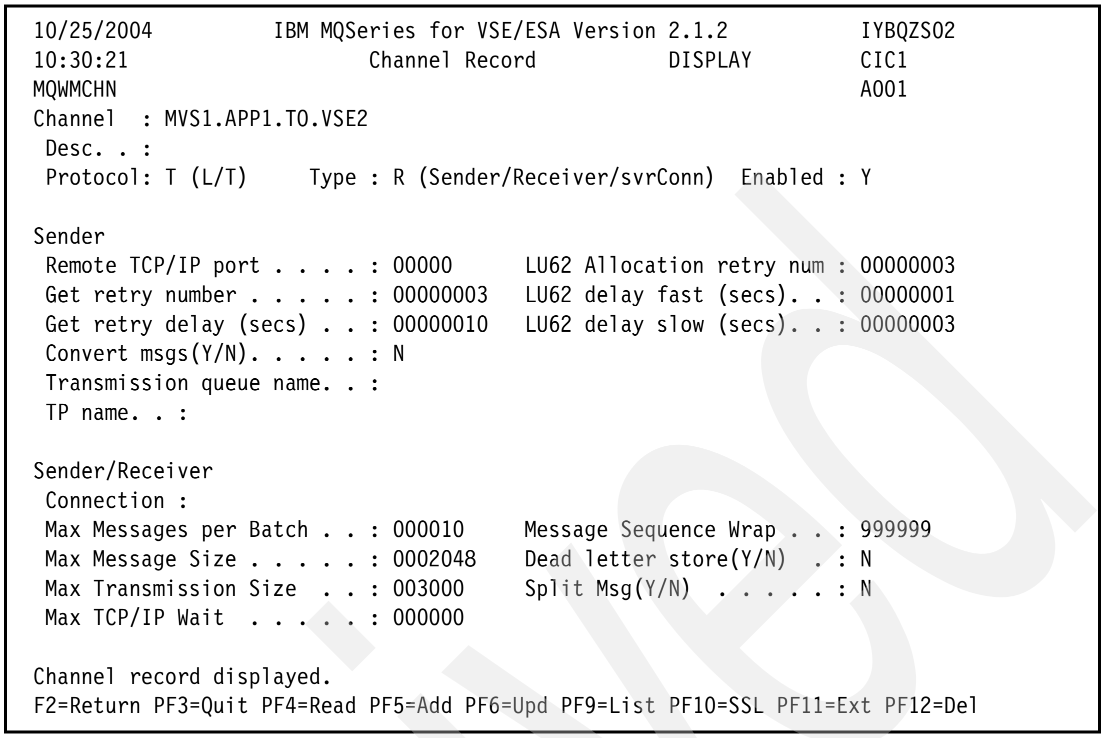
```
 Display a Sender Channel

Press F8 to see further fields, or Enter to refresh details.

                                                                More:    +

Channel name  . . . . . . . . MVS1.APP1.TO.VSE2
Description . . . . . . . . : SENDER CHANNEL FROM MVS1 TO VSE2


Transport type  . . . . . . : T  L=LU6.2,T=TCP/IP
Connection name . . . . . . : 9.20.46.234(1414)
MCA user ID
Nonpersistent messages  . . : N  F=Fast,N=Normal

Maximum message length  . . : 2048          0 - 4194304
Batch size  . . . . . . . . : 10            1 - 9999
Sequence number wrap  . . . : 999999        100 - 999999999
Heartbeat interval  . . . . : 300           0 - 999999 seconds


Command ===> _________________________________________________________________
 F1=Help      F2=Split     F3=Exit      F7=Bkwd      F8=Fwd       F9=Swap
F10=Messages F11=Status   F12=Cancel
```

Figure A-24 TCP/IP sender channel for OS/390 to VSE distributed queuing

Note that the connection name value identifies the IP address. This is the IP address of the remote VSE 2.6 system. The value in brackets is the TCP/IP port number bound to the remote MQSeries listener program. The value 1414 is the MQSeries default; however, you can use any port number approved by your TCP/IP administrator. Also note that the name of our sender channel matches the name we identified in the transmission queue definition.

OS/390 to VSE receiver channel definition

Finally, we define the TCP/IP receiver channel on VSE2. Figure A-25 on page 232 shows the receiver channel definition we used.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS02
10:30:21                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A001
Channel  : MVS1.APP1.TO.VSE2
 Desc. . :
 Protocol: T (L/T)     Type : R (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000003
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000001
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000003
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000010     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0002048    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 003000     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-25 TCP/IP receiver channel for OS/390 to VSE distributed queuing

Note that the receiver channel name is identical to the sender channel name on MVS1. This is mandatory. Also note that the receiver channel definition values match the values specified in the sender channel. This is necessary for the two MCAs to agree on how the connection operates. The Port, Partner, Transmission Queue Name, and TP Name are not relevant for TCP/IP listener channels.

This completes our definitions.

OS/390 to VSE distributed test

Before we run a test program to exploit our distributed configuration, we can check that we have a valid TCP/IP connection between the two hosts. This can be done using the TCP/IP **ping** program. It is important that we test the connection *from* MVS1 *to* VSE2.

On OS/390, you can use ISPF option 6 from the primary options menu. Option 6 is the ISPF command shell. At the command prompt, we can ping the VSE 2.3 system as follows:

```
ping 9.20.46.234
```

We expect output similar to following:

```
Ping V3R2: Pinging host 9.20.46.234.  Use ATTN to interrupt.
PING: Ping #1 response took 0.020 seconds. Successes so far 1.
```

At this point, we have established that we have a TCP/IP connection between the two hosts. It is also necessary to check that the MQSeries TCP/IP listener program is running on the remote host VSE2. If this program is not running, no TCP/IP channel connections are possible.

You can check the listener program's status on VSE via the CEMT transaction in native CICS. For example:

```
CEMT I TASK
```

You should see a display similar to the following:

```
i task
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(00376) Tra(MQTL)           Act Tas
 Tas(00017) Tra(CXPB)           Act Tas
 Tas(00405) Tra(CEMT) Fac(SFC3) Act Ter
 Tas(00377) Tra(MQSM)           Sus Tas
```

The MQTL transaction is the MQSeries for VSE 2.1 TCP/IP listener program. It is started automatically when you start MQSeries if you have specified a non-0 port number in your global system definition. MQSeries only accepts connections from remote hosts that identify the port number specified in the global system definition. Use MQMT option 1.1 to check your port number specification. The port number you use must match the port number specified in relevant sender channel definitions.

Assuming we do not have any TCP/IP problems and the MQSeries listener program is active, we can begin the distributed test.

The distributed test is initiated on the OS/390 system. In our case, we use a simple PL/I program that reads input to identify a target queue, the number of messages, message text, and message length. For example:

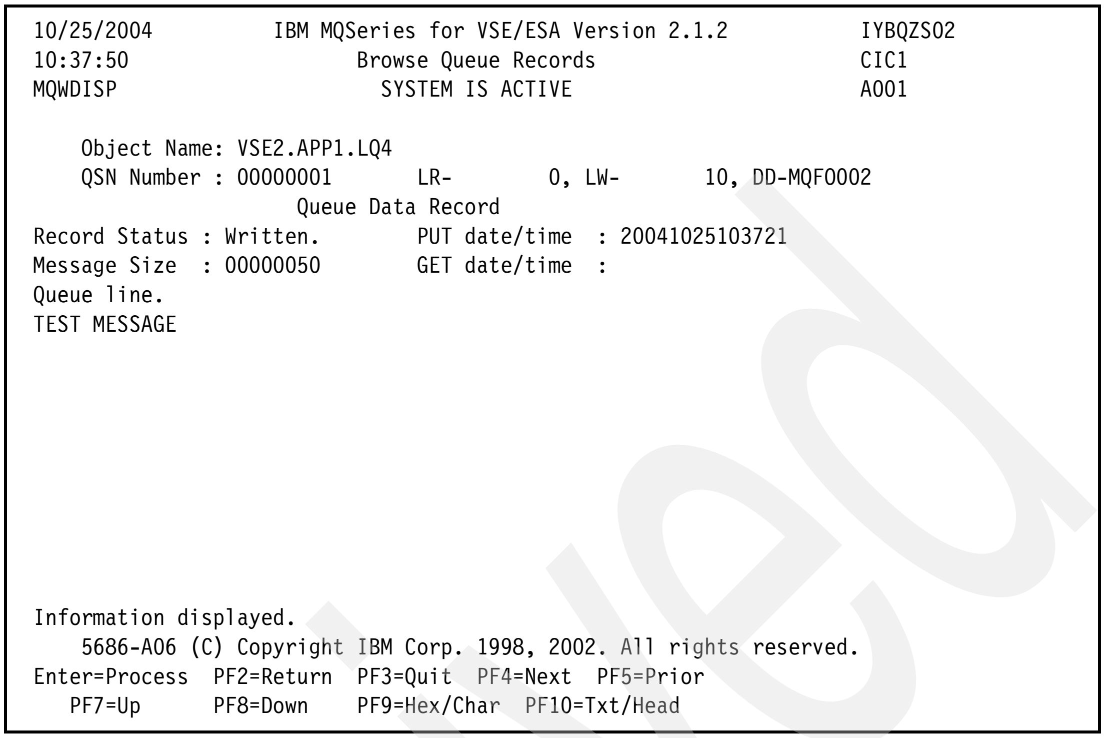
```
//MQTEST1  EXEC PGM=MQTPUT,REGION=512K,PARM='/MVS1'
//STEPLIB  DD  DSN=MQ.TEST.LOADLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//INPUT    DD  *
QN:MVS1.APP1.RQ4
QC:10
MT:TEST MESSAGE
ML:50
/*
```

This test program is not shipped with MQSeries for OS/390. Our test attempts to write 10 messages to remote queue MVS1.APP1.RQ4, that is defined to queue manager MVS1. The messages are actually written to the transmission queue identified by the remote queue definition (MVS1.APP1.XQ4). The act of putting a message on the transmission queue causes the MQSeries sender MCA program to establish a connection with the VSE 2.6 system. The sender MCA uses the sender channel definition to establish a TCP/IP connection. Establishing the connection causes the receiver MCA to be invoked on the remote system.

The sender MCA then takes messages from the transmission queue and sends them to the receiver MCA, which in turn, writes them to a local queue on VSE2. The local queue is identified in the remote queue definition. Once a batch of messages (the size of the batch was negotiated when channels connected) has been successfully transferred, the messages are committed and the sender MCA then deletes the messages from the transmission queue. The sender MCA continues to send messages until it detects that the transmission queue is empty.

For our example, after running the test program, we expect 10 messages to be available on the VSE2.APP1.LQ4 local queue on the VSE2 system. This is checked using MQMT option 4.

Option 4 of the MQMT main menu lets you browse local queues. If our test is successful, we can expect the display under MQMT option 4 shown in Figure A-26 on page 234.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          IYBQZS02
10:37:50                  Browse Queue Records                     CIC1
MQWDISP                     SYSTEM IS ACTIVE                       A001

    Object Name: VSE2.APP1.LQ4
    QSN Number : 00000001       LR-        0, LW-       10, DD-MQF0002
                      Queue Data Record
Record Status : Written.        PUT date/time  : 20041025103721
Message Size  : 00000050        GET date/time  :
Queue line.
TEST MESSAGE











Information displayed.
    5686-A06 (C) Copyright IBM Corp. 1998, 2002. All rights reserved.
Enter=Process  PF2=Return  PF3=Quit  PF4=Next  PF5=Prior
   PF7=Up      PF8=Down    PF9=Hex/Char  PF10=Txt/Head
```

Figure A-26 Test results for OS/390 to VSE distributed queuing

The fact that we can see 10 messages on our local queue on VSE2 suggests our test was successful.

Windows XP and VSE distributed example

In this section, we provide you with examples to define peer-to-peer communication between VSE and Windows XP connected through a TCP/IP network. We assume that MQSeries is currently installed on both systems.

In this example, we show both VSE to Windows XP and Windows XP to VSE.

VSE definitions

In our scenario, we wish to exchange messages in both directions between VSE and Windows XP. Therefore, under VSE, we define:

- A sender channel called VSE.TO.WINXP
- A receiver channel called WINXP.TO.VSE
- A transmission queue called VSE.TO.WINXP_XMIT.Q
- A local queue to receive messages called WIN_INPUT
- A remote queue to send messages called WIN_OUTPUT

VSE and Windows XP global system definition

An example of the communication settings for our VSE queue manager is shown in Figure A-27 on page 235.

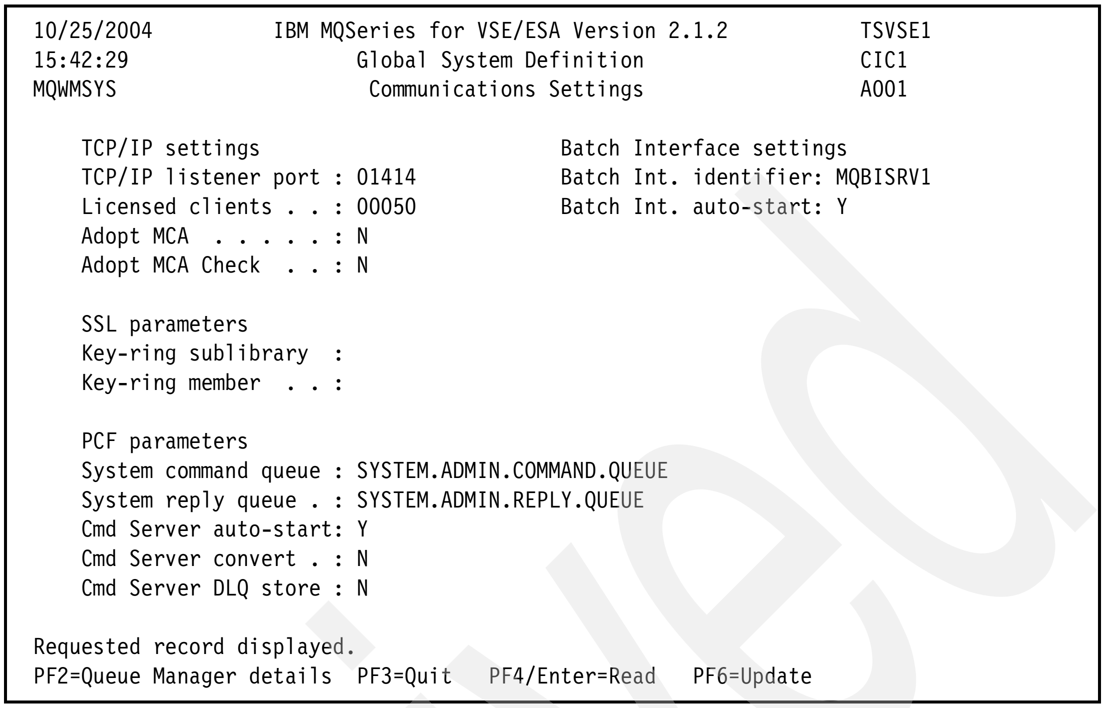

```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
15:42:29                 Global System Definition                  CIC1
MQWMSYS                   Communications Settings                  A001

    TCP/IP settings                         Batch Interface settings
    TCP/IP listener port : 01414            Batch Int. identifier: MQBISRV1
    Licensed clients . . : 00050            Batch Int. auto-start: Y
    Adopt MCA  . . . . . : N
    Adopt MCA Check  . . : N

    SSL parameters
    Key-ring sublibrary  :
    Key-ring member  . . :

    PCF parameters
    System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
    System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
    Cmd Server auto-start: Y
    Cmd Server convert . : N
    Cmd Server DLQ store : N

Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

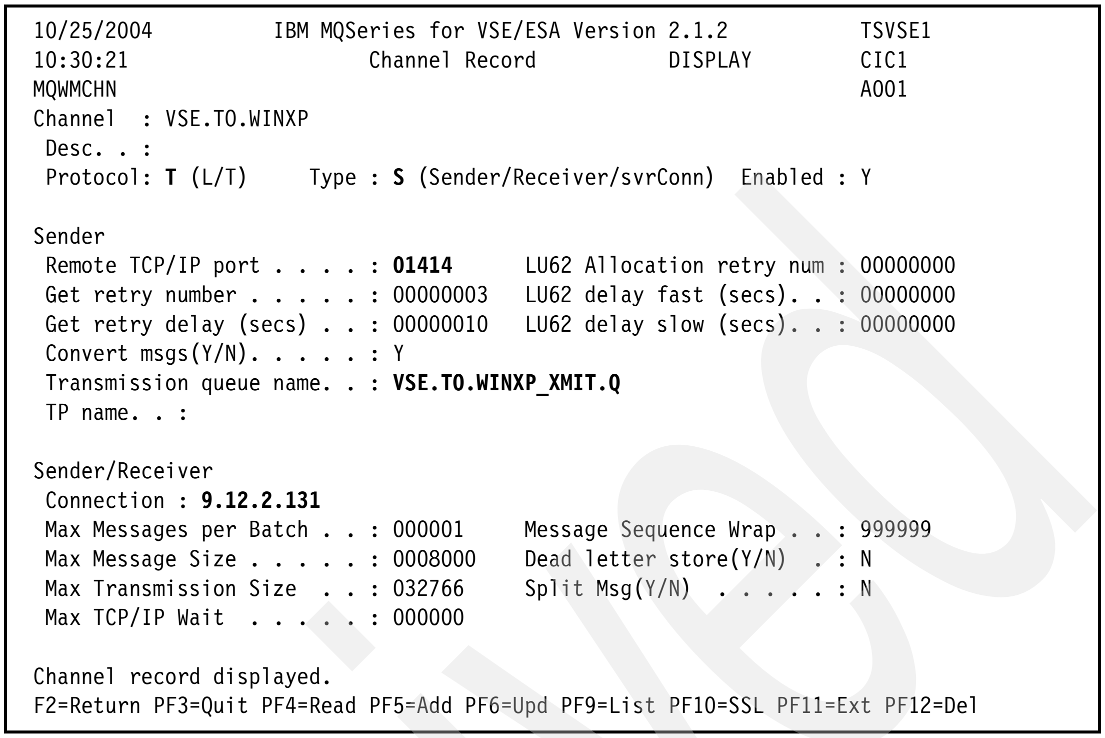
Figure A-27 Queue manager communication settings for VSE and Windows XP

In this definition, there are no more comments to add to what we already explained in 2.1, “Global System Definition (GSD)” on page 29. The important parameter for TCP/IP communication from Windows XP to VSE is the TCP/IP Listener Port number. In our example, we use 1414.

VSE to Windows XP sender channel definition

In this example, we are working in a peer-to-peer mode (for client/server mode refer to “Windows XP client to VSE distributed example” on page 244). Windows communication is to take place between two Message Channel Agents (MCAs). If we want to exchange messages in both directions, two channels need to be defined.

Figure A-28 on page 236 shows an example of our VSE sender channel definition.



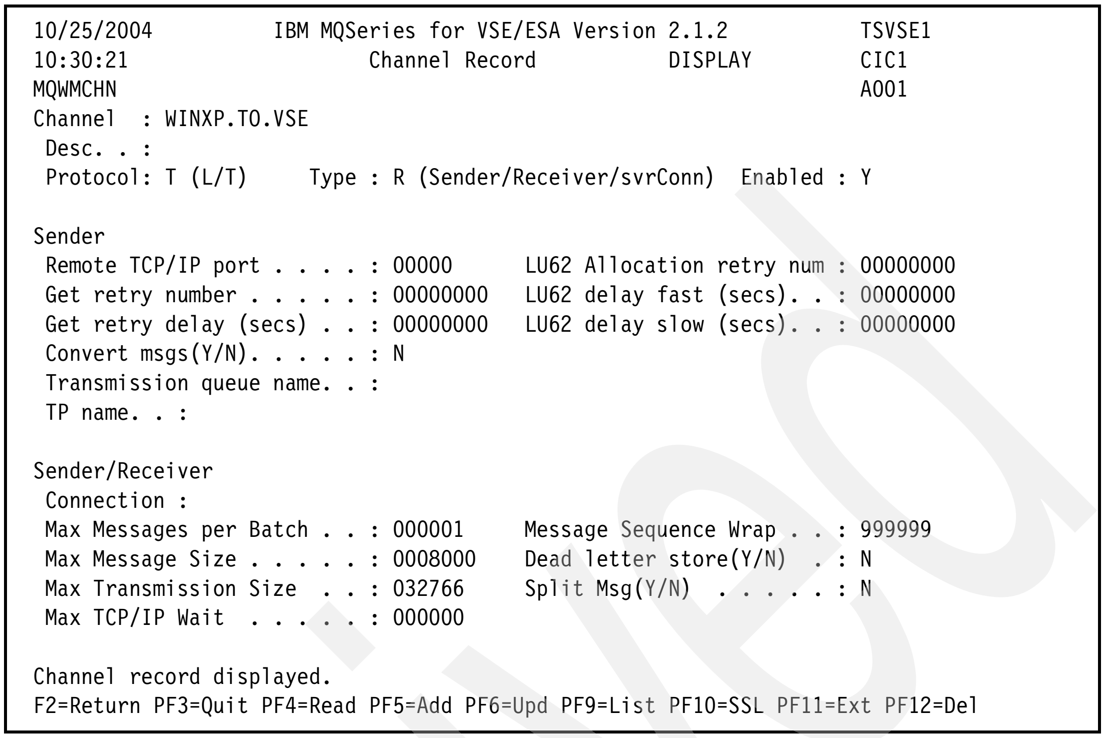
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:30:21                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A001
Channel  : VSE.TO.WINXP
 Desc. . :
 Protocol: T (L/T)     Type : S (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 01414      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000003   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000010   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : Y
 Transmission queue name. . : VSE.TO.WINXP_XMIT.Q
 TP name. . :

Sender/Receiver
 Connection : 9.12.2.131
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0008000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-28 Sender channel definition for VSE to Windows XP

This example shows the sender channel characteristics. Fields specific to TCP/IP channels have been highlighted. You may notice that the Connection field contains the IP address. The Port refers to the port number that identifies the Windows XP MQSeries queue manager.

Windows XP to VSE receiver channel definition

An example of our VSE receiver channel definition is shown in Figure A-29 on page 237.



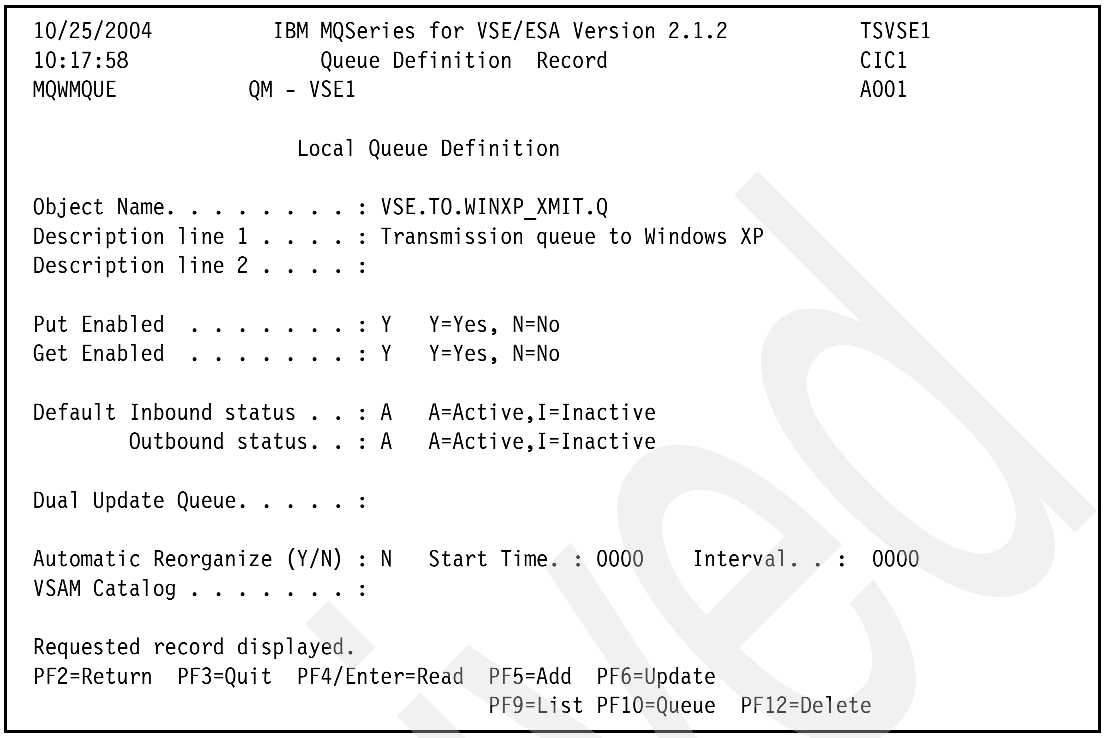
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSVSE1
10:30:21                   Channel Record          DISPLAY        CIC1
MQWMCHN                                                           A001
Channel  : WINXP.TO.VSE
 Desc. . :
 Protocol: T (L/T)     Type : R (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000000   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000000   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0008000    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-29 Receiver channel definition for Windows XP to VSE

Once again, the receiver channel matches a sender channel under Windows XP. The name and characteristics must match the Windows XP sender channel.

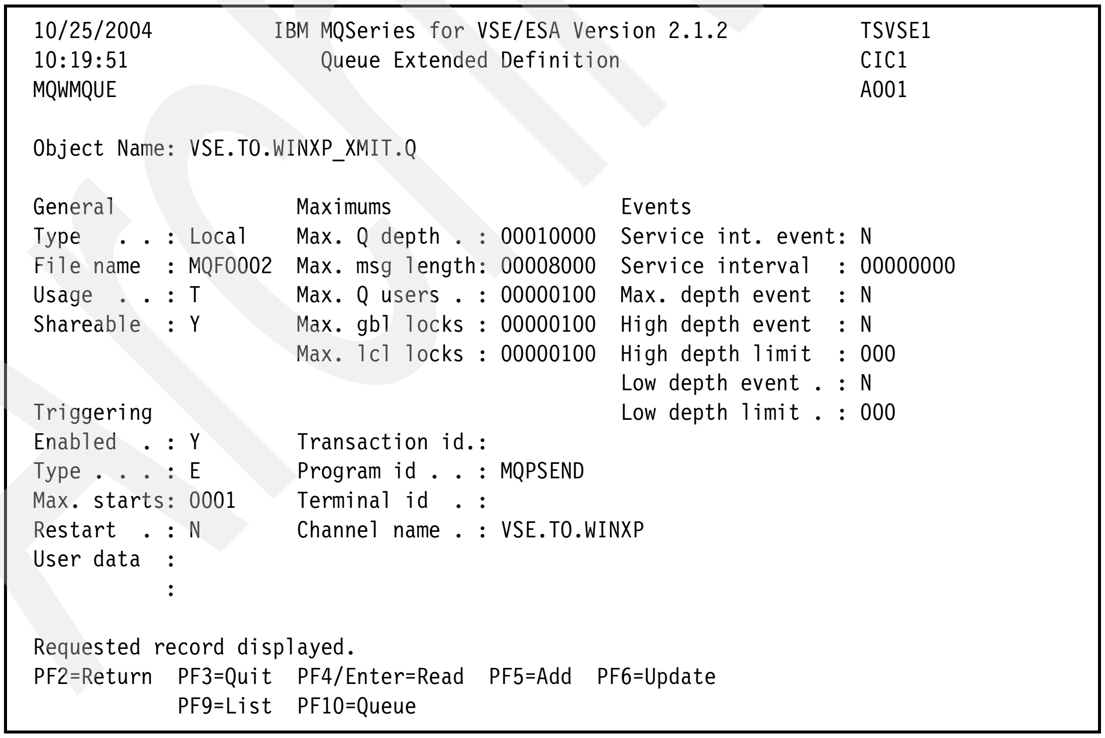
VSE to Windows XP transmission queue definition

An example of our VSE transmission queue definition is shown in Figure A-30 on page 238.



```
10/25/2004          IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:17:58                Queue Definition  Record                    CIC1
MQWMQUE           QM - VSE1                                         A001

                      Local Queue Definition

Object Name. . . . . . . . : VSE.TO.WINXP_XMIT.Q
Description line 1 . . . . : Transmission queue to Windows XP
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-30   Transmission Queue Definition for VSE to Windows XP



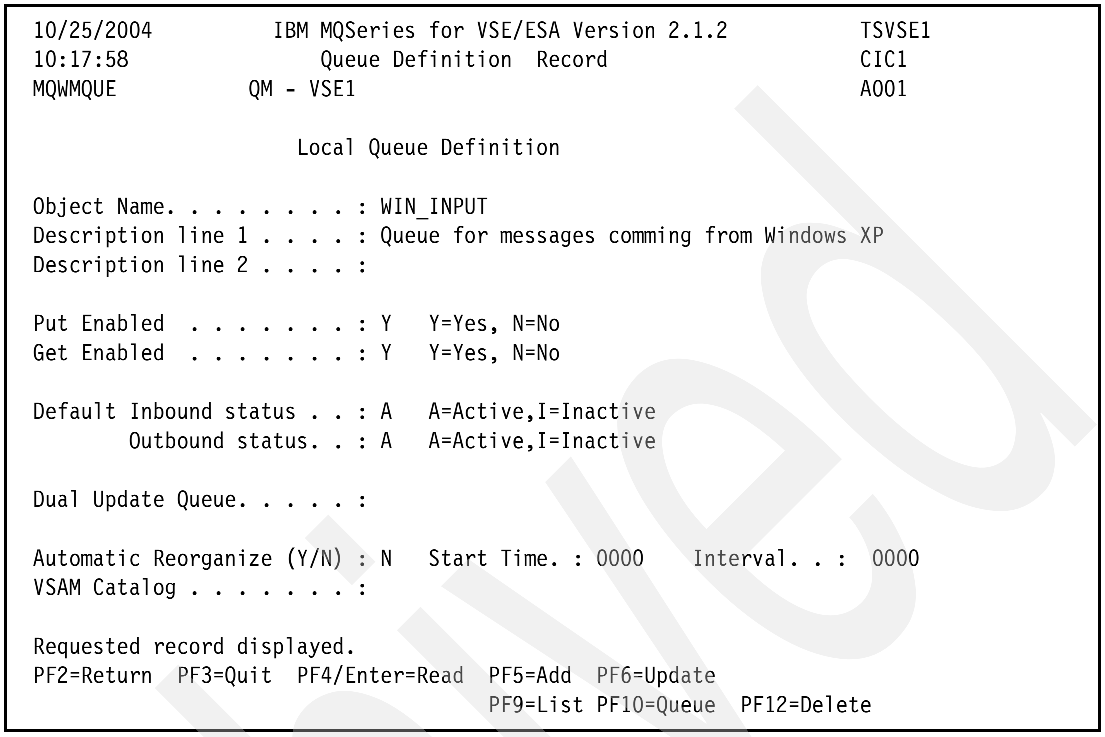
```
10/25/2004          IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51                Queue Extended Definition                   CIC1
MQWMQUE                                                             A001

Object Name: VSE.TO.WINXP_XMIT.Q

General               Maximums                 Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQF0002  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : T        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.:
Type . . . : E        Program id . . : MQPSEND
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . : VSE.TO.WINXP
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-31   Transmission Queue Extended Definition for VSE to Windows XP

The transmission queue identifies the channel to use when transmitting messages from VSE to Windows XP.

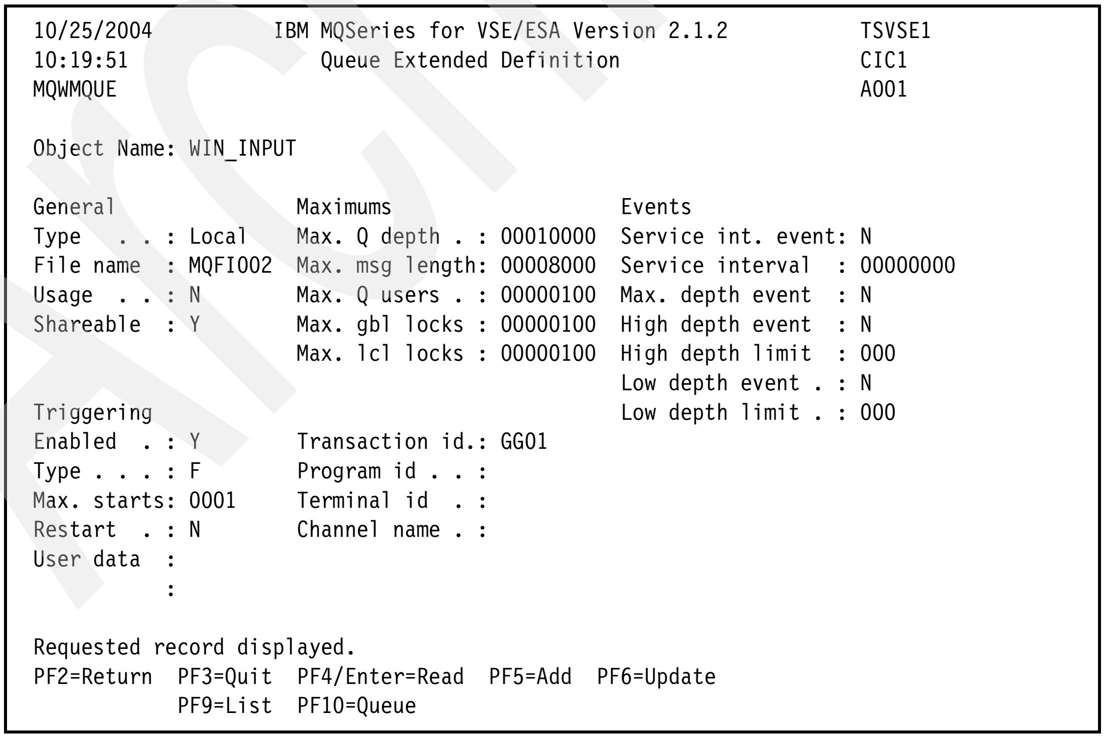
Windows XP to VSE local queue definition

An example of our VSE local queue definition is shown in Figure A-32.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:17:58               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE1                                         A001

                     Local Queue Definition

Object Name. . . . . . . . : WIN_INPUT
Description line 1 . . . . : Queue for messages comming from Windows XP
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-32   Local Queue Definition for Windows XP to VSE distributed queuing



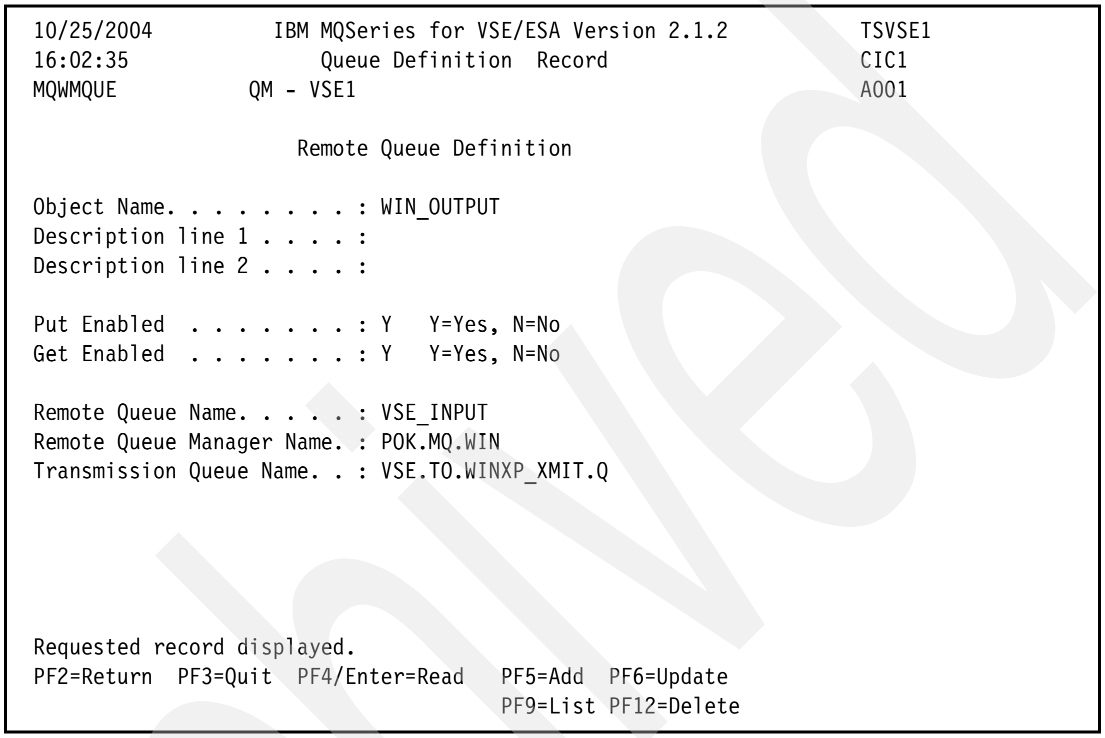
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: WIN_INPUT

General               Maximums                  Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI002  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.: GG01
Type . . . : F        Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-33   Local Queue Extended Definition for Windows XP to VSE

This queue is filled up with messages coming from Windows XP. When the Queue Depth passes from zero to at least one, the transaction GG01 is started. Since Trigger Type is set to F, other messages will not trigger a new transaction.

VSE to Windows XP remote queue definition

An example of our VSE remote queue definition is shown in Figure A-34.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
16:02:35               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE1                                         A001

                     Remote Queue Definition

Object Name. . . . . . . . : WIN_OUTPUT
Description line 1 . . . . :
Description line 2 . . . . :

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Remote Queue Name. . . . . : VSE_INPUT
Remote Queue Manager Name. : POK.MQ.WIN
Transmission Queue Name. . : VSE.TO.WINXP_XMIT.Q





Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read   PF5=Add  PF6=Update
                                       PF9=List PF12=Delete
```

Figure A-34 Remote Queue Definition for VSE to Windows XP

The remote queue identifies a local queue on Windows XP. It also identifies the transmission queue for messages in-transit to Windows XP.

Windows XP definitions

There are several ways to create and modify MQSeries objects on Windows XP. You can use the interactive dialogs available with the WebSphere MQ Explorer product. Alternatively, you can use utility programs accessible from a command prompt. For our example, we will use the utility programs.

Queue manager creation

You can create a queue manager under Windows XP as follows:

1. Open a session by clicking the command prompt icon.
2. Enter the command:

   ```
   crtmqm -q qmanager_name
   ```

   In our example, we are creating a queue manager called POK.MQ.WIN.

   This command creates a directory structure for this queue manager. You may have multiple queue managers on the same Windows XP system.

MQSeries object definitions

As we did for MQSeries for VSE, we need to define five objects. Remember that definitions are case sensitive. For Windows XP, we need to define the following objects:

- One sender and one receiver channel. They *must* have the same name as defined under VSE. For example, WINXP.TO.VSE and VSE.TO.WINXP.
- A transmission queue called WINXP.TO.VSE_XMIT.Q.
- A local queue to receive incoming messages called VSE_INPUT.
- A remote queue called VSE_OUTPUT.

We used WordPad to type in all the following definitions and save them in a file called pokwin.def. This name is arbitrary.

```
*******************************************************************/
*                                                                 */
* Definitions of Queues and Channels for POK.QM.WIN Manager       */
*                                                                 */
*******************************************************************/



*
*  Sender channel
*

   DEFINE CHANNEL  ('WINXP.TO.VSE')           +      (1)
          CHLTYPE  (SDR)                      +      (2)
          CONNAME  (1.2.3.4)                  +      (3)
          XMITQ    ('WINXP.TO.VSE_XMIT.Q')    +      (4)
          TRPTYPE  (TCP)                      +      (5)
          DESCR('Outbound channel to VSE')    +
          BATCHSZ(1)                          +
          SEQWRAP(999999)                     +
          CONVERT(YES)                               (6)

*
*  Receiver channel
*
   DEFINE CHANNEL  ('VSE.TO.WINXP')           +      (7)
          CHLTYPE  (RCVR)                     +      (8)
          DESCR('Inbound channel from VSE')   +
          SEQWRAP  (999999)                   +
          TRPTYPE  (TCP)                             (5)

*
*  Transmission queue to send messages to VSE.
*
   DEFINE QLOCAL ('WINXP.TO.VSE_XMIT.Q') REPLACE +     (9)
          DESCR  (Transmission Q to VSE')        +
          USAGE  (XMITQ)                               (10)

*
*  Input queue for incoming VSE messages.
*
    DEFINE QLOCAL   (VSE_INPUT) REPLACE          +     (11)
           DESCR    ('Queue for VSE Incoming messages')


*
*  Remote queue for outgoing messages to VSE.
```

```
*
DEFINE QREMOTE     ('VSE_OUTPUT) REPLACE        +     (12)
           DESCR   ('Outgoing messages to VSE')  +
           XMITQ   ('WINXP.TO.VSE_XMIT.Q')      +     (13)
           RNAME   ('WIN_INPUT')                +     (14)
           RQMNAME ('VSE1')                     +     (15)
           SCOPE   (QMGR)
```

Please refer to the MQSeries documentation for detailed explanations on these parameters. The following points correspond to the values shown in parenthesis on the right side of the definitions:

1. Name of the sender channel. It must be the same as the VSE receiver channel.
2. Channel type. It should be SDR for a sender.
3. CONNAME is the IP address of VSE.
4. Name of your local transmission queue. It should also be specified with the remote queue.
5. Specifies the protocol. For TCP/IP, it is TCP.
6. The CONVERT option translates the data portion of messages to the receiver code page before messages are sent. This is possible for our example because the data content is text only. Care should be taken when converting binary values.
7. Name of the receiver channel. It must be the same as the VSE sender channel.
8. Channel type. It should be RCVR for a receiver channel.
9. This is the name of the transmission queue. It is also specified when defining the sender channel and the remote queue.
10. Indicates that this local queue is a transmission queue.
11. Here you define the local queue in which messages coming from VSE will be written.
12. Specifies the name of the remote queue.
13. Specifies your transmission queue.
14. Name of the actual VSE local queue receiving your outgoing messages.
15. Name of the VSE queue manager.

After all these definitions are created and saved into a file, you need to physically configure and create these objects. First, you need to start the Windows XP queue manager from a command window as follows:

```
strmqm POK.MQ.WIN
```

This should start the queue manager. You can then create your MQSeries objects using the RUNMQSC utility. You can pipe your definitions to this utility with the following command:

```
RUNMQSC POK.MQ.WIN < pokwin.def > config.log
```

Remember, POK.MQ.WIN is the Windows XP queue manager name. This command creates the objects defined in pokwin.def and writes all output to config.log. You should check the contents of config.sys using an editor. Check your definitions if you have any errors.

Communications setup

We are now ready to start exchanging messages between VSE and Windows XP. Before we run any tests, we should check that communication is possible between the two systems.

From the VSE side, verify that MQSeries for VSE is active, and the relevant queues are in an IDLE state (use MQMT option 3).

From the Windows XP side, verify that you are able to connect to VSE by using the **ping** command. From a command window, use the following command:

```
ping 1.2.3.4
```

Where 1.2.3.4 is the IP address of your VSE system. You should also ping from VSE to Windows XP.

Once we are sure we can communicate between the two hosts using TCP/IP, we can activate the communication path by running a user defined procedure as follows:

```
mqstart.bat
```

The contents of this file may be similar to the following:

```
@echo on
Rem  Start MQSeries Queue Manager
Start STRMQM
Rem  Start the Listener channel
Start runmqlsr -t tcp -m POK.MQ.WIN
Rem  Start the sender channel
Start runmqchl -c WINXP.TO.VSE -m POK.MQ.WIN
Exit
```

The batch file starts the queue manager, the TCP/IP listener program, and the Windows XP sender channel. You may notice that we have not specified a port number when starting the TCP/IP listener program (runmqlsr). This is because we have set the default port number for `mqseries` to 1414 in the TCP/IP services file. The entry looks like the following:

```
mqseries            1414/TCP
```

Windows XP to VSE distributed test

Windows XP is shipped with a number of test programs. Among these is AMQSPUT. To send a message from Windows XP to VSE, open a command window and run the AMQSPUT utility as follows:

```
AMQSPUT VSE_OUTPUT POK.MQ.WIN
```

The AMQSPUT utility is interactive and allows you to enter test messages. Each message is sent as it is entered. A blank entry terminates the program.

To check the success of this test, first check the Q Depth of the WIN_INPUT queue on VSE using MQMT option 3. The Q Depth should match the number of messages you entered while running the AMQSPUT utility.

Secondly, use MQMT option 4 to examine the messages themselves in the WIN_INPUT queue. Because we defined CONVERT(YES), the message data should now be in EBCDIC.

If the Q Depth or messages are wrong, check the system.log on VSE and the error logs on Windows XP.

VSE to Windows XP distributed test

If you want to send a message from VSE to Windows XP, use the TST2 transaction to put a message to the remote queue WIN_OUTPUT. In native CICS, run the transaction as follows:

```
TST2 PUT 001 WIN_OUTPUT
```

This should put one test message to local queue VSE_INPUT on Windows XP. To verify the success of this test, start a command window on Windows XP and enter the following command:

```
RUNMQSC pok.mq.win
```

This allows you to interactively enter the following command:

```
DISPLAY QUEUE (VSE_INPUT) CURDEPTH
```

If everything was correct, the CURDEPTH should be 1. Otherwise, check your system logs on both sides. You can also check the contents of the actual message using the AMQSBCG utility.

At this point, you are sure that the communications between VSE and your Windows XP system work correctly in both directions. You are now ready to test your own applications.

Windows XP client to VSE distributed example

Client programs can run on a system that does not have an MQSeries queue manager. This is possible because the MQI objects linked by the client program include logic necessary to establish and use inter-platform communication. From the client program perspective, the attached remote queue manager appears just like a local queue manager. Remember that client communication is only available using the TCP/IP protocol.

Since no queue manager necessarily exists on the client system, no MQSeries definitions are required for Windows XP in this example.

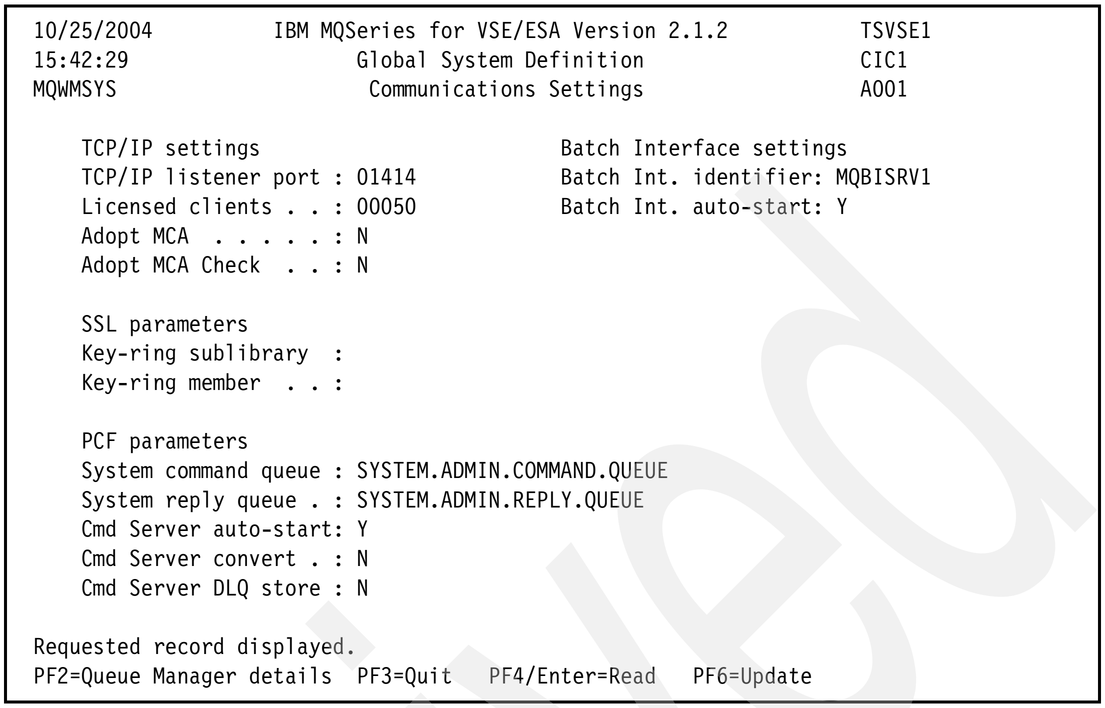
VSE definitions

In our example, we want a Windows XP client program to both put and get messages from a local queue defined to VSE. Therefore, under VSE, we need to define:

- A client channel called WINXP.TO.VSE.
- A local queue into which the client puts messages called WIN_INPUT.
- A local queue from which the client gets messages called WIN_OUTXP.

VSE global system definition

An example of the communication settings for our VSE queue manager is shown in Figure A-35 on page 245.



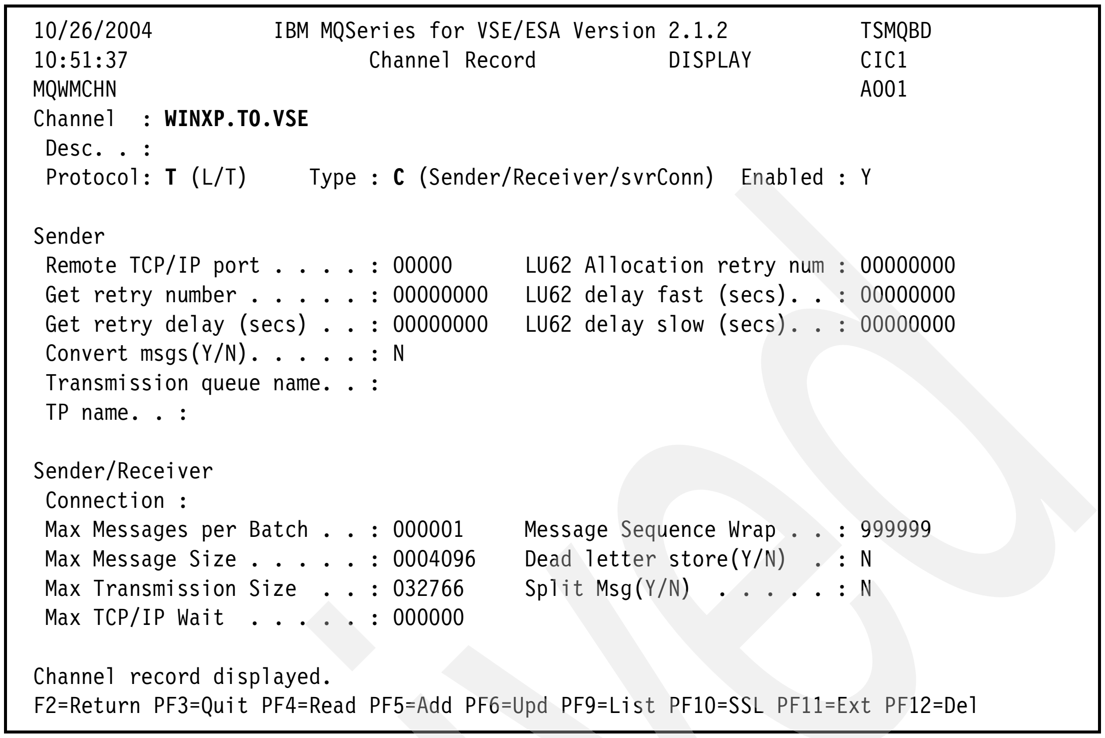
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
15:42:29                 Global System Definition                  CIC1
MQWMSYS                   Communications Settings                  A001

    TCP/IP settings                        Batch Interface settings
    TCP/IP listener port : 01414           Batch Int. identifier: MQBISRV1
    Licensed clients . . : 00050           Batch Int. auto-start: Y
    Adopt MCA  . . . . . : N
    Adopt MCA Check  . . : N

    SSL parameters
    Key-ring sublibrary  :
    Key-ring member  . . :

    PCF parameters
    System command queue : SYSTEM.ADMIN.COMMAND.QUEUE
    System reply queue . : SYSTEM.ADMIN.REPLY.QUEUE
    Cmd Server auto-start: Y
    Cmd Server convert . : N
    Cmd Server DLQ store : N

Requested record displayed.
PF2=Queue Manager details  PF3=Quit   PF4/Enter=Read   PF6=Update
```

Figure A-35 Queue manager communication settings for Windows XP client

Once again, the important parameter to note is the TCP/IP listener port. The client environment must identify both the IP address or host name of the VSE system and the port number of the MQ/VSE listener program to establish a client connection.

Windows XP client to VSE client channel definition

An example of our VSE client channel definition is shown in Figure A-36 on page 246.



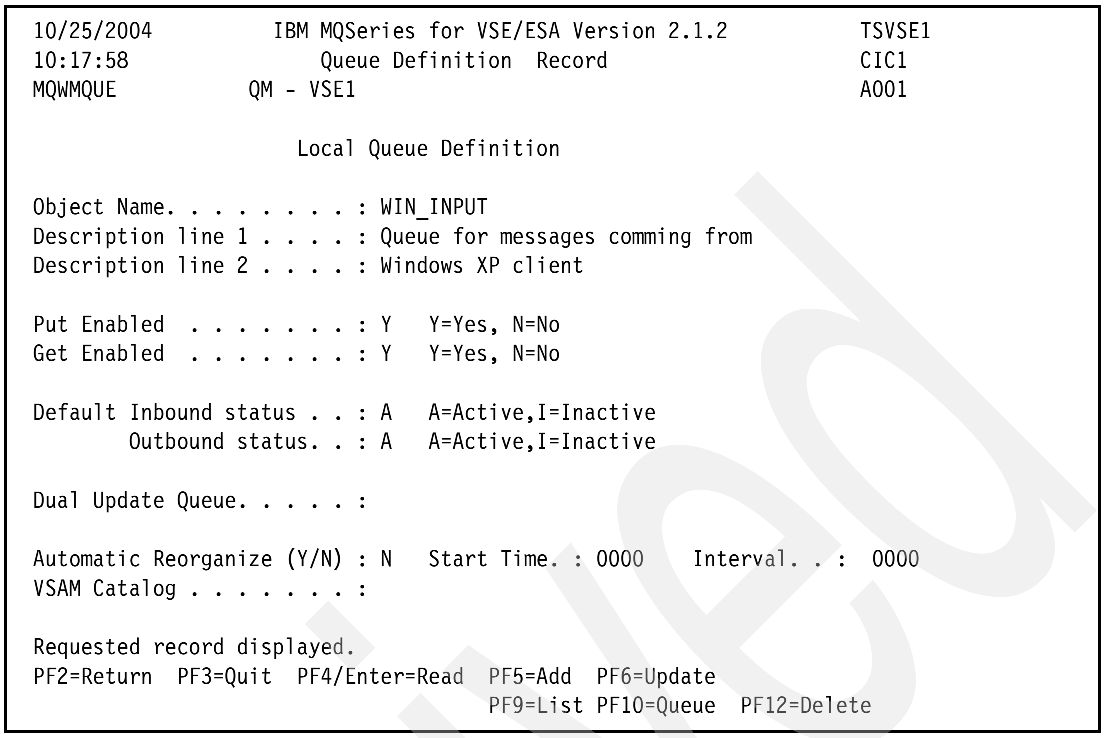
```
10/26/2004         IBM MQSeries for VSE/ESA Version 2.1.2         TSMQBD
10:51:37                   Channel Record          DISPLAY        CIC1
MQWMCHN                                                           A001
Channel  : WINXP.TO.VSE
 Desc. . :
 Protocol: T (L/T)     Type : C (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000000   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000000   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0004096    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

Figure A-36 Windows XP client to VSE client channel definition

The client channel definition is simple. Parameters specific to TCP/IP have been highlighted. You may notice that the Connection field is left blank and the Port field is left at 0. They are not needed here because the connection is always initiated by the client. In this respect, a client channel is identical to a receiver channel. In TCP/IP, each packet contains header information about the sender (for example, IP address, port, and protocol).

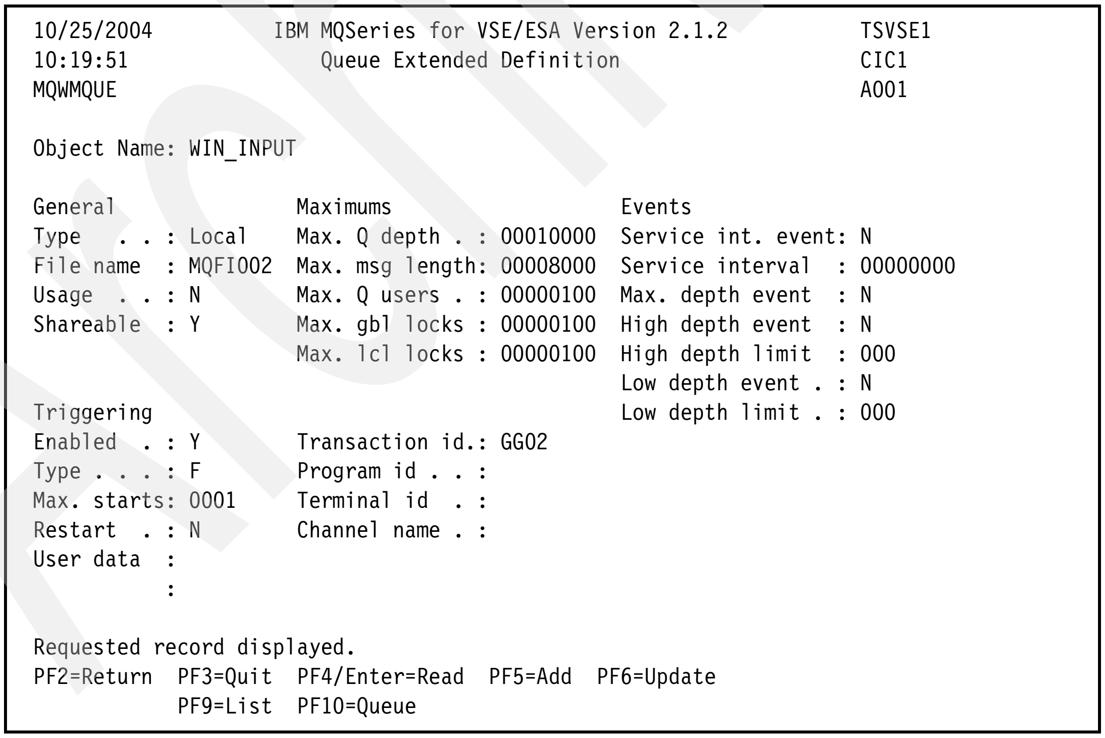
VSE local queue definition for put messages

An example of our VSE local queue for our client program to put messages is shown in Figure A-37 on page 247.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:17:58               Queue Definition  Record                    CIC1
MQWMQUE          QM - VSE1                                         A001

                      Local Queue Definition

Object Name. . . . . . . . : WIN_INPUT
Description line 1 . . . . : Queue for messages comming from
Description line 2 . . . . : Windows XP client

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-37 Local Queue Definition for Windows XP to VSE client put message



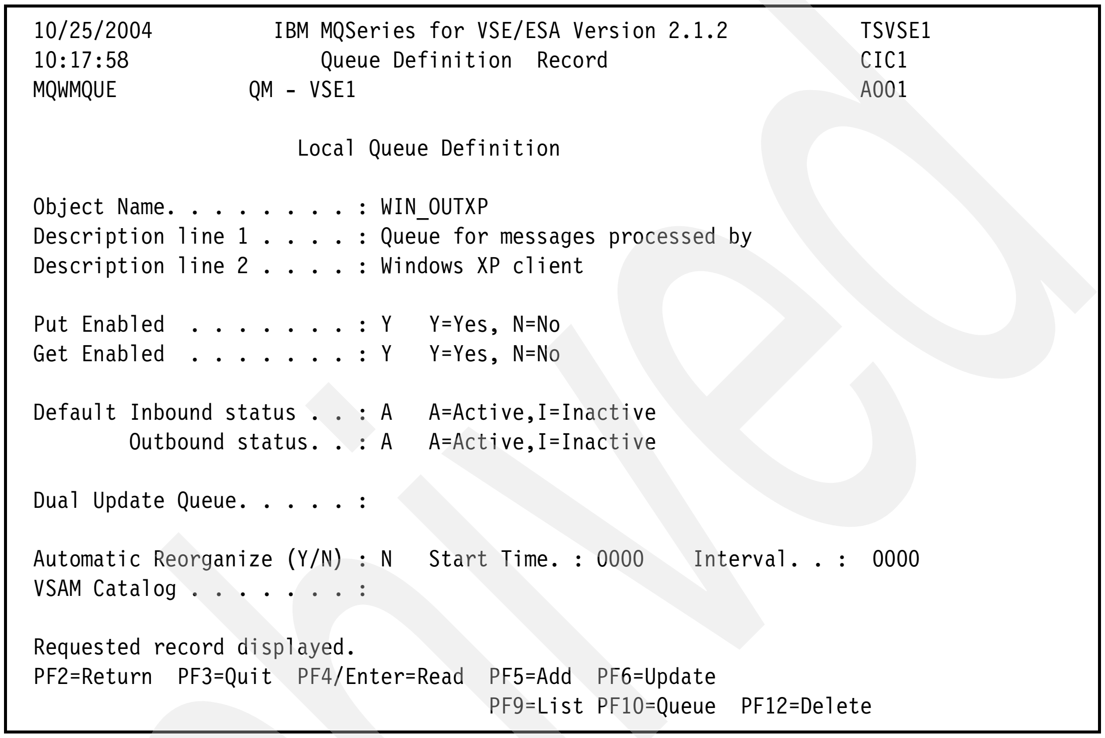
```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: WIN_INPUT

General               Maximums                    Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI002  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : Y        Transaction id.: GG02
Type . . . : F        Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-38 Queue Extended Definition for Windows XP to VSE put message

This queue is filled with messages sent by the Windows XP client. When the Q Depth passes from zero to at least one, the transaction GG02 is triggered. Since the Trigger Type is set to F, other messages will not trigger a new transaction. For our example, the GG02 transaction runs a program that puts messages to the local queue used by the Windows XP client to get messages.

VSE local queue definition for get messages

An example of our VSE local queue for our client program to get messages is shown in Figure A-39.



```
10/25/2004          IBM MQSeries for VSE/ESA Version 2.1.2           TSVSE1
10:17:58                Queue Definition  Record                     CIC1
MQWMQUE          QM - VSE1                                           A001

                      Local Queue Definition

Object Name. . . . . . . . : WIN_OUTXP
Description line 1 . . . . : Queue for messages processed by
Description line 2 . . . . : Windows XP client

Put Enabled  . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . : A   A=Active,I=Inactive
        Outbound status. . : A   A=Active,I=Inactive

Dual Update Queue. . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-39 Local Queue Definition for Windows XP to VSE client get message

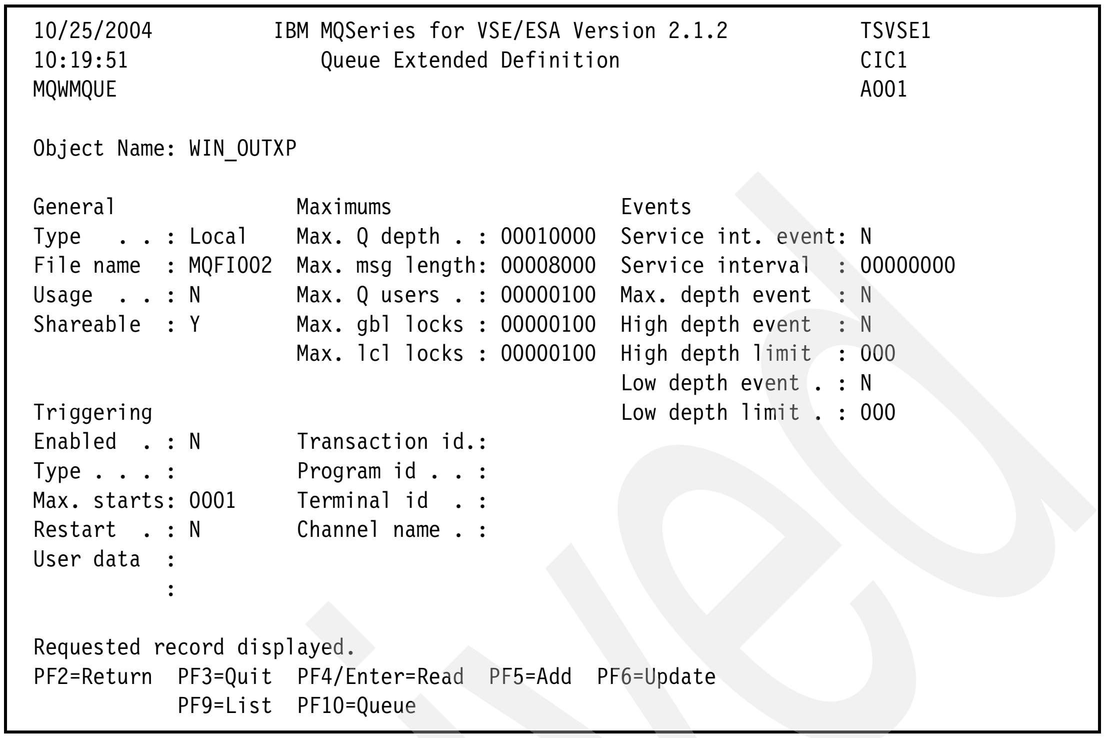

```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: WIN_OUTXP

General              Maximums                    Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI002  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : N        Transaction id.:
Type . . . :          Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-40 Queue Extended Definition for Windows XP to VSE get message

In our example, this queue is filled up by transaction GG02 so that the client can issue MQGETs against it. Transaction GG02 is triggered when the client program puts messages to the WIN_INPUT local queue.

Windows XP definitions

As we previously said, no MQSeries definitions are required on the client system. However, we must at least provide information to allow the client to communicate with the server on the VSE system. We do this by using environment variables as follows:

```
SET MQCCSID=850
SET MQSERVER=WINXP.TO.VSE/TCP/1.2.3.4(1414)
```

The MQCCSID variable identifies the local code page of the MQSeries client. Code page conversion for clients is handled by the server. In the case of a VSE server that uses Language Environment/VSE services, you must ensure the code page you select for the client can be translated to and from the code page of the server. The server code page is specified in the global system definition. For more details on code page conversion, refer to 6.6.1, “Client code page conversion” on page 119.

The MQSERVER variable identifies the client channel on the server system, the protocol to use, and the IP address and port number of the server.

In this example we are using MQ utility programs to test the client function. If you write your own client program using the MQ classes for Java, you do not need to set the environment variables. Instead, the Java classes provide the MQEnvironment class which allows you to set equivalent variables. Specifically:

```
MQEnvironment.channel  = "WINXP.TO.VSE";
MQEnvironment.hostname = "1.2.3.4";
MQEnvironment.port     = 1414;
MQEnvironment.CCSID    = 850;
```

Windows XP to VSE client test

Prior to running our test, we should check that we can ping the VSE system from our Windows XP system.

Like all MQSeries systems, Web sphere MQ for XP comes with a set of test programs. These include AMQSPUTC and AMQSGETC, both of which are client programs to put and get messages respectively.

To put a message to local queue WIN_INPUT on our VSE system, start a command window on Windows XP and issue the following commands:

```
CD \MQM\BIN
AMQSPUTC WIN_INPUT VSE1
```

Note that your sample programs may be in a directory other than \MQM\BIN. Also note that command windows work as a shell, which means the MQCCSID and MQSERVER environment variables must be defined in this command window before you run the AMQSPUTC utility.

When running the AMQSPUTC utility, we specify the target queue and the VSE queue manager. These parameters must match an actual queue manager defined at the IP address identified by the MQSERVER variable. They must also identify a queue defined to that queue manager. For our example, this is the case.

The AMQSPUTC utility is interactive and allows you to enter test messages. Each messages is sent as it is entered. A blank message terminates the utility.

After running the utility, we can check the VSE side to see whether the messages have been written to the WIN_INPUT local queue. We can use MQMT option 4 to look at the VSE queue. We can also check the Q Depth using MQMT option 3.1.

Because we have a trigger (GG02) that reads messages off WIN_INPUT and puts them on WIN_OUTXP, when we check the messages themselves, we should notice they are in a *deleted* state (because they have been processed). Do not be surprised if these messages looks hieroglyphic. This is because normally the data part of a message needs a code page conversion by the application. The data appears hieroglyphic because it is being viewed on an EBCDIC machine.

Having populated the WIN_OUTXP local queue, we can now use the client utility AMQSGETC to get messages from this queue. From the same Windows XP command window we used to put client messages, we can enter the following command:

```
AMQSGETC WIN_OUTXP VSE1
```

AMQSGETC displays the received messages on the screen. Since the message data remains in ASCII throughout our test, it is readable again when it arrives back at the client system.

VM/CMS client to VSE distributed example

As already stated, client programs can run on a system that does not have an MQSeries queue manager. From the client program perspective, the attached remote queue manager appears just like a local queue manager. Once again, client communication with MQSeries for VSE 2.1 is only available using the TCP/IP protocol.

As with the Windows XP client, no MQSeries definitions are required for VM/CMS in this example.

Remember that MQSeries client has been a standard feature of VM since VM/ESA 2.3.

VSE definitions

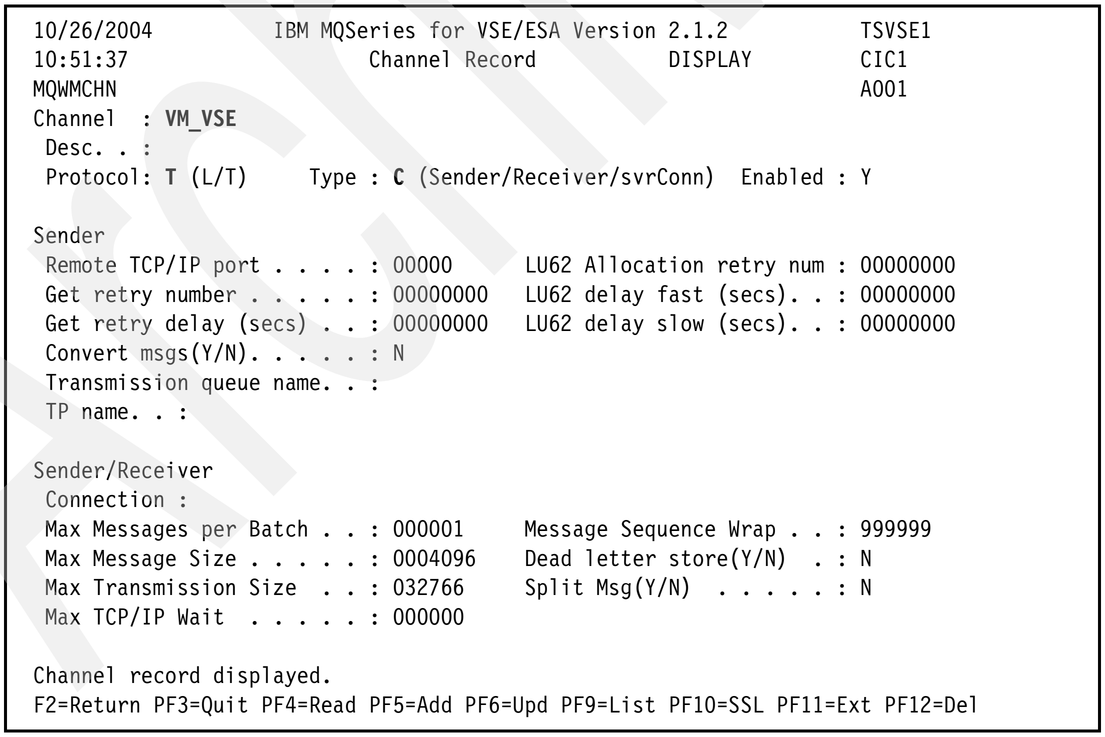
In this example, we want an MQSeries CMS client to write messages to a VSE queue and read messages back from the same queue. Therefore, under VSE, we need to define:

- A client channel called VM_VSE
- A local queue to both put and get messages on VSE

In our tests, we use the sample queue ANYQ as both our input and output queue for the client. As in our previous examples, the global system definition on VSE should identify a port number (we use 1414) and licensed clients.

VM/CMS client to VSE client channel definition

An example of our VSE client channel definition is shown in Figure A-41.



```
10/26/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:51:37                   Channel Record          DISPLAY         CIC1
MQWMCHN                                                            A001
Channel  : VM_VSE
 Desc. . :
 Protocol: T (L/T)     Type : C (Sender/Receiver/svrConn)  Enabled : Y

Sender
 Remote TCP/IP port . . . . : 00000      LU62 Allocation retry num : 00000000
 Get retry number . . . . . : 00000000   LU62 delay fast (secs). . : 00000000
 Get retry delay (secs) . . : 00000000   LU62 delay slow (secs). . : 00000000
 Convert msgs(Y/N). . . . . : N
 Transmission queue name. . :
 TP name. . :

Sender/Receiver
 Connection :
 Max Messages per Batch . . : 000001     Message Sequence Wrap . . : 999999
 Max Message Size . . . . . : 0004096    Dead letter store(Y/N)  . : N
 Max Transmission Size  . . : 032766     Split Msg(Y/N)  . . . . . : N
 Max TCP/IP Wait  . . . . . : 000000

Channel record displayed.
F2=Return PF3=Quit PF4=Read PF5=Add PF6=Upd PF9=List PF10=SSL PF11=Ext PF12=Del
```

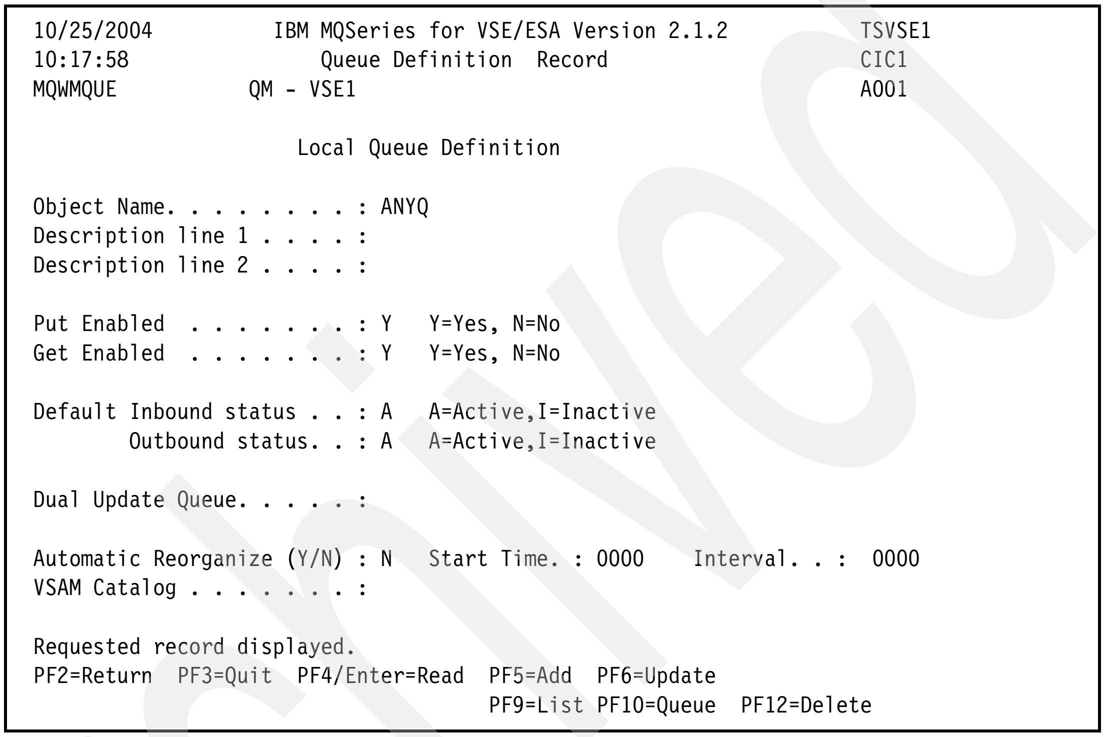
Figure A-41 VM/CMS client to VSE client channel definition

This definition specifies the client channel characteristics. Parameters specific to TCP/IP have been highlighted. You may notice that the Connection and Port fields are blank and 0 respectively. They are not needed here because the connection is always initiated by the client. In TCP/IP, each packet contains in header information about the sender (for example, IP address, port, and protocol).

If you follow the instructions for verifying your MQSeries system on VSE, you would have created a sample queue called ANYQ. For convenience, we repeat the queue definition here.

VSE local queue definition for get and put messages

An example of our VSE sample queue, ANYQ, is shown in Figure A-42.



```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2           TSVSE1
10:17:58               Queue Definition  Record                     CIC1
MQWMQUE           QM - VSE1                                         A001

                     Local Queue Definition

Object Name. . . . . . . . . : ANYQ
Description line 1 . . . . . :
Description line 2 . . . . . :

Put Enabled  . . . . . . . . : Y   Y=Yes, N=No
Get Enabled  . . . . . . . . : Y   Y=Yes, N=No

Default Inbound status . . . : A   A=Active,I=Inactive
        Outbound status. . . : A   A=Active,I=Inactive

Dual Update Queue. . . . . . :

Automatic Reorganize (Y/N) : N   Start Time. : 0000    Interval. . :  0000
VSAM Catalog . . . . . . . . :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
                                      PF9=List PF10=Queue  PF12=Delete
```

Figure A-42 VM/CMS client to VSE Local Queue Definition sample

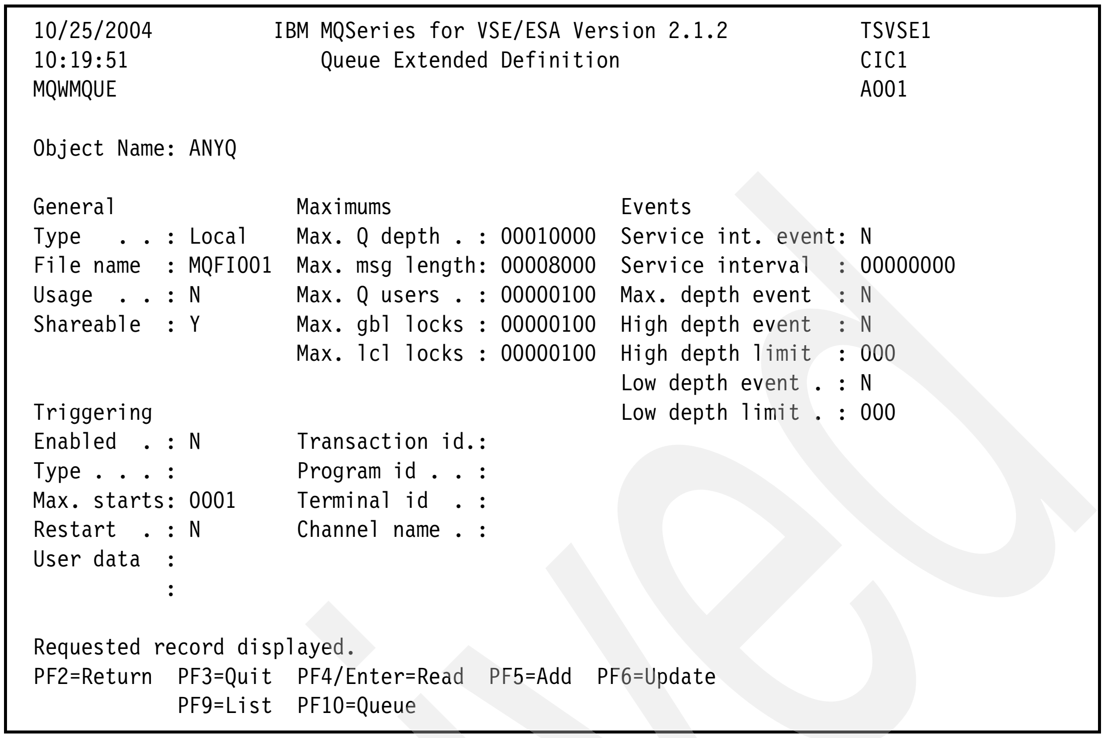

```
10/25/2004         IBM MQSeries for VSE/ESA Version 2.1.2          TSVSE1
10:19:51               Queue Extended Definition                   CIC1
MQWMQUE                                                            A001

Object Name: ANYQ

General               Maximums                    Events
Type   . . : Local    Max. Q depth . : 00010000  Service int. event: N
File name  : MQFI001  Max. msg length: 00008000  Service interval  : 00000000
Usage  . . : N        Max. Q users . : 00000100  Max. depth event  : N
Shareable  : Y        Max. gbl locks : 00000100  High depth event  : N
                      Max. lcl locks : 00000100  High depth limit  : 000
                                                 Low depth event . : N
Triggering                                       Low depth limit . : 000
Enabled  . : N        Transaction id.:
Type . . . :          Program id . . :
Max. starts: 0001     Terminal id  . :
Restart  . : N        Channel name . :
User data  :
           :

Requested record displayed.
PF2=Return  PF3=Quit  PF4/Enter=Read  PF5=Add  PF6=Update
            PF9=List  PF10=Queue
```

Figure A-43 VM/CMS client to VSE Extended Local Queue Definition sample

VM/CMS definitions

As we already mentioned, we do not need to define any MQSeries objects on the client side. However, we must at least provide information to allow the client to communicate with the server on the VSE system. You do this by using environment variables put in a small EXEC procedure as follows:

```
/*                                                                    */
"CP LINK MAINT 193 193 RR"          /* Contains MQSeries LIbraries    */
"ACCESS 193 R"                      /* access this Minidisk           */
"GLOBAL LOADLIB SCEERUN AMQLLIB"    /* LE and MQSeries libraries      */
vse_ip         = "1.2.3.4"          /* VSE IP address                 */
vse_port       = "1414"             /* VSE MQseries TCP.IP port       */
qm             = "VSE1"             /* Queue Manager name             */
cli_chn        = "VM_VSE"           /* MQSeries Client Channel name   */
mqserver = cli_chn"/TCP/"vse_ip"("vse_port")"    /* build env.  string*/
"GLOBALV SELECT CENV SETLP MQSERVER" mqserver
"GLOBALV SELECT CENV SETLP MQ_USER_ID" userid()
"GLOBALV SELECT CENV SETLP MQTRACE MQCLIENT.TRACE.A"
```

Note that the variable MQTRACE refers to the CMS file, MQCLIENT.TRACE.A. It is optional. If used, all MQSeries MQI calls are logged to this file. Therefore, be careful in a production environment, you may quickly run out of disk space and impair performance.

The queue manager name (qm) is not used to set the MQSeries environment. It will be used only with MQCONN calls.

VM/CMS to VSE client test

The best way to test an MQSeries client under CMS is by using a REXX procedure. See "REXX client program" on page 260 for the complete code.

In this test program, we:

1. Write 10 messages to ANYQ. Then, prompt the user to check the Q Depth.
2. Reread the queue until there are no messages left. Then, prompt the user to checking the Q Depth again. It should be zero.

The program starts by setting all environment variables as previously explained. In addition, it loads in storage the MQSeries REXX Interface module RXMQV. For example:

```
"NUCXLOAD RXMQV (SYSTEM"              /* make module resident for   */
                                      /* REXX MQSeries  interface   */
```

The program then sets the MQSeries REXX environment with the INIT function and connects to the queue manager:

```
rcx  = RXMQV("INIT")
If word(rcx,1)<> 0 then
  Do
      "CP MSG" admin "MQ/Series Interface Initialization failure."
      Exit 8
  End
rcx   = RXMQV("CONN",qm)        /* qm is the queue manager name */
x= word(rcx,1)
If x<> 0 & x<>2002  then        /* 2002 means already connected */
  Do
      "CP MSG" admin "Connection to Q-Manager failed."
      Exit 8
  End
```

After INIT and CONN, many stems or variables are created and set. They are used with all further RXMQV calls.

As with VSE application programs, before issuing the MQPUT, you need to set the options. You do this by setting the imd.* stem variables. For example, imd.MSGID. You are free to choose the stem name (in our case, imd), but not the subscript variable names (for example, MSGID). For example:

```
imd.PER         =  MQPER_PERSISTENT
imd.FORM        =  MQFMT_STRING
.............
Do i=1 to maxmsg              /* write 10 messages   */
   msg.1  = "This is message number" i
   msg.0  = length(msg.1)
   rcx   = RXMQV("PUT",hqn,'msg.','imd.','omd.','ipmo.','opmo.')
End
If word(rcx,1)<> 0 then Return rcx
rcx = RXMQV("CLOSE",hqn,MQCO_NONE)
If word(rcx,1)<> 0 then Return rcx
```

At this point, you can verify from the VSE side (with MQMT option 3.1) that the ANYQ queue depth is 10 (assuming it was zero before). By pressing Enter, the program continues. We are now going to read all messages from the sample queue. If there is no more messages, we wait for five seconds and leave the loop. For example:

```
oo= mqoo_input_as_q_def
rcx   = RXMQV("OPEN",q_in,oo,'hqn',ood.)
If word(rcx,1)<> 0 then
      Return rcx
Do forever                                    /*Loop until the queue   */
                                              /* is empty.             */
   g.0 = 200                                  /*  Maximum message size */
   g.1 = ""
   igmo.opt = MQGMO_WAIT + MQGMO_CONVERT
   igmo.WAIT = 05000                        /* wait up to 5 seconds    */
   rcx   = RXMQV("GET",hqn,'g.','igmd.','ogmd.','igmo.','ogmo.')
   If word(rcx,1)<> 0 then leave
   say g.1                /* Display message */
End
x= word(rcx,1)
If x<> 0 & x<>2033  then                     /* 2033 means queue empty */
   Return rcx
```

All messages are now read, and the queue depth should now be zero.

MQSeries for VM/VSE environment for e-business

There is no doubt that having a Web server on an S/390 provides you with scalability and reliability advantages. On most sites, in VM/VSE environments, the legacy data is usually located on the VSE side, either in native VSAM files or in databases, such as DB/2, DATACOM, or ADABAS. VM/ESA is being used to provide hypervisor capability, development, and info center environments.

With the emerging e-business market, it may be valuable to have a Web server under VM/ESA and keep the legacy data on VSE. Today, most VM Web servers provide you with functions for revamping 3270 panels of existing applications, but for new applications, MQSeries may be a good vehicle for transferring data between the Web server (CMS) and VSE.

We have set up such an environment, but it would be beyond the scope of this document to describe it in detail or provide all the code we have written. However, we would like to explain a logical overview of what we have achieved. It is depicted in the diagram shown in Figure A-44 on page 256.

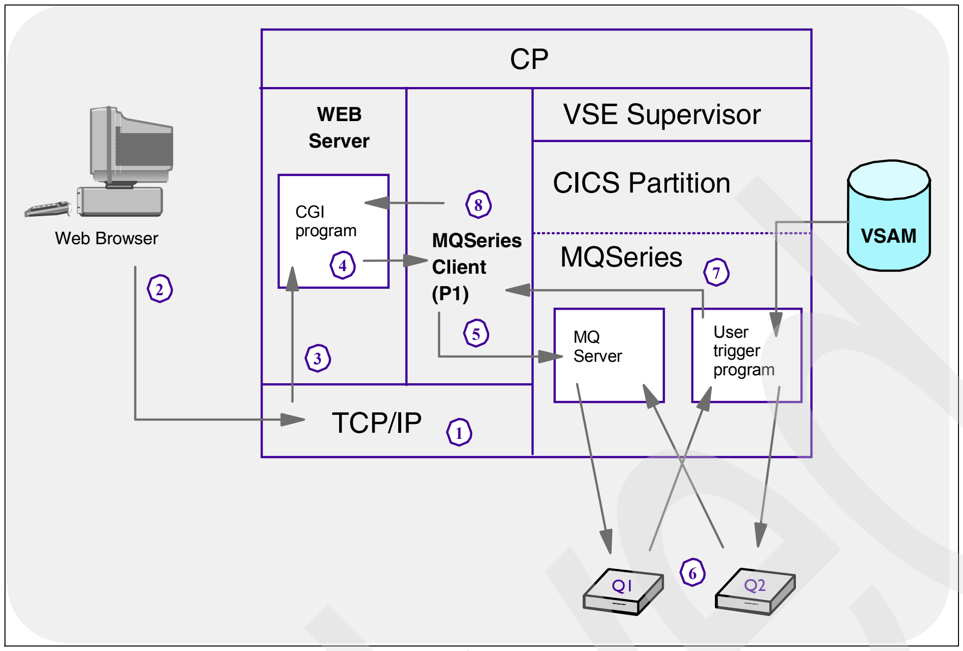


Figure A-44 MQSeries VM/VSE in a Web environment

The scenario depicted in this diagram is as follows:

1. The communication protocol used is TCP/IP via a Virtual Channel-to-Channel Adapter (VCTCA) between TCP/IP/VM and the VSE virtual machines.
2. A request is issued from a workstation connected to an intranet by a user working with a Web browser.
3. This request is received by a Web server running under VM and passed to a Common Gateway Interface (CGI) program.
4. This CGI transfers this request to a CMS program P1 running in another virtual machine (it could also have been a subroutine of this CGI program).
5. P1 is written in REXX and uses the client MQSeries/REXX API. Its objective is to retrieve, from a VSAM file dedicated to the CICS partition, all information related to an employee of this enterprise. Basically, it issues the following commands:
   - MQPUT to send a message to the message queue Q1. This message includes the employee serial number.
   - MQGET to read feedback from the server. The program is, thus, waiting for an incoming message from message queue Q2.

   Indeed, from a performance point of view, the application would run faster if coded, for example, in COBOL or C.
6. When a message is written into queue Q1, it triggers a CICS transaction program that issues an MQGET against Q1, validates the serial number, and retrieves the related record from the VSAM file. A message is created from the selected record and sent to Q2.

   We used a VSAM file for simplicity; however, it could have been, for example, a database table in DB/2.
7. The client program wakes up (since it was still waiting for a message from Q2).
8. The message is passed back to the CGI program, analyzed, and formatted into an HTML page that is sent to the workstation in response to the user request. (In this HTML page, we added pictures and Java applets.)

MQSeries client for Java

The MQSeries classes for Java allow a program written in the Java programming language to connect to MQSeries as an MQSeries client using TCP/IP, or directly to an MQSeries server using the Java Native Interface (JNI). They allow Java applets, applications, and servlets access to the messaging and queuing services of MQSeries. If the client-style connection is used, no additional MQSeries code is required on the client machine. The MQSeries classes for Java enable a message-based approach to application integration using Java.

Note: The MQSeries Client for Java product extension is available with the IBM MA88 SupportPac, which can be downloaded from:

`http://www.ibm.com/software/integration/support/supportpacs/`

Before downloading, check whether this product extension is already available with your WebSphere MQ product.

B

Sample programs

In this appendix, we provide a number of sample programs in a variety of languages. These samples are for reference only and may help you when writing your own programs.

REXX client program

```
/*-------------------------------------------------------------------*/
/*                    MQSeries Client test program                   */
/*                    ----------------------------                   */
/*                                                                   */
/*  This is a test program to allow a CMS user to send messages to   */
/* an MQSeries server.  Here we are talking to a MQSeries/VSE system  */
/* but it could also be used on any other platform.                  */
/*                                                                   */
/* The scenario of this procedure is the following:                  */
/*                                                                   */
/*    1.  Set the MQSeries environment by defining global variables  */
/*    2.  INIT and connect this application to the MQSeries Server   */
/*    3.  Send messages to a specified queue                         */
/*    4.  Ask the user to check from the VSE side if these messages  */
/*        have been written. (check Queue Depth                      */
/*    5.  Re-read those messages.                                    */
/*    6.  Disconnect the application from the MQseries Server        */
/*                                                                   */
/*-------------------------------------------------------------------*/
/* Author : Gerard MARTINELLI.                          September 99 */
/*-------------------------------------------------------------------*/
admin          = "GEGE"            /* Administrator Virtual Machine  */

Address command
"EXECIO 0 CP (STRING SPOOL CONS" admin "START"/* Log the console     */
"CP TERM MORE 0 0"                           /* avoid to be stuck at  */
"CP TERM HOLD OFF"                           /* the console.          */


vse_ip         = "1.2.3.4"         /* VSE IP address                 */
vse_port       = "1414"            /* VSE MQseries TCP.IP port nb    */
qm             = "DCT.MQ.VSE"      /* Queue Manager name             */
Cli_chn        = "VM_VSE"          /* MQSeries Client Channel name   */
q_in           = "ANYQ"            /* MQSeries Input queue           */
q_out          = "ANYQ"            /* MQSeries output Queue          */
tracesw        = 1                 /* activate message trace if 1    */
linecnt        = 500               /* Force console closing  ever 500 */
                                   /* lines                          */
maxmsg         = 10                /* number of mesages to be written */

/*----------- Set MQ/series environment  to talk to VSE.---------------*/

"NUCXDROP RXMQV"                        /* Get rid of previous if any */
"NUCXLOAD RXMQV (SYSTEM"                /* make module resident for   */
                                        /* REXX MQSEries  interface   */
"SET STORECLR ENDCMD"                   /* Release storage when over  */
"GLOBAL LOADLIB SCEERUN AMQLLIB"        /*  LE and MQSeries libraries */

mqserver = cli_chn"/TCP/"vse_ip"("vse_port")"    /* build env.  string */
"GLOBALV SELECT CENV SETLP MQSERVER" mqserver    /* Set MQ environment */
"GLOBALV SELECT CENV SETLP MQ_USER_ID" userid()  /* Identify myself    */

/*  Move this line after the next one for activate the QMSeries Trace
"GLOBALV SELECT CENV SETLP MQTRACE MQCLIENT.TRACE.A"  /* trace MQ API */
*/

/*-------------------------------------------------------------------*/
/*                   INIT and connect to the Queue Manager            */
```

```
/*--------------------------------------------------------------------*/

Call msgrout "MQSeries INIT started"
rcx  = RXMQV("INIT")
Call msgrout "MQSeries INIT ended"
If word(rcx,1)<> 0 then
  Do
      "CP MSG" admin "MQ/Series Interface Initialization failure."
      Exit 8
  End
Call msgrout "MQSeries CONNECT started"
rcx   = RXMQV("CONN",qm)
Call msgrout "MQSeries CONNECT Ended "
x= word(rcx,1)
If x<> 0 & x<>2002  then
  Do
      "CP MSG" admin "Connection to Q-Manager failed."
      Exit 8
  End

Call Process                                 /* proceed with mesages */

If word(rcx,1)<> 0 then
   Say "Return code from  last function was:" rcx

/*--------------------------------------------------------------------*/
/*                   Close and terminate gently this procedure        */
/*--------------------------------------------------------------------*/
Call msgrout "MQSeries DISCON  started"
rcx= RXMQV("DISC")
Call msgrout "MQSeries DISCON  Ended"
If word(rcx,1)<> 0 then
  Say "Return code from Disconnect=" rcx

Call msgrout "MQSeries TERM started"
rcx= RXMQV("TERM")
Call msgrout "MQSeries TERM ended"
If word(rcx,1)<> 0 then
  Say "Return code from TERMinate=" rcx
"EXECIO 0 CP  (STRING CLOSE CONSOLE"
Exit 0

Process:          /*   Main  procedure  */

/*--------------------------------------------------------------------*/
/*                     Open the MQ/Series Output queue                 */
/*--------------------------------------------------------------------*/
oo= mqoo_output                     /*previously initialized by MQCONN */
Call msgrout "MQSeries OPEN output queue started"
rcx   = RXMQV("OPEN",q_out,oo,'hqn',ood.)
Call msgrout "MQSeries OPEN output queue ended"
If word(rcx,1)<> 0 then
      Return rcx


/*--------------------------------------------------------------------*/
/*                     Issue multiple MQPUTs                          */
/*--------------------------------------------------------------------*/
imd.PER         =  MQPER_PERSISTENT
imd.FORM        =  MQFMT_STRING
imd.MSGTYPE     =  MQMT_DATAGRAM
imd.REPLYTOQ    =  ' '
imd.REPLYTOQMGR =  ' '
imd.MSGID       =  MQMI_NONE
imd.CORRELID    =  MQCI_NONE
ipmo.opt        =  MQPMO_NO_SYNCPOINT

Call msgrout "MQSeries PUTs started"
Do i=1 to maxmsg              /* write 10 messages   */
   msg.1  = "This is message number" i
   msg.0  = length(msg.1)
   rcx   = RXMQV("PUT",hqn,'msg.','imd.','omd.','ipmo.','opmo.')
End
Call msgrout "MQSeries PUTs ended"
If word(rcx,1)<> 0 then
      Return rcx
Call msgrout "MQSeries COMMIT started"
/*-----------------------------------------------------------------------*/
/*                    Commit all messages and close the queue.          */
/*-----------------------------------------------------------------------*/
rcx   = RXMQV("CMIT")
Call msgrout "MQSeries COMMIT ended"
Call msgrout "MQSeries CLOSE Output queue started"
rcx = RXMQV("CLOSE",hqn,MQCO_NONE)
Call msgrout "MQSeries CLOSE Output queue ended"
If word(rcx,1)<> 0 then
      Return rcx

/*-----------------------------------------------------------------------*/
/*                Ask for checking monitor Qdepth on VSE side           */
/*-----------------------------------------------------------------------*/
Say "Check QDepth with the VSE Monitor dialog. Then press Enter"
pull xxxx

/*-----------------------------------------------------------------------*/
/*                     Now we retrieve  messages                        */
/*-----------------------------------------------------------------------*/

oo= mqoo_input_as_q_def
Call msgrout "MQSeries OPEN Input queue started"
rcx   = RXMQV("OPEN",q_in,oo,'hqn',ood.)
Call msgrout "MQSeries OPEN Input queue ended"
If word(rcx,1)<> 0 then
      Return rcx

Call msgrout "MQSeries GETs started"
Do forever                                   /*Loop until the queue   */
                                             /* is empty.             */
   g.0 = 200                                 /*  Maximum message size */
   g.1 = ""
   igmo.opt = MQGMO_WAIT + MQGMO_CONVERT
   igmo.WAIT = 05000                       /* wait up to 5 seconds    */
   rcx   = RXMQV("GET",hqn,'g.','igmd.','ogmd.','igmo.','ogmo.')
   If word(rcx,1)<> 0 then
      leave
   say g.1               /* Display message */
End
x= word(rcx,1)
If x<> 0 & x<>2033  then                    /* 2033 means queue empty */
   Return rcx
```

```
Call msgrout "MQSeries GETs ended"

/*------------------------------------------------------------------*/
/*                   Commit all MQGETS and close the queue.          */
/*------------------------------------------------------------------*/
Call msgrout "MQSeries COMMIT started"
rcx   = RXMQV("CMIT")
Call msgrout "MQSeries COMMIT ended"
Call msgrout "MQSeries CLOSE Input queue started"
rcx = RXMQV("CLOSE",hqn,MQCO_NONE)
Call msgrout "MQSeries CLOSE Input queue ended"

/*-----------------------------------------------------------------*/
/* The following 2 lines update  monitor counters. It is a VSE bug    */
/* on the current MQSeries for VSE level.                           */
             rcx = RXMQV("OPEN",q_out,oo,'hqn',ood.)
             rcx = RXMQV("CLOSE",hqn,MQCO_NONE)
/*  get rid of them when a PTF fixes the problem.                   */
/*-----------------------------------------------------------------*/

Say "Check again QDepth with the VSE Monitor dialog. Then press Enter"
Pull xxxx
Return rcx

/*------------------------------------------------------------------*/
/*                   User Trace Routine                              */
/*------------------------------------------------------------------*/
Msgrout: Procedure expose tracesw linecnt
 Parse arg msg
 If tracesw then              /* Do if trace activated               */
   Do
    linecnt=linecnt+1         /* Add one line to count               */
    Say msg
    if linecnt>500 then        /* close console every 500 messages    */
      Do
        linecnt=0
        "EXECIO 0 CP  (STRING CLOSE CONSOLE"
      End
   End
Return
```

COBOL language programs

Rather than reproduce information available in the *MQSeries for VSE System Management Guide*, GC34-5364, we do not provide a sample COBOL MQI program here.

There are three sample COBOL MQI programs in Appendix E of the *MQSeries for VSE System Management Guide*, GC34-5364.

C language programs

The following sample program is a CICS/C example using CICS and MQI calls. You may notice that we have typecast some of the parameters to MQI calls. This is because some documentation is misleading and suggests input parameters are passed by value. This is not the case. All parameters are passed by reference.

```
/*********************************************************************
 *
 *  Program:     MQPCTST
 *  Description: This is a test CICS/C test program that demonstrates
 *               MQI calls from a C program.
 *
 ********************************************************************/

/* Header file includes */
#include <stdio.h>
#include <string.h>
#include <cics.h>
#define _far
#define _pascal
#define _loadds
#include <cmqc.h>

/* Definitions */
#define Q_NAME              "BRET.LQ1"
#define TEST_MESSAGE        "This is a test message."
#define MQ_RESERVED         (MQLONG)0

/* Function prototypes */
void Term(char *);
void Fail(char *, int, int);
void wcon(char *);

/*********************************************************************
 *
 *  Function:    main
 *  Description: The main routine connects to a default queue
 *               manager, opens a queue, puts one message, closes
 *               the queue and disconnects from the queue manager.
 *               Success or failure is logged to the console along
 *               with reason code and completion code details.
 *
 ********************************************************************/

main()
{
    MQLONG    rc, cc;
    MQHCONN   hcon;
    MQOD      objDesc;
    MQLONG    options;
    MQHOBJ    hobj;
    MQMD      msgDesc;
    MQPMO     pmo;
    MQLONG    msglen;
    char      qm[48];
    char      msg[60];


    wcon("Test program begins");
```

```
EXEC CICS ADDRESS
          EIB(dfheiptr);

/* Connect to Queue Manager */
memset(qm, 0x40, 48);
MQCONN(qm, &hcon, &rc, &cc);
if ( rc )
{
    Fail("MQCONN", rc, cc);
    Term("Test program failed");
}

/* Initialize Object Descriptor for MQOPEN */
objDesc.Version    = MQOD_VERSION_1;
objDesc.ObjectType = MQOT_Q;
memcpy(objDesc.StrucId, MQOD_STRUC_ID, 4);
strcpy(objDesc.ObjectName, Q_NAME);
memset(objDesc.ObjectQMgrName, 0, 48);
memset(objDesc.DynamicQName, 0, 48);
memset(objDesc.AlternateUserId, 0, 12);

/* Open option to PUT messages */
options = MQOO_OUTPUT;

/* Open the target queue */
MQOPEN((MQHCONN)&hcon,
       &objDesc,
       (MQLONG)&options,
       &hobj,
       &rc,
       &cc);
if ( rc )
{
    Fail("MQOPEN", rc, cc);
    MQDISC(&hcon, &rc, &cc);
    Term("Test program failed");
}

/* Initialize the Message Descriptor for MQPUT */
msgDesc.Version      = MQMD_VERSION_1;
msgDesc.Report       = MQRO_NONE;
msgDesc.MsgType      = MQMT_DATAGRAM;
msgDesc.Expiry       = -1;
msgDesc.Feedback     = MQFB_NONE;
msgDesc.Encoding     = MQENC_NATIVE;
msgDesc.CodedCharSetId = MQCCSI_Q_MGR;
msgDesc.Priority     = MQPRI_PRIORITY_AS_Q_DEF;
msgDesc.Persistence  = MQPER_PERSISTENT;
msgDesc.BackoutCount = MQ_RESERVED;
msgDesc.PutApplType  = MQ_RESERVED;
memcpy(msgDesc.StrucId, MQMD_STRUC_ID, 4);
memcpy(msgDesc.Format, MQFMT_NONE, 8);
memset(msgDesc.MsgId, 0, 24);
memcpy(msgDesc.CorrelId, MQCI_NONE, 24);
memset(msgDesc.ReplyToQ, 0, 48);
memset(msgDesc.ReplyToQMgr, 0, 48);
memset(msgDesc.UserIdentifier, 0, 12);
memset(msgDesc.AccountingToken, 0, 32);
memset(msgDesc.ApplIdentityData, 0, 32);
memset(msgDesc.PutApplName, 0, 28);
memset(msgDesc.PutDate, 0, 8);
memset(msgDesc.PutTime, 0, 8);
memset(msgDesc.ApplOriginData, 0, 4);

/* Initialize Put Message Options for MQPUT */
pmo.Version           = MQPMO_VERSION_1;
pmo.Options           = MQPMO_NONE;
pmo.Timeout           = MQ_RESERVED;
pmo.Context           = (MQHOBJ)(MQ_RESERVED);
pmo.KnownDestCount    = MQ_RESERVED;
pmo.UnknownDestCount  = MQ_RESERVED;
pmo.InvalidDestCount  = MQ_RESERVED;
memcpy(pmo.StrucId, MQPMO_STRUC_ID, 4);
memset(pmo.ResolvedQName, 0, 48);
memset(pmo.ResolvedQMgrName, 0, 48);

/* Initialize test message for MQPUT */
strcpy(msg, TEST_MESSAGE);
msglen = strlen(msg);

/* Put message to target queue */
MQPUT((MQHCONN)&hcon,
      (MQHOBJ)&hobj,
      &msgDesc,
      &pmo,
      (MQLONG)&msglen,
      msg,
      &rc,
      &cc);
if ( rc )
{
    Fail("MQPUT", rc, cc);
    options = MQCO_NONE;
    MQCLOSE((MQHCONN)&hcon,
            &hobj,
            (MQLONG)&options,
            &rc,
            &cc);
    MQDISC(&hcon, &rc, &cc);
    Term("Test program failed");
}

options = MQCO_NONE;
MQCLOSE((MQHCONN)&hcon,
         &hobj,
         (MQLONG)&options,
         &rc,
         &cc);
if ( rc )
{
    Fail("MQCLOSE", rc, cc);
    MQDISC(&hcon, &rc, &cc);
    Term("Test program failed");
}

/* Commit work */
EXEC CICS SYNCPOINT;

MQDISC(&hcon, &rc, &cc);
```

```
    if ( rc )
    {
        Fail("MQDISC", rc, cc);
        Term("Test program failed");
    }

    Term("Test program successful");
}

/*********************************************************************
 *
 *  Function:    Term
 *  Description: The Term function puts a message to the console,
 *               clears and frees the terminal and returns to CICS.
 *
 ********************************************************************/
void Term(char *msg)
{
    wcon(msg);

    EXEC CICS SEND CONTROL ERASE FREEKB;
    EXEC CICS RETURN;
}

/*********************************************************************
 *
 *  Function:    Fail
 *  Description: The Fail function formats a message based on passed
 *               parameters and puts the message to th console.
 *
 ********************************************************************/
void Fail(char *op, int rc, int cc)
{
    char emsg[60];

    sprintf(emsg, "Operation %s failed. [%d,%d]", op, rc, cc);
    wcon(emsg);
}

/*********************************************************************
 *
 *  Function:    wcon
 *  Description: The wcon function formats and writes a message to
 *               the console.
 *
 ********************************************************************/
void wcon(char *msg)
{
    int  mlen;
    char cmsg[60];

    sprintf(cmsg, "MQPCTST: %s", msg);
    mlen = strlen(cmsg);
    EXEC CICS WRITE OPERATOR
              TEXT(cmsg)
              TEXTLENGTH(mlen);
}
```

Assembler language programs

The following sample program is a CICS Assembler program that issues MQI calls. Because you cannot have a Language Environment Assembler main in CICS, this sample program must be called by a Language Environment main program written in COBOL, C, or PL/I. See 8.1, “CICS application programs” on page 148 for more details.

```
* ----------------------------------------------------------------------
*
*        MQSeries for VSE:   Test Program Only
*
*        This is a sample test program that shows MQI API calls
*        being made from a CICS Assembler program. Note that the
*        use of MQI structures is hard-coded, and for the use of
*        such structures to be meaningful, you should refer to the
*        following manual:
*
*        MQSeries Application Programming Reference (SC33-1673)
*
* ----------------------------------------------------------------------
MQASMEX  DFHEIENT CODEREG=(12),DATAREG=(11),EIBREG=(10)
MQASMEX  AMODE 31
MQASMEX  RMODE ANY
         SPACE
*
*        The following GETMAIN is for an arbitrary 4k which is
*        for the purposes of this test program enough for the MQI
*        structures, a 1k message buffer, a 1k of screen messages.
*
         EXEC  CICS GETMAIN                                            *
               SET(7)                                                  *
               FLENGTH(4096)                                           *
               INITIMG(NULLS)
         CLI   EIBRCODE,X'00'
         BE    GETMOK
         EXEC  CICS SEND FROM(TSKERR1) LENGTH(L'TSKERR1) ERASE
         EXEC  CICS RETURN
         SPACE
GETMOK   DS    0H                       Got storage ok
         USING TSKDATA,7                Using R7 for storage dsect
         ST    13,TSKSAVE+4             Save the callers save address
         LA    13,TSKSAVE               Set up our own savearea
         LA    9,TSKPARM                Set R9 to parm list area
         SPACE
         LA    6,TSKMSGA                Set up for screen message area
         ST    6,TSKMSGO
         LA    5,TSKMSG1                Tell user program started
         BAL   8,TSKMSG
         LA    5,TSKMSG2                Tell user getmain ok
         BAL   8,TSKMSG
         SPACE
         BAL   8,TSKINIT                Set up MQI structure
         LA    5,TSKMSG3                Tell user initialization ok
         BAL   8,TSKMSG
         SPACE
         BAL   8,TSKCONN                Attempt MQCONN
         CLC   TSKCCOD,=F'4'            Check comp code
         BNH   CONNOK                   Branch if ok
         LA    5,TSKERR2                Else tell user MQCONN failed
         BAL   8,TSKMSG
         B     TSKEND                  And finish up
         SPACE
CONNOK   DS    0H                      MQCONN ok
         LA    5,TSKMSG4               Tell user MQCONN ok
         BAL   8,TSKMSG
         SPACE
         BAL   8,TSKOPEN               Attempt MQOPEN
         CLC   TSKCCOD,=F'0'           Check comp code
         BZ    OPENOK                  Branch if ok
         LA    5,TSKERR3               Else tell user MQOPEN failed
         BAL   8,TSKMSG
         B     DISCONN                 And disconnect
         SPACE
OPENOK   DS    0H                      MQOPEN ok
         LA    5,TSKMSG5               Tell user MQOPEN ok
         BAL   8,TSKMSG
         SPACE
         LA    3,10                    Prepare to write 10 messages
PUTLOOP  DS    0H
         BAL   8,TSKPUT                Attempt to MQPUT
         CLC   TSKCCOD,=F'0'           Check comp code
         BZ    PUTOK                   Branch if ok
         LA    5,TSKERR4               Else tell user MQPUT failed
         BAL   8,TSKMSG                Then, rollback
         EXEC  CICS SYNCPOINT ROLLBACK
         B     CLOSEQ                  And close the queue
         SPACE
PUTOK    DS    0H                      MQPUT ok
         BCT   3,PUTLOOP               Loop until all 10 written
         SPACE
*                                      Commit the changes
         EXEC  CICS SYNCPOINT
         LA    5,TSKMSG6               And tell user all MQPUT ok
         BAL   8,TSKMSG
         SPACE
CLOSEQ   DS    0H                      Now close the Queue
         BAL   8,TSKCLOS               Attempt to MQCLOSE
         CLC   TSKCCOD,=F'0'           Check comp code
         BZ    CLOSEOK                 Branch if ok
         LA    5,TSKERR5               Else tell user MQCLOSE failed
         BAL   8,TSKMSG
         B     DISCONN                 And disconnect
         SPACE
CLOSEOK  DS    0H                      MQCLOSE ok
         LA    5,TSKMSG7               Tell user MQCLOSE ok
         BAL   8,TSKMSG
         SPACE
DISCONN  DS    0H                      Now disconnect from QM
         BAL   8,TSKDISC               Attempt to MQDISC
         CLC   TSKCCOD,=F'0'           Check comp code
         BZ    DISCOK                  Branch if ok
         LA    5,TSKERR6               Else, tell user MQDISC failed
         BAL   8,TSKMSG
         B     TSKEND                  And finish up
         SPACE
DISCOK   DS    0H                      MQDISC ok
         LA    5,TSKMSG8               Tell user MQDISC ok
         BAL   8,TSKMSG
         SPACE
TSKEND   DS    0H                      Finish up
```

```
         LA    5,TSKMSG9               Tell user finishing
         BAL   8,TSKMSG                Then, free storage and return
         EXEC  CICS FREEMAIN DATAPOINTER(7)
         EXEC  CICS RETURN
         DROP  7
         EJECT
* ------------------------------------------------------------------
* SUBROUTINES
* ------------------------------------------------------------------
* ------------------------------------------------------------------
* TSKMSG:      Puts runtime messages to the users screen.
* ------------------------------------------------------------------
TSKMSG   DS    0H
         USING TSKDATA,7
         L     6,TSKMSGO
         MVC   0(80,6),0(5)
         LA    6,80(,6)
         ST    6,TSKMSGO
         LA    5,TSKMSGA
         SR    6,5
         STH   6,TSKMSGL
         EXEC  CICS SEND FROM(TSKMSGA) LENGTH(TSKMSGL) ERASE
         BR    8
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKINIT:     Initializes MQI structures.
* ------------------------------------------------------------------
TSKINIT  DS    0H
         USING TSKDATA,7
         MVC   TSKMNGR(4),=C'mmmm'     mmmm = your queue manager
         MVI   TSKMNGR+4,X'00'         Null terminate
         MVC   TSKOSTR,=C'OD  '        OD structure id
         MVC   TSKOVER,=F'1'           OD Version
         MVC   TSKOTYP,=F'1'           OD Type
         MVC   TSKONAM(8),=C'qqqq.qqq' qqqq = your queue name
         MVI   TSKOMGR+8,X'00'         Null terminate
         MVC   TSKOMGR(4),=C'mmmm'     mmmm = your queue manager
         MVI   TSKOMGR+4,X'00'         Null terminate
         MVC   TSKOOPT,=F'16'          OD Open option
         MVC   TSKMSTR,=C'MD  '        MD strutcure id
         MVC   TSKMVER,=F'1'           MD Version
         MVC   TSKMREP,=F'0'           MD Report option none
         MVC   TSKMTYP,=F'8'           MD Type
         MVC   TSKMEXP,=F'-1'          MD Expiry
         MVC   TSKMFED,=F'0'           MD Feedback
         MVC   TSKMENC,=F'785'         MD Encoding
         MVC   TSKMSET,=F'0'           MD coded char set id
         MVI   TSKMFMT,X'00'           MD Format
         MVC   TSKMPRI,=F'0'           MD Priority
         MVC   TSKMPER,=F'2'           MD Persistence
         MVI   TSKMMID,X'00'           MD message id
         MVC   TSKMMID+1(L'TSKMMID-1),TSKMMID
         MVI   TSKMCOR,X'00'           MD Correlation id
         MVC   TSKMCOR+1(L'TSKMCOR-1),TSKMCOR
         MVC   TSKMCNT,=F'0'           MD backout count
         MVI   TSKMRPQ,X'00'           MD reply to queue
         MVI   TSKMRPM,X'00'           MD reply to qmgr
         MVC   TSKMOST,=C'PMO '        PMO structure id
         MVC   TSKMOVR,=F'1'           PMO version
         MVC   TSKMOOP,=F'2'           PMO options
         MVI   TSKMORQ,X'00'           PMO resolved qname
         MVC   TSKMORQ+1(L'TSKMORQ-1),TSKMORQ
         MVI   TSKMORM,X'00'           PMO resolved qmgr
         MVC   TSKMORM+1(L'TSKMORM-1),TSKMORM
         MVC   TSKBFLN,=F'1024'        Message length 1k
         MVC   TSKCOPT,=F'0'           Close option
         SPACE
*
*        The following loop sets up a message buffer with
*        "This is a ASM task test messages!" repeated 32 times
*        to form a 1k queue message.
*
         LA    1,=C'This is a ASM task test message!'
         LA    2,32
         LA    4,TSKBUFR
INITLOOP DS    0H
         MVC   0(32,4),0(1)
         LA    4,32(,4)
         BCT   2,INITLOOP
         BR    8
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKCONN:     Sets up and issues an MQCONN MQI call.
* ------------------------------------------------------------------
TSKCONN  DS    0H
         USING TSKDATA,7
         USING CONPARML,9
         LA    1,TSKMNGR
         ST    1,CONAQNME
         LA    1,TSKHCON
         ST    1,CONAHCON
         LA    1,TSKCCOD
         ST    1,CONACCOD
         LA    1,TSKRCOD
         ST    1,CONARCOD
         LR    1,9
         L     15,=V(MQCONN)
         BALR  14,15
         BR    8
         DROP  9
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKOPEN:     Sets up and issues an MQOPEN MQI call.
* ------------------------------------------------------------------
TSKOPEN  DS    0H
         USING TSKDATA,7
         USING OPEPARML,9
         LA    1,TSKHCON
         ST    1,OPEAHCON
         LA    1,TSKDESC
         ST    1,OPEAOBJD
         LA    1,TSKOOPT
         ST    1,OPEAOOPT
         LA    1,TSKHOBJ
         ST    1,OPEAHOBJ
         LA    1,TSKCCOD
         ST    1,OPEACCOD
         LA    1,TSKRCOD
         ST    1,OPEARCOD
         LR    1,9
         L     15,=V(MQOPEN)
         BALR  14,15
         BR    8
         DROP  9
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKPUT:      Sets up and issues an MQPUT MQI call.
* ------------------------------------------------------------------
TSKPUT   DS    0H
         USING TSKDATA,7
         USING PUTPARML,9
         LA    1,TSKHCON
         ST    1,PUTAHCON
         LA    1,TSKHOBJ
         ST    1,PUTAHOBJ
         LA    1,TSKMDSC
         ST    1,PUTAMDSC
         LA    1,TSKMOPT
         ST    1,PUTAMOPT
         LA    1,TSKBFLN
         ST    1,PUTABFLN
         LA    1,TSKBUFR
         ST    1,PUTABUFF
         LA    1,TSKCCOD
         ST    1,PUTACCOD
         LA    1,TSKRCOD
         ST    1,PUTARCOD
         LR    1,9
         L     15,=V(MQPUT)
         BALR  14,15
         BR    8
         DROP  9
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKCLOS:     Sets up and issues an MQCLOSE MQI call.
* ------------------------------------------------------------------
TSKCLOS  DS    0H
         USING TSKDATA,7
         USING CLOPARML,9
         LA    1,TSKHCON
         ST    1,CLOAHCON
         LA    1,TSKHOBJ
         ST    1,CLOAHOBJ
         LA    1,TSKCOPT
         ST    1,CLOAOOPT
         LA    1,TSKCCOD
         ST    1,CLOACCOD
         LA    1,TSKRCOD
         ST    1,CLOARCOD
         LR    1,9
         L     15,=V(MQCLOSE)
         BALR  14,15
         BR    8
         DROP  9
         DROP  7
         SPACE
* ------------------------------------------------------------------
* TSKDISC:      Sets up and issues an MQDISC MQI call.
* ------------------------------------------------------------------
TSKDISC  DS    0H
         USING TSKDATA,7
         USING DISPARML,9
         LA    1,TSKHCON
         ST    1,DISAHCON
         LA    1,TSKCCOD
         ST    1,DISACCOD
         LA    1,TSKRCOD
         ST    1,DISARCOD
         LR    1,9
         L     15,=V(MQDISC)
         BALR  14,15
         BR    8
         DROP  12
         DROP  11
         DROP  10
         DROP  9
         DROP  7
         SPACE
* ------------------------------------------------------------------
* CONSTANTS:
* ------------------------------------------------------------------
NULLS    DC    X'00'
         SPACE
TSKMSG1  DC    CL80'MQSeries ASM Test Program Begins'
TSKMSG2  DC    CL80'MQSeries ASM Test Program GETMAIN successful'
TSKMSG3  DC    CL80'MQSeries ASM Test Program Initial successful'
TSKMSG4  DC    CL80'MQSeries ASM Test Program Connect successful'
TSKMSG5  DC    CL80'MQSeries ASM Test Program Openq   successful'
TSKMSG6  DC    CL80'MQSeries ASM Test Program Putq    successful'
TSKMSG7  DC    CL80'MQSeries ASM Test Program Closeq  successful'
TSKMSG8  DC    CL80'MQSeries ASM Test Program Disconn successful'
TSKMSG9  DC    CL80'MQSeries ASM Test Program Ends'
         SPACE
TSKERR1  DC    CL80'MQSeries ASM Test Program GETMAIN failed'
TSKERR2  DC    CL80'MQSeries ASM Test Program Connect failed'
TSKERR3  DC    CL80'MQSeries ASM Test Program Openq   failed'
TSKERR4  DC    CL80'MQSeries ASM Test Program Putq    failed'
TSKERR5  DC    CL80'MQSeries ASM Test Program Closeq  failed'
TSKERR6  DC    CL80'MQSeries ASM Test Program Disconn failed'
         SPACE
         LTORG
         EJECT
* ------------------------------------------------------------------
* MAIN STORAGE DSECT
* ------------------------------------------------------------------
TSKDATA  DSECT
TSKSAVE  DS    18F
         SPACE
TSKMNGR  DS    XL48
TSKQNME  DS    XL48
TSKHCON  DS    F
TSKHOBJ  DS    F
TSKCCOD  DS    F
TSKRCOD  DS    F
TSKOOPT  DS    F
```

```
TSKCOPT  DS    F
TSKOPTN  DS    F
TSKBFLN  DS    F
         SPACE
TSKDESC  DS    0F
TSKOSTR  DS    F
TSKOVER  DS    F
TSKOTYP  DS    F
TSKONAM  DS    XL48
TSKOMGR  DS    XL48
TSKODYN  DS    XL48
TSKOALT  DS    XL12
         SPACE
TSKMDSC  DS    0F
TSKMSTR  DS    F
TSKMVER  DS    F
TSKMREP  DS    F
TSKMTYP  DS    F
TSKMEXP  DS    F
TSKMFED  DS    F
TSKMENC  DS    F
TSKMSET  DS    F
TSKMFMT  DS    XL8
TSKMPRI  DS    F
TSKMPER  DS    F
TSKMMID  DS    XL24
TSKMCOR  DS    XL24
TSKMCNT  DS    F
TSKMRPQ  DS    XL48
TSKMRPM  DS    XL48
TSKMUSR  DS    XL12
TSKMTOK  DS    XL32
TSKMAPI  DS    XL32
TSKMATY  DS    F
TSKMANM  DS    XL28
TSKMPDT  DS    XL8
TSKMPTM  DS    XL8
TSKMAPO  DS    F
         SPACE
TSKMOPT  DS    0F
TSKMOST  DS    F
TSKMOVR  DS    F
TSKMOOP  DS    F
TSKMOTO  DS    F
TSKMOCX  DS    F
TSKMOKD  DS    F
TSKMOUD  DS    F
TSKMOID  DS    F
TSKMORQ  DS    XL48
TSKMORM  DS    XL48
         SPACE
TSKPARM  DS    10F
TSKBUFR  DS    CL(1024)
TSKMSGO  DS    F
TSKMSGL  DS    H
TSKMSGA  DS    CL(1024)
TSKDATAL EQU   *-TSKDATA
         EJECT
* ------------------------------------------------------------------
*              MQCONN function  Dsect Definitions
* -------------------------------------------------------------------
*
CONPARML DSECT
CONAQNME DS    A                       Queue Name area address
CONAHCON DS    A                       Connection Handle area address
CONACCOD DS    A                       Completion code area address
CONARCOD DS    A                       Reason code area address
         SPACE
* -------------------------------------------------------------------
*              MQDISC function  Dsect Definitions
* -------------------------------------------------------------------
*
DISPARML DSECT
DISAHCON DS    A                       Connection Handle area address
DISACCOD DS    A                       Completion code area address
DISARCOD DS    A                       Reason code area address
         SPACE
*
* -------------------------------------------------------------------
*              MQOPEN function  Dsect Definitions
* -------------------------------------------------------------------
*
OPEPARML DSECT
OPEAHCON DS    A                       Connection Handle area address
OPEAOBJD DS    A                       Object Descript area address
OPEAOOPT DS    A                       Open Options area Address
OPEAHOBJ DS    A                       Object Handle area Address
OPEACCOD DS    A                       Completion code area address
OPEARCOD DS    A                       Reason code area address
         SPACE
* -------------------------------------------------------------------
*              MQCLOSE function  Dsect Definitions
* -------------------------------------------------------------------
*
CLOPARML DSECT
CLOAHCON DS    A                       Connection Handle area address
CLOAHOBJ DS    A                       Object Handle area Address
CLOAOOPT DS    A                       Open Options area Address
CLOACCOD DS    A                       Completion code area address
CLOARCOD DS    A                       Reason code area address
         SPACE
* -------------------------------------------------------------------
*              MQPUT function  Dsect Definitions
* -------------------------------------------------------------------
*
PUTPARML DSECT
PUTAHCON DS    A                       Connection Handle area address
PUTAHOBJ DS    A                       Object Handle area Address
PUTAMDSC DS    A                       Message Descriptor area Address
PUTAMOPT DS    A                       Message Options area Address
PUTABFLN DS    A                       Buffer Length area address
PUTABUFF DS    A                       Data buffer address
PUTACCOD DS    A                       Completion code area address
PUTARCOD DS    A                       Reason code area address
         SPACE
         END
```

PCF sample

The following sample program is a CICS/COBOL example using CICS and MQI calls. The sample builds a PCF request message and puts it on the system command queue for processing by the MQSeries PCF Command Server. This sample is explained in detail in 5.1.2, “PCF programming” on page 85.

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    MQTSTPCF.
       AUTHOR.        IBM.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.

       DATA DIVISION.
      *-------------------------------------------------------*
       WORKING-STORAGE SECTION.
      *-------------------------------------------------------*

       01 WS-CMQV.
          COPY CMQV.

       01 WS-CMQCFV.
          COPY CMQCFV.

       01 WS-MQOD.
          COPY CMQODV.

       01 WS-MQMD.
          COPY CMQMDV.

       01 WS-MQPMO.
          COPY CMQPMOV.

       01 WS-HCONN.
          05 WS-HCONN-VALUE             USAGE POINTER.

       01 WS-CCODE.
          05 WS-CCODE-VALUE             PIC S9(8) COMP.

       01 WS-RCODE.
          05 WS-RCODE-VALUE             PIC S9(8) COMP.

       01 WS-QM-NAME                    PIC  X(48).

       01 WS-PCF-LEN                    PIC S9(8) COMP.

       01 WS-PCF-MSG.
          05 WS-HEADER.
             COPY CMQCFHV.
          05 WS-QNAME-PARM.
             COPY CMQCFSTV.
             15 WS-TARGET-Q             PIC  X(48).
          05 WS-QTYPE-PARM.
             COPY CMQCFINV.
          05 WS-QDEPTH-PARM.
             COPY CMQCFINV.

      *-------------------------------------------------------*
       LINKAGE SECTION.
```

```
*------------------------------------------------------*

*------------------------------------------------------*
 PROCEDURE DIVISION.
*------------------------------------------------------*

 0000-MAIN.

*    Connect to the queue manager
*
     SET WS-HCONN-VALUE TO NULL.
     MOVE SPACES TO WS-QM-NAME.

     CALL 'MQCONN' USING WS-QM-NAME
                         WS-HCONN
                         WS-CCODE
                         WS-RCODE.

     IF WS-CCODE-VALUE NOT = MQCC-OK
         PERFORM 0000-TERM
     END-IF.

*    Build the PCF command message
*
     MOVE MQCMD-CHANGE-Q   TO MQCFH-COMMAND.
     MOVE 3                TO MQCFH-PARAMETERCOUNT.

     ADD  MQ-Q-NAME-LENGTH TO MQCFST-STRUCLENGTH.
     MOVE MQ-Q-NAME-LENGTH TO MQCFST-STRINGLENGTH.
     MOVE MQCA-Q-NAME      TO MQCFST-PARAMETER.
     MOVE 'ANYQ'           TO WS-TARGET-Q.

     MOVE MQIA-Q-TYPE      TO MQCFIN-PARAMETER
                           OF WS-QTYPE-PARM.
     MOVE MQQT-LOCAL       TO MQCFIN-VALUE
                           OF WS-QTYPE-PARM.

     MOVE MQIA-MAX-Q-DEPTH TO MQCFIN-PARAMETER
                           OF WS-QDEPTH-PARM.
     MOVE 5000             TO MQCFIN-VALUE
                           OF WS-QDEPTH-PARM.

     MOVE LENGTH OF WS-PCF-MSG TO WS-PCF-LEN.

*    Put the PCF message on the system command queue
*
     MOVE 'SYSTEM.ADMIN.COMMAND.QUEUE'
       TO MQOD-OBJECTNAME.
     MOVE SPACES
       TO MQOD-OBJECTQMGRNAME.
     MOVE MQMT-REQUEST                 TO MQMD-MSGTYPE.
     MOVE MQFMT-ADMIN                  TO MQMD-FORMAT.
     MOVE 'PCF.REPLY'                  TO MQMD-REPLYTOQ.
     MOVE MQOO-OUTPUT                  TO MQPMO-OPTIONS.

     CALL 'MQPUT1' USING WS-HCONN
                         WS-MQOD
                         WS-MQMD
                         WS-MQPMO
                         WS-PCF-LEN
                         WS-PCF-MSG
                         WS-CCODE
                         WS-RCODE.

*    Syncpoint or rollback
*
     IF WS-CCODE-VALUE = MQCC-OK
         EXEC CICS SYNCPOINT END-EXEC
         CALL 'MQCMIT' USING WS-HCONN
                             WS-CCODE
                             WS-RCODE
         END-CALL
     ELSE
         EXEC CICS SYNCPOINT ROLLBACK END-EXEC
         CALL 'MQBACK' USING WS-HCONN
                             WS-CCODE
                             WS-RCODE
         END-CALL
     END-IF.


 0000-TERM.

*    Disconnect from the queue manager
*
     IF WS-HCONN-VALUE NOT = NULL
         CALL 'MQDISC' USING WS-HCONN
                             WS-CCODE
                             WS-RCODE
         END-CALL
     END-IF.

*    And terminate
*
     EXEC CICS SEND CONTROL
               ERASE
               FREEKB
     END-EXEC.

     EXEC CICS RETURN END-EXEC.
```

PCF Escape sample

The following sample program is a CICS/COBOL example using CICS and MQI calls. The sample builds a PCF Escape message and puts it on the system command queue for processing by the MQSeries PCF Command Server. This sample is explained in detail in 5.2, “MQSeries Commands (MQSC)” on page 88.

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    MQTSTESC.
       AUTHOR.        IBM.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.

       DATA DIVISION.
      *-------------------------------------------------------*
       WORKING-STORAGE SECTION.
      *-------------------------------------------------------*

       01 WS-CMQV.
          COPY CMQV.

       01 WS-CMQCFV.
          COPY CMQCFV.

       01 WS-MQOD.
          COPY CMQODV.

       01 WS-MQMD.
          COPY CMQMDV.

       01 WS-MQPMO.
          COPY CMQPMOV.

       01 WS-HCONN.
          05 WS-HCONN-VALUE             USAGE POINTER.

       01 WS-CCODE.
          05 WS-CCODE-VALUE             PIC S9(8) COMP.

       01 WS-RCODE.
          05 WS-RCODE-VALUE             PIC S9(8) COMP.

       01 WS-QM-NAME                    PIC X(48).

       01 WS-PCF-LEN                    PIC S9(8) COMP.

       01 WS-PCF-MSG.
          05 WS-HEADER.
             COPY CMQCFHV.
          05 WS-ESC-TYPE.
             COPY CMQCFINV.
          05 WS-ESC-TEXT.
             COPY CMQCFSTV.
             15 WS-MQ-CMD               PIC X(100).

      *-------------------------------------------------------*
       LINKAGE SECTION.
      *-------------------------------------------------------*
```

```
*-------------------------------------------------------*
 PROCEDURE DIVISION.
*-------------------------------------------------------*

 0000-MAIN.

*    Connect to the queue manager
*
     SET WS-HCONN-VALUE TO NULL.
     MOVE SPACES TO WS-QM-NAME.

     CALL 'MQCONN' USING WS-QM-NAME
                         WS-HCONN
                         WS-CCODE
                         WS-RCODE.

     IF WS-CCODE-VALUE NOT = MQCC-OK
         PERFORM 0000-TERM
     END-IF.

*    Build the PCF Escape message
*
     MOVE MQCMD-ESCAPE         TO MQCFH-COMMAND.
     MOVE 2                    TO MQCFH-PARAMETERCOUNT.

     MOVE MQIACF-ESCAPE-TYPE  TO MQCFIN-PARAMETER.
     MOVE MQET-MQSC           TO MQCFIN-VALUE.

     MOVE LENGTH OF WS-MQ-CMD TO MQCFST-STRINGLENGTH.
     ADD  MQCFST-STRINGLENGTH TO MQCFST-STRUCLENGTH.
     MOVE MQCACF-ESCAPE-TEXT  TO MQCFST-PARAMETER.
     MOVE 'ALTER QLOCAL(ANYQ) MAXDEPTH(5000)'
                              TO WS-MQ-CMD.

     MOVE LENGTH OF WS-PCF-MSG TO WS-PCF-LEN.

*    Put the PCF Escape on the system command queue
*
     MOVE 'SYSTEM.ADMIN.COMMAND.QUEUE'
       TO MQOD-OBJECTNAME.
     MOVE SPACES
       TO MQOD-OBJECTQMGRNAME.
     MOVE MQMT-REQUEST                 TO MQMD-MSGTYPE.
     MOVE MQFMT-ADMIN                  TO MQMD-FORMAT.
     MOVE 'PCF.REPLY'                  TO MQMD-REPLYTOQ.
     MOVE MQOO-OUTPUT                  TO MQPMO-OPTIONS.

     CALL 'MQPUT1' USING WS-HCONN
                         WS-MQOD
                         WS-MQMD
                         WS-MQPMO
                         WS-PCF-LEN
                         WS-PCF-MSG
                         WS-CCODE
                         WS-RCODE.

*    Syncpoint or rollback
*
     IF WS-CCODE-VALUE = MQCC-OK
         EXEC CICS SYNCPOINT END-EXEC
```

```
               CALL 'MQCMIT' USING WS-HCONN
                                   WS-CCODE
                                   WS-RCODE
               END-CALL
           ELSE
               EXEC CICS SYNCPOINT ROLLBACK END-EXEC
               CALL 'MQBACK' USING WS-HCONN
                                   WS-CCODE
                                   WS-RCODE
               END-CALL
           END-IF.


       0000-TERM.

      *    Disconnect from the queue manager
      *
           IF WS-HCONN-VALUE NOT = NULL
               CALL 'MQDISC' USING WS-HCONN
                                   WS-CCODE
                                   WS-RCODE
               END-CALL
           END-IF.

      *    And terminate
      *
           EXEC CICS SEND CONTROL
                     ERASE
                     FREEKB
           END-EXEC.

           EXEC CICS RETURN END-EXEC.
```

Instrumentation event sample

The following sample program is a CICS/C example using CICS and MQI calls. The sample reads Instrumentation Event messages from an event queue, and writes a message describing the event to the VSE console. This sample is explained in detail in "Processing logic for event messages" on page 94.

```
/* Includes */
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <cmqc.h>
  #include <cmqcfc.h>

 /* Defines */
  #define FALSE      0
  #define TRUE       1
  #define SYS_PERF_Q "SYSTEM.ADMIN.PERFM.EVENT"

 /* Prototypes */
  void ProcessEvents(MQHCONN, MQHOBJ);


 /************************************************************
  *   Main program connects to the queue manager, opens
  *   the performance event queue, and calls function
  *   ProcessEvents() until MQSeries is shutdown. It then
  *   closes the queue disconnects from the queue manager.
  ***********************************************************/
  int main(int argc, char **argv)
  {
     /*
      * Local variables.
      */
      MQHCONN  Hcon;
      MQHOBJ   Hobj;
      MQLONG   O_options;
      MQLONG   C_options;
      MQLONG   CompCode;
      MQLONG   Reason;
      MQCHAR48 QMName;
      MQOD     od = { MQOD_DEFAULT };


     /*
      * Connect to the queue manager.
      */
      memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
      MQCONN(QMName,
             &Hcon,
             &CompCode,
             &Reason);

      if ( CompCode == MQCC_OK )
      {
         /*
          * Open the performance event queue.
          */
          memset(od.ObjectQMgrName, ' ',
                 MQ_Q_MGR_NAME_LENGTH);
```

```
        strncpy(od.ObjectName, SYS_PERF_Q,
               MQ_Q_NAME_LENGTH);
        O_options = MQOO_INPUT_SHARED;
        MQOPEN(Hcon,
               &od,
               O_options,
               &Hobj,
               &CompCode,
               &Reason);

        if ( CompCode == MQCC_OK )
        {
           /*
            * Process event messages until shutdown.
            */
            ProcessEvents(Hcon, Hobj);

           /*
            * Close the performance event queue.
            */
            C_options = MQCO_NONE;
            MQCLOSE(Hcon,
                    &Hobj,
                    C_options,
                    &CompCode,
                    &Reason);
        }

       /*
        * Disconnect from the queue manager.
        */
        MQDISC(&Hcon, &CompCode, &Reason);
    }

   /*
    * Terminate the transaction.
    */
    EXEC CICS SEND CONTROL ERASE FREEKB;

    EXEC CICS RETURN;
}


/************************************************************
 * This function gets messages, as they arrive, from the
 * performance event queue. It then writes an appropriate
 * message the VSE console. The loop ends when
 * the MQGET fails (i.e. when MQSeries is shutdown).
 ***********************************************************/
 void ProcessEvents(MQHCONN hcon, MQHOBJ hobj)
 {
   /*
    * Local variables.
    */
    MQLONG     ccode;
    MQLONG     rcode;
    MQLONG     mlen;
    MQLONG     dlen;
    MQLONG     processing;
    PMQVOID    emsg;
    char       alert[20];
    char       conmsg[80];
    long       conlen;
    MQMD       md  = { MQMD_DEFAULT };
    MQGMO      gmo = { MQGMO_DEFAULT };

   /*
    * Data structure for performance event messages.
    */
    struct tagPerfEvent
    {
      MQCFH    Header;
      struct
      {
        _Packed MQCFST Qmgr;
        MQCHAR QMgrName[MQ_Q_MGR_NAME_LENGTH-1];
      } QmgrParm;
      struct
      {
        _Packed MQCFST Q;
        MQCHAR QName[MQ_Q_NAME_LENGTH-1];
      } QParm;
      MQCFIN   Reset;
      MQCFIN   Depth;
      MQCFIN   Puts;
      MQCFIN   Gets;
    } Evt;


   /*
    * Initialize variables.
    */
    emsg = (PMQVOID)&Evt;
    mlen = sizeof(struct tagPerfEvent);

   /*
    * Get performance messages in a loop. The loop
    * will terminate when the MQGET fails, i.e. when
    * MQSeries is shutdown.
    */
    processing = TRUE;
    while ( processing )
    {
       /*
        * Prepare to get a message.
        */
        gmo.WaitInterval = MQWI_UNLIMITED;
        gmo.Options = MQGMO_WAIT;
        memcpy(gmo.StrucId, "GMO ", 4);
        memset(md.MsgId, 0, MQ_MSG_ID_LENGTH);
        memset(md.CorrelId, 0, MQ_CORREL_ID_LENGTH);

       /*
        * Get a message, wait for one to arrive.
        */
        MQGET(hcon, hobj, &md, &gmo, mlen,
              emsg, &dlen, &ccode, &rcode);

       /*
        * If we got a message, check its reason and prepare
```

```
        * an appropriate text for our console message.
        */
        if ( ccode == MQCC_OK )
        {
            switch ( Evt.Header.Reason )
            {
                case MQRC_Q_FULL:
                    strcpy(alert, "Queue full");
                    break;
                case MQRC_Q_DEPTH_HIGH:
                    strcpy(alert, "Queue depth high");
                    break;
                case MQRC_Q_DEPTH_LOW:
                    strcpy(alert, "Queue depth low");
                    break;
                default:
                    strcpy(alert, "Unexpected event");
                    break;
            }
           /*
            * Build the console message.
            */
            Evt.QParm.Q.String[Evt.QParm.Q.StringLength-1]
                = 0;
            sprintf(conmsg,
                    "MQAlert - %s - for %s",
                    alert,
                    Evt.QParm.Q.String);
            conlen = strlen(conmsg);
           /*
            * Write the message to the console.
            */
            EXEC CICS WRITE OPERATOR
                      TEXT(conmsg)
                      TEXTLENGTH(conlen);
           /*
            * Commit after each message processed.
            */
            MQCMIT(hcon, &ccode, &rcode);
        }
        else
        {
           /*
            * Terminate the loop if the MQGET failed.
            */
            processing = FALSE;
        }
    }

   /*
    * Return to caller (main).
    */
    return;
}
```

DPL bridge sample

The following sample program is a C example that shows how the MQSeries–CICS bridge can be used to run a CICS DPL program. This sample is explained in detail in “DPL bridge programming” on page 169.

```
/* Includes */
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <cmqc.h>
  #include "dfhbrmqh.h"

 /* Defines */
  #define BRIDGE_Q   "SYSTEM.CICS.BRIDGE.QUEUE"
  #define REPLY_Q    "REPLY.Q"

 /* Prototypes */
  void RunDPLProgram(MQHCONN, MQHOBJ, MQHOBJ);


 /************************************************************
  *   Main program connects to the queue manager, opens the
  *   bridge request queue and a reply queue. It then calls
  *   RunDPLProgram() to put a request message and get a
  *   reply.
  ***********************************************************/
  int main(int argc, char **argv)
  {
     /*
      * Local variables.
      */
      MQHCONN  Hcon;
      MQHOBJ   Hobj_B;
      MQHOBJ   Hobj_R;
      MQLONG   O_options;
      MQLONG   C_options;
      MQLONG   CompCode;
      MQLONG   Reason;
      MQCHAR48 QMName;
      MQOD     od = { MQOD_DEFAULT };


      printf("MQTSTBR1: Program starts\n");

     /*
      * Connect to the queue manager.
      */
      memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
      MQCONN(QMName,
             &Hcon,
             &CompCode,
             &Reason);

      if ( CompCode == MQCC_OK )
      {
         /*
          * Open the bridge request queue.
          */
          memset(od.ObjectQMgrName, ' ',
                 MQ_Q_MGR_NAME_LENGTH);
          memcpy(od.ObjectName, BRIDGE_Q, strlen(BRIDGE_Q));
          O_options = MQOO_OUTPUT;
          MQOPEN(Hcon,
                 &od,
                 O_options,
                 &Hobj_B,
                 &CompCode,
                 &Reason);

          if ( CompCode == MQCC_OK )
          {
             /*
              * Open the reply queue.
              */
              memset(od.ObjectQMgrName, ' ',
                     MQ_Q_MGR_NAME_LENGTH);
              memset(od.ObjectName, ' ', MQ_Q_NAME_LENGTH);
              memcpy(od.ObjectName, REPLY_Q, strlen(REPLY_Q));
              O_options = MQOO_INPUT_SHARED;
              MQOPEN(Hcon,
                     &od,
                     O_options,
                     &Hobj_R,
                     &CompCode,
                     &Reason);

              if ( CompCode == MQCC_OK )
              {
                 /*
                  * Run a CICS DPL program.
                  */
                  RunDPLProgram(Hcon, Hobj_B, Hobj_R);

                 /*
                  * Close the reply queue.
                  */
                  C_options = MQCO_NONE;
                  MQCLOSE(Hcon,
                          &Hobj_R,
                          C_options,
                          &CompCode,
                          &Reason);
              }

             /*
              * Close the bridge request queue.
              */
              C_options = MQCO_NONE;
              MQCLOSE(Hcon,
                      &Hobj_B,
                      C_options,
                      &CompCode,
                      &Reason);
          }

         /*
          * Disconnect from the queue manager.
          */
          MQDISC(&Hcon, &CompCode, &Reason);
```

```
    }

    printf("MQTSTBR1: Program ends\n");

   /*
    * End program.
    */
    return(0);
}


/*********************************************************
 * This function puts a request message on the bridge
 * request queue, and waits for a reply from the bridge.
 * The request is to run DPL program MQBRTESO.
 *********************************************************/
 void RunDPLProgram(MQHCONN hcon,
                    MQHOBJ  hobj_b,
                    MQHOBJ  hobj_r)
 {
   /*
    * Local variables.
    */
    MQLONG     cc;
    MQLONG     rc;
    MQLONG     blen;
    MQLONG     mlen;
    MQLONG     dlen;
    PMQVOID    reqmsg;
    PMQVOID    repmsg;
    char       msgbuf[100];
    MQMD       md  = { MQMD_DEFAULT };
    MQGMO      gmo = { MQGMO_DEFAULT };
    MQPMO      pmo = { MQPMO_DEFAULT };


   /*
    * Prepare to put request.
    */
    reqmsg = (PMQVOID)&msgbuf;
    strcpy(reqmsg, "MQBRTESODATA IN ");
    blen = strlen(reqmsg);
    memset(md.ReplyToQ, 0, 48);
    memcpy(md.ReplyToQ, REPLY_Q, strlen(REPLY_Q));
    memcpy(md.CorrelId, MQCI_NEW_SESSION,
           MQ_CORREL_ID_LENGTH);
    memcpy(md.Format,   MQFMT_STRING, MQ_FORMAT_LENGTH);

   /*
    * Put the request on the bridge request queue.
    */
    MQPUT(hcon, hobj_b, &md, &pmo, blen, reqmsg, &cc, &rc);

    if ( cc != MQCC_OK )
        return;

   /*
    * Commit the request.
    */
    MQCMIT(hcon, &cc, &rc);
   /*
    * Prepare to get reply.
    */
    repmsg = (PMQVOID)&msgbuf;
    mlen = blen;
    gmo.WaitInterval = 10000;
    gmo.Options = MQGMO_WAIT | MQGMO_CONVERT;
    memcpy(md.CorrelId, md.MsgId, MQ_CORREL_ID_LENGTH);

   /*
    * Get the reply from the ReplyTo queue.
    */
    MQGET(hcon, hobj_r, &md, &gmo, mlen, repmsg, &dlen,
          &cc, &rc);

   /*
    * If we got a reply, print out the commarea.
    */
    if ( cc == MQCC_OK )
    {
        printf("MQTSTBR1: Commarea on return [%s]\n",
               (char *)repmsg+8);
    }

   /*
    * Return to caller (main).
    */
    return;
 }
```

3270 bridge sample

The following sample program is a C example that shows how the MQSeries-CICS bridge can be used to run a CICS 3270 transaction. This sample is explained in detail in “3270 bridge programming” on page 171.

```
/* Includes */
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <cmqc.h>
  #include "dfhbrmqh.h"

 /* Defines */
  #define BRIDGE_Q    "SYSTEM.CICS.BRIDGE.QUEUE"
  #define REPLY_Q     "REPLY.Q"
  #define ENTER       "'   "
  #define PF3         "3   "
  #define YES         "Y   "
  #define NO          "N   "

 /* Prototypes */
  void Run3270Transaction(MQHCONN, MQHOBJ, MQHOBJ);


 /************************************************************
  *   Main program connects to the queue manager, opens the
  *   bridge request queue and a reply queue. It then calls
  *   Run3270Transaction() to put a request message and get a
  *   reply.
  ***********************************************************/
  int main(int argc, char **argv)
  {
     /*
      * Local variables.
      */
      MQHCONN  Hcon;
      MQHOBJ   Hobj_B;
      MQHOBJ   Hobj_R;
      MQLONG   O_options;
      MQLONG   C_options;
      MQLONG   CompCode;
      MQLONG   Reason;
      MQCHAR48 QMName;
      MQOD     od = { MQOD_DEFAULT };


      printf("MQTSTBR2: Program starts\n");

     /*
      * Connect to the queue manager.
      */
      memset(QMName, ' ', MQ_Q_MGR_NAME_LENGTH);
      MQCONN(QMName,
             &Hcon,
             &CompCode,
             &Reason);

      if ( CompCode == MQCC_OK )
      {
```

```
/*
 * Open the bridge request queue.
 */
 memset(od.ObjectQMgrName, ' ',
        MQ_Q_MGR_NAME_LENGTH);
 memcpy(od.ObjectName, BRIDGE_Q, strlen(BRIDGE_Q));
 O_options = MQOO_OUTPUT;
 MQOPEN(Hcon,
        &od,
        O_options,
        &Hobj_B,
        &CompCode,
        &Reason);

 if ( CompCode == MQCC_OK )
 {
    /*
     * Open the reply queue.
     */
     memset(od.ObjectQMgrName, ' ',
            MQ_Q_MGR_NAME_LENGTH);
     memset(od.ObjectName, ' ', MQ_Q_NAME_LENGTH);
     memcpy(od.ObjectName, REPLY_Q, strlen(REPLY_Q));
     O_options = MQOO_INPUT_SHARED;
     MQOPEN(Hcon,
            &od,
            O_options,
            &Hobj_R,
            &CompCode,
            &Reason);

     if ( CompCode == MQCC_OK )
     {
        /*
         * Run a CICS DPL program.
         */
         Run3270Transaction(Hcon, Hobj_B, Hobj_R);

        /*
         * Close the reply queue.
         */
         C_options = MQCO_NONE;
         MQCLOSE(Hcon,
                 &Hobj_R,
                 C_options,
                 &CompCode,
                 &Reason);
     }

    /*
     * Close the bridge request queue.
     */
     C_options = MQCO_NONE;
     MQCLOSE(Hcon,
             &Hobj_B,
             C_options,
             &CompCode,
             &Reason);
 }
```

```
        /*
         * Disconnect from the queue manager.
         */
         MQDISC(&Hcon, &CompCode, &Reason);
     }

     printf("MQTSTBR2: Program ends\n");

    /*
     * End program.
     */
     return(0);
 }


/*************************************************************
 * This function puts a request message on the bridge
 * request queue, and waits for a reply from the bridge.
 * The request is to run transaction "CEMT".
 ************************************************************/
 void Run3270Transaction(MQHCONN hcon,
                         MQHOBJ  hobj_b,
                         MQHOBJ  hobj_r)
 {
    /*
     * Local variables.
     */
     MQLONG     cc;
     MQLONG     rc;
     MQLONG     blen;
     MQLONG     mlen;
     MQLONG     dlen;
     PMQCHAR    reqmsg;
     MQCHAR     repmsg[1000];
     MQMD       md  = { MQMD_DEFAULT };
     MQGMO      gmo = { MQGMO_DEFAULT };
     MQPMO      pmo = { MQPMO_DEFAULT };
     MQCIH      cih = { MQCIH_DEFAULT };

     struct tagCEMT
     {
         MQCIH                cih;
         union
         {
           brmq_vector_header  vh;
           brmq_receive        rv;
         } rec;
         MQCHAR               Data[12];
         union
         {
           brmq_vector_header  vh;
           brmq_converse       cv;
         } con;
     } CEMT;


    /*
     * Prepare MQMD.
     */
     memset(md.ReplyToQ, ' ', 48);
     memcpy(md.ReplyToQ, REPLY_Q, strlen(REPLY_Q));
     memcpy(md.CorrelId, MQCI_NEW_SESSION,
            MQ_CORREL_ID_LENGTH);
     memcpy(md.Format,   MQFMT_CICS, MQ_FORMAT_LENGTH);

    /*
     * Prepare MQCIH.
     */
     memcpy(&CEMT.cih, &cih, sizeof(MQCIH));
     CEMT.cih.LinkType           = MQCLT_TRANSACTION;
     CEMT.cih.ConversationalTask = MQCCT_NO;
     CEMT.cih.UOWControl         = MQCUOWC_ONLY;
     CEMT.cih.Encoding           = MQENC_NATIVE;
     CEMT.cih.CodedCharSetId     = 850;
     memcpy(CEMT.cih.TransactionId, "CEMT", strlen("CEMT"));
     memcpy(CEMT.cih.Format, "CSQCBDCI", MQ_FORMAT_LENGTH);

    /*
     * Prepare RECEIVE vector header.
     */
     CEMT.rec.vh.brmq_vector_length =
         sizeof(brmq_receive) + 12;
     memcpy(CEMT.rec.vh.brmq_vector_descriptor, "0402", 4);
     memcpy(CEMT.rec.vh.brmq_vector_type,       "I   ", 4);
     memcpy(CEMT.rec.vh.brmq_vector_version,    "0000", 4);

    /*
     * Prepare RECEIVE vector.
     */
     memcpy(CEMT.rec.rv.brmq_re_transmit_send_areas,YES,4);
     memcpy(CEMT.rec.rv.brmq_re_buffer_indicator,NO,4);
     memcpy(CEMT.rec.rv.brmq_re_aid, ENTER, 4);
     CEMT.rec.rv.brmq_re_data_len = 12;

    /*
     * Prepare RECEIVE vector data.
     */
     memcpy(CEMT.Data, "CEMT I TASK ", 12);

    /*
     * Prepare CONVERSE vector header.
     */
     CEMT.con.vh.brmq_vector_length = sizeof(brmq_receive);
     memcpy(CEMT.con.vh.brmq_vector_descriptor, "0406", 4);
     memcpy(CEMT.con.vh.brmq_vector_type,       "I   ", 4);
     memcpy(CEMT.con.vh.brmq_vector_version,    "0000", 4);

    /*
     * Prepare CONVERSE vector.
     */
     memcpy(CEMT.con.cv.brmq_co_transmit_send_areas,YES,4);
     memcpy(CEMT.con.cv.brmq_co_aid, PF3, 4);
     CEMT.con.cv.brmq_co_data_len = 0;

    /*
     * Put the request on the bridge request queue.
     */
     reqmsg = (PMQCHAR)&CEMT;
     blen = sizeof(struct tagCEMT);
     MQPUT(hcon, hobj_b, &md, &pmo, blen, reqmsg, &cc, &rc);
```

```
        if ( cc != MQCC_OK )
            return;

       /*
        * Commit the request.
        */
        MQCMIT(hcon, &cc, &rc);

       /*
        * Prepare to get reply.
        */
        mlen = 1000;
        gmo.WaitInterval = 10000;
        gmo.Options = MQGMO_WAIT |
                      MQGMO_CONVERT |
                      MQGMO_ACCEPT_TRUNCATED_MSG;
        memcpy(md.CorrelId, md.MsgId, MQ_CORREL_ID_LENGTH);

       /*
        * Get the reply from the ReplyTo queue.
        */
        MQGET(hcon, hobj_r, &md, &gmo, mlen, &repmsg, &dlen,
              &cc, &rc);

       /*
        * If we got a reply, print ok.
        */
        if ( cc != MQCC_FAILED )
        {
            printf("MQTSTBR2: Transaction ran ok\n");
        }

       /*
        * Return to caller (main).
        */
        return;
    }
```

Java sample

The following sample program is a Java example that shows how an MQ Java client can connect to a VSE queue manager and use MQSeries resources.

```
/**
 * Imports
 */
import java.lang.*;
import com.ibm.mq.*;

/**
 * Main: This program connects to a queue manager as an
 *       MQ client, and puts a message to a target queue.
 *       This program illustrates how a java program can
 *       access VSE queue manager resources.
 */
public static void main(String args[])
{
    int                 opts;
    String              hostname  = "9.185.191.27";
    String              channel   = "WIN1.CLI.VSE8";
    String              qManager  = "VSE.QM1";
    String              targQueue = "ANYQ";
    MQQueueManager      qMgr;
    MQQueue             tq;
    MQMessage           msg;
    MQPutMessageOptions pmo;


    MQEnvironment.hostname = hostname;
    MQEnvironment.port     = 1414;
    MQEnvironment.channel  = channel;
    MQEnvironment.CCSID    = 850;

    try
    {
        System.out.println("Program starting");

        System.out.println("Connecting to queue manager");

        qMgr = new MQQueueManager(qManager);

        System.out.println("Opening target queue");

        opts = MQC.MQOO_OUTPUT;
        tq   = qMgr.accessQueue(targQueue,opts,qManager,
                                null,null);
        msg  = new MQMessage();
        msg.format = MQC.MQFMT_STRING;
        msg.writeChars("Hello from Java");

        pmo = new MQPutMessageOptions();

        System.out.println("Putting message to target queue");

        tq.put(msg, pmo);

        System.out.println("Closing target queue");
```

```
            tq.close();

            System.out.println("Committing work");

            qMgr.commit();

            System.out.println("Disconnecting from qmgr");

            qMgr.disconnect();

            System.out.println("Program ending");
        }
        catch (MQException ex)
        {
            System.out.println("MQ error: Completion code " +
                               ex.completionCode            +
                               " Reason code "              +
                               ex.reasonCode);
        }
        catch (java.io.IOException ex)
        {
            System.out.println("IO exception: " + ex);
        }
    }
```

Sample JCL

This appendix contains sample JCL that you may find useful. The samples cover the following categories:

- MQ program build JCL
- CICS trace format JCL
- MQSC JCL

There are numerous ways of writing such JCL, so you are not limited to the examples provided. The samples are for reference only.

Program build JCL

The JCL in this section is intended to help you build MQI application programs.

JCL to build a CICS COBOL program

```
* $$ JOB JNM=job,CLASS=c,LDEST=(host,user)
* $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB CCOBBLD PREPROCESSOR STEP
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
$ $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB CCOBBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// EXEC LIBR
   ACCESS S=user.lib
   CATALOG mqprog.OBJ   R=Y EOD=/+
$ $$ END
/*
// OPTION DECK
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM=''
  PROCESS LIB,APOST,NOADV,RENT,NOXREF,BUF(4096),NODYNAM
  PROCESS NOSEQ,TRUNC(OPT),DATA(31)
* $$ END
/*
// EXEC LIBR
 ACCESS SUBLIB=user.lib
 CATALOG mqprog.COBOL R=Y EOD=/+
 *
 * Program source goes here!
 *
/+
/*
// EXEC DFHECP1$,SIZE=512K
 CBL XOPTS(COBOL2 NOSEQ)
* $$ SLI MEM=mqprog.COBOL,S=user.lib
/*
// EXEC IESINSRT
/*
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
/+
/*
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=user.lib
// OPTION CATAL
   PHASE mqprog,*
   INCLUDE mqprog
   INCLUDE DFHELII
// EXEC LNKEDT
/. NOCATAL
#&
$ $$ END
```

```
/*
#*
* $$ END
/*
/&
* $$ EOJ
```

JCL to build a batch COBOL program

```
* $$ JOB JNM=job,CLASS=c,LDEST=(host,user)
* $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB BCOBBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
// JOB BCOBBLD
// EXEC LIBR
   ACCESS S=user.lib
   CATALOG mqprog.OBJ   R=Y EOD=/+
* $$ END
/*
// EXEC LIBR
 ACCESS SUBLIB=user.lib
 CATALOG mqprog.COBOL R=Y EOD=/+
 *
 * Program source goes here!
 *
/+
/*
// OPTION DECK
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC IGYCRCTL,SIZE=IGYCRCTL,PARM='
  PROCESS LIB,APOST,NOADV,RENT,BUF(4096),NODYNAM
  PROCESS NOSEQ,TRUNC(OPT)
* $$ SLI MEM=mqprog.COBOL,S=user.lib
/*
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
/+
/*
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=user.lib
// OPTION CATAL
   PHASE mqprog,*
   INCLUDE mqprog
   INCLUDE MQBIBTCH
// EXEC LNKEDT
/. NOCATAL
#&
* $$ END
/*
/&
* $$ EOJ
```

JCL to build a CICS C program

```
* $$ JOB JNM=job,CLASS=c,LDEST=(host,user)
* $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
```

```
// JOB CCBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
$ $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB CCBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// EXEC LIBR
   ACCESS S=user.lib
   CATALOG mqprog.OBJ   R=Y EOD=/+
$ $$ END
/*
// OPTION DECK
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='SOURCE,RENT,LIST'
* $$ END
/*
// EXEC LIBR
 ACCESS SUBLIB=user.lib
 CATALOG mqprog.C  R=Y EOD=/+
 *
 * C language source goes here!
 *
/+
/*
// EXEC DFHEDP1$,SIZE=512K
* $$ SLI MEM=mqprog.C,S=user.lib
/*
// EXEC IESINSRT
/*
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
/+
/*
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=user.lib
// OPTION CATAL
   PHASE mqprog,*
   INCLUDE mqprog
   INCLUDE DFHELII
// EXEC EDCPRLK
/*
// EXEC LNKEDT
/. NOCATAL
#&
$ $$ END
/*
#*
* $$ END
/*
/&
* $$ EOJ
```

JCL to build a CICS Assembler program

```
* $$ JOB JNM=job,CLASS=c,LDEST=(host,user)
* $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB CASMBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
$ $$ PUN DISP=I,CLASS=8,PRI=9,DEST=(*)
// JOB CASMBLD
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// EXEC LIBR
   ACCESS S=user.lib
   CATALOG mqprog.OBJ   R=Y EOD=/+
$ $$ END
/*
// OPTION DECK
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC ASMA90,SIZE=(ASMA90,64K),PARM='EXIT(LIBEXIT(EDECKXIT)),SIZE(MAXC
               -200K,ABOVE)'
* $$ END
/*
// EXEC LIBR
 ACCESS SUBLIB=user.lib
 CATALOG mqprog.ASSEMBLE R=Y EOD=/+
 *
 * Assembler source goes here!
 *
/+
/*
// EXEC DFHEAP1$,SIZE=512K
*ASM XOPTS(NOEPILOG)
* $$ SLI MEM=mqprog.ASSEMBLE,S=user.lib
/*
// EXEC IESINSRT
/*
// ASSGN SYS005,SYSIPT
// EXEC IESINSRT
/+
/*
// IF $MRC GT 4 THEN
// GOTO NOCATAL
// LIBDEF *,SEARCH=(user.lib,PRD2.MQSERIES,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=user.lib
// OPTION CATAL
   PHASE mqprog,*
   INCLUDE DFHEAI
   INCLUDE mqprog
// EXEC LNKEDT
/. NOCATAL
#&
$ $$ END
/*
#*
* $$ END
/*
/&
* $$ EOJ
```

CICS trace format JCL

The JCL in this section may be helpful for formatting CICS trace data for both CICS/VSE and CICS TS.

JCL to format CICS/VSE V2.3 traces

```
// JOB PRTAUX23 **** Print Aux Trace CICS/VSE 2.3
// DLBL DFHAUXT,'your.cics.auxt.dataset',0,VSAM,CAT=catalog,            *
               DISP=(OLD,KEEP)
// EXEC DFHTUP,SIZE=AUTO
 DEVICE=DISK
/* ALL
/*
/&
```

JCL to format CICS TS V1.1 traces

```
// JOB PRTAUX11 **** Print Aux Trace CICS/VSE 2.3
// DLBL DFHAUXT,'your.cics.auxt.a',0,SD
// EXTENT SYS002,,1,0,3278,15
// DLBL DFHBUXT,'your.cics.auxt.b,0,SD
// EXTENT SYS002,,1,0,33318,15
// ASSGN SYS002,DISK,VOL=volume,SHR
// LIBDEF *,SEARCH=(IJSYSRS.SYSLIB,                                     X
               PRD2.SCEEBASE),TEMP
*  ALL
// EXEC DFHTU410,SIZE=1M,PARM='ABBREV'
/*
/&
```

MQSC sample JCL

The JCL in this section shows how to use the MQPMQSC utility to define, alter, and display VSE queue manager objects. The MQPMQSC utility is run as a batch job and uses the MQSeries for VSE Batch Interface (see section 8.2.1, “Batch interface overview” on page 151).

JCL to define a queue

```
// JOB MQSCRUN
// SETPARM MQBISRV='MQBISERV'
// LIBDEF *,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPMQSC,SIZE=MQPMQSC
*
*  Define a local queue.
*
DEFINE QLOCAL(EXAMPLE.Q)                       +
       DESCR('Example queue')                  +
       CICSFILE(MQF0001)                       +
       USAGE(NORMAL)                           +
       SHARE                                   +
       PUT(ENABLED)                            +
       GET(ENABLED)                            +
*
       MAXDEPTH(5000)                          +
       MAXMSGL(2000)                           +
       MAXQUSER(100)                           +
       MAXGLOCK(200)                           +
       MAXLLOCK(200)                           +
*
       NOTRIGGER                               +
       MAXTRIGS(1)                             +
       NOTRIGREST                              +
*
       QSVCIEV(NONE)                           +
       QSVCINT(0)                              +
       QDPMAXEV(DISABLED)                      +
       QDPHIEV(DISABLED)                       +
       QDEPTHHI(0)                             +
       QDPLOEV(DISABLED)                       +
       QDEPTHLO(0)                             +
*
       REORG(DISABLED)                         +
       REORGINT(0000)                          +
       REORGTI(0000)
/*
/&
```

JCL to alter the queue manager

```
// JOB MQSCRUN
// SETPARM MQBISRV='MQBISERV'
// LIBDEF *,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPMQSC,SIZE=MQPMQSC
*
*  Alter the queue manager.
*
ALTER QMGR                                     +
      MAXMSGL(8192)                          +
      MAXDEPTH(10000)                        +
      MAXHANDS(100)                          +
      MAXQOPEN(100)                          +
      MAXQUSER(100)                          +
      MAXGLOCK(1000)                         +
      MAXLLOCK(1000)
/*
/&
```

JCL to display a channel

```
// JOB MQSCRUN
// SETPARM MQBISRV='MQBISERV'
// LIBDEF *,SEARCH=(PRD2.MQSERIES,PRD2.SCEEBASE)
// EXEC MQPMQSC,SIZE=MQPMQSC
*
*  Display a channel's attributes.
*
DISPLAY CHANNEL(WIN1.TO.VSE8)                +
        SENDEXIT                             +
        SENDDATA                             +
        RCVEXIT                              +
        RCVDATA                              +
        SCYEXIT                              +
        SCYDATA                              +
        MSGEXIT                              +
        MSGDATA
/*
/&
```

D

Sample communication configuration

LU6.2 configuration

In this section, we provide several sample configurations, all of which are related to MQSeries intercommunication between MQSeries for VSE and MQSeries for OS/2.

Even if you are not using OS/2 as your remote system, many of the principles and settings discussed in this section will be applicable.

CM/2 configuration file MQSERIES.NDF

The following definitions are only a subset of the NDF file. We have included definitions immediately relevant to APPC communication between VSE and OS/2. The NDF file is relevant to CM/2 configuration on OS/2.

The PU name, APPC LU name, and CICS applid ID we use are DEP390, DEP3900, and CICSP390 respectively.

```
DEFINE_LOCAL_CP  FQ_CP_NAME(VTAM1.DEP390      )
                 CP_ALIAS(DEP390  )
                 NAU_ADDRESS(INDEPENDENT_LU)
                 NODE_TYPE(EN)
                 NODE_ID(X'05D00390')
                 NW_FP_SUPPORT(NONE)
                 HOST_FP_SUPPORT(YES)
                 HOST_FP_LINK_NAME(HOST0001)
                 MAX_COMP_LEVEL(NONE)
                 MAX_COMP_TOKENS(0);

DEFINE_LOGICAL_LINK  LINK_NAME(HOST0001)
                     FQ_ADJACENT_CP_NAME(VTAM1.SSCP02      )
                     ADJACENT_NODE_TYPE(LEN)
                     DLC_NAME(IBMTRNET)
                     ADAPTER_NUMBER(0)
                     DESTINATION_ADDRESS(X'40005A0C7DD9')
                     ETHERNET_FORMAT(NO)
                     CP_CP_SESSION_SUPPORT(NO)
                     SOLICIT_SSCP_SESSION(YES)
                     NODE_ID(X'05D00390')
                     ACTIVATE_AT_STARTUP(YES)
                     USE_PUNAME_AS_CPNAME(NO)
                     LIMITED_RESOURCE(USE_ADAPTER_DEFINITION)
                     LINK_STATION_ROLE(USE_ADAPTER_DEFINITION)
                     MAX_ACTIVATION_ATTEMPTS(USE_ADAPTER_DEFINITION)
                     EFFECTIVE_CAPACITY(USE_ADAPTER_DEFINITION)
                     COST_PER_CONNECT_TIME(USE_ADAPTER_DEFINITION)
                     COST_PER_BYTE(USE_ADAPTER_DEFINITION)
                     SECURITY(USE_ADAPTER_DEFINITION)
                     PROPAGATION_DELAY(USE_ADAPTER_DEFINITION)
                     USER_DEFINED_1(USE_ADAPTER_DEFINITION)
                     USER_DEFINED_2(USE_ADAPTER_DEFINITION)
                     USER_DEFINED_3(USE_ADAPTER_DEFINITION);

DEFINE_LOCAL_LU  LU_NAME(DEP3900 )
                 LU_ALIAS(DEP3900 )
                 NAU_ADDRESS(INDEPENDENT_LU);

DEFINE_PARTNER_LU  FQ_PARTNER_LU_NAME(VTAM1.CICSP390    )
                   PARTNER_LU_ALIAS(CICSP390)
                 PARTNER_LU_UNINTERPRETED_NAME(CICSP390)
                 MAX_MC_LL_SEND_SIZE(32767)
                 CONV_SECURITY_VERIFICATION(NO)
                 PARALLEL_SESSION_SUPPORT(YES);

DEFINE_PARTNER_LU_LOCATION  FQ_PARTNER_LU_NAME(VTAM1.CICSP390   )
                            WILDCARD_ENTRY(NO)
                            FQ_OWNING_CP_NAME(VTAM1.SSCP02      )
                            LOCAL_NODE_NN_SERVER(NO);

DEFINE_MODE  MODE_NAME(NORMAL  )
             DESCRIPTION(Connexion avec CICSAPP)
             COS_NAME(#CONNECT)
             DEFAULT_RU_SIZE(NO)
             MAX_RU_SIZE_UPPER_BOUND(5000)
             RECEIVE_PACING_WINDOW(63)
             MAX_NEGOTIABLE_SESSION_LIMIT(10)
             PLU_MODE_SESSION_LIMIT(10)
             MIN_CONWINNERS_SOURCE(5)
             COMPRESSION_NEED(PROHIBITED)
             PLU_SLU_COMPRESSION(NONE)
             SLU_PLU_COMPRESSION(NONE);

DEFINE_MODE  MODE_NAME(#INTER  )
             COS_NAME(#INTER  )
             DEFAULT_RU_SIZE(YES)
             RECEIVE_PACING_WINDOW(7)
             MAX_NEGOTIABLE_SESSION_LIMIT(256)
             PLU_MODE_SESSION_LIMIT(256)
             MIN_CONWINNERS_SOURCE(4)
             COMPRESSION_NEED(PROHIBITED)
             PLU_SLU_COMPRESSION(NONE)
             SLU_PLU_COMPRESSION(NONE);

DEFINE_DEFAULTS  IMPLICIT_INBOUND_PLU_SUPPORT(YES)
                 DEFAULT_MODE_NAME(BLANK)
                 DEFAULT_LOCAL_LU_ALIAS(DEP3900 )
                 MAX_MC_LL_SEND_SIZE(32767)
                 DIRECTORY_FOR_INBOUND_ATTACHES(*)
                 DEFAULT_TP_OPERATION(NONQUEUED_AM_STARTED)
                 DEFAULT_TP_PROGRAM_TYPE(BACKGROUND)
                 DEFAULT_TP_CONV_SECURITY_RQD(NO)
                 MAX_HELD_ALERTS(10);

DEFINE_TP  TP_NAME(AMQCRS6A)
           PIP_ALLOWED(NO)
           FILESPEC(D:\MQM\BIN\AMQCRS6A.EXE)
           PARM_STRING(-n AMQCRS6A -m DCT.MQ.OS2)
           CONVERSATION_TYPE(ANY_TYPE)
           CONV_SECURITY_RQD(NO)
           SYNC_LEVEL(EITHER)
           TP_OPERATION(NONQUEUED_AM_STARTED)
           PROGRAM_TYPE(FULL_SCREEN)
           RECEIVE_ALLOCATE_TIMEOUT(INFINITE);

DEFINE_CPIC_SIDE_INFO  SYMBOLIC_DESTINATION_NAME(MQ01    )
                       DESCRIPTION(MQ/Series VSE Start channel Transact.)
                       PARTNER_LU_ALIAS(CICSP390        )
                       MODE_NAME(NORMAL  )
                       TP_NAME(MQ01);
```

```
  START_ATTACH_MANAGER;
```

VSE VTAM definitions

The following definitions are relevant to APPC LU 6.2 connections for a VSE system to communicate with OS/2. These definitions are books utilized by VTAM during normal operation.

VTAM book: XCA.B

```
TXCA252   VBUILD TYPE=XCA
POCA252   PORT   CUADDR=252,ADAPNO=1,MEDIUM=RING,SAPADDR=4,TIMER=60
GXCA252   GROUP   DIAL=YES,                                         X
                      ISTATUS=ACTIVE,                               X
                      ANSWER=ON,                                    X
                      CALL=INOUT,                                   X
                      DYNPU=YES,                                    X
                      DYNPUPFX=U3
LXCA252   LINE
PXCA252   PU
```

VTAM book: PS2.B

```
**                                                                 **
*********************************************************************
*    WARP Work station (P/390 Server) definitions                  *
*     1 APPC Connection + 3 x 3270 emulation sessions              *
*********************************************************************
SW       VBUILD TYPE=SWNET,            REQUIRED                      X
               MAXNO=12,               REQUIRED                      X
               MAXGRP=5
*
*
DEP390   PU    ADDR=C1,                COULD BE ANYTHING (NOT USED)  X
               IDBLK=05D,                                            X
               IDNUM=00390,            PC 3274 EMULATOR              X
               DISCNT=NO,                                            X
               IRETRY=NO,              NOT USED                      X
               ISTATUS=ACTIVE,                                       X
               LANACK=(0,0),                                         X
               LANCON=(5,2),                                         X
               LANINACT=4.8,                                         X
               LANRESP=(5,2),                                        X
               LANSDWDW=(7,1),                                       X
               LANSW=YES,                                            X
               MACADDR=40005A0E0AB1,                                 X
               MAXDATA=4000,                                         X
               MAXOUT=7,               NOT USED FOR /LAN             X
               MAXPATH=4,                                            X
               NETID=VTAM1,                                          X
               PACING=0,                                             X
               PUTYPE=2,                                             X
               SAPADDR=4,                                            X
               SSCPFM=USSSCS,                                        X
               VPACING=0
**
PATH     PATH  DIALNO=000440005A0C7DD9,                              X
               GRPNM=GXCA252,                                        X
               GID=1,PID=1,                                          X
               USE=YES
**
DEP3900  LU    LOCADDR=0,                FOR APPC LU6.2               X
               DLOGMOD=#INTER,           Single session               X
               ISTATUS=ACTIVE
DEP390L1 LU    LOCADDR=1,                FOR 3270 EMULATOR            X
               LOGAPPL=SYS1VSCS,                                      X
               DLOGMOD=D4C32XX3,                                      X
               ISTATUS=ACTIVE
DEP390L2 LU    LOCADDR=2,                FOR 3270 EMULATOR            X
               LOGAPPL=SYS1VSCS,                                      X
               DLOGMOD=D4C32XX3,                                      X
               ISTATUS=ACTIVE
DEP390L3 LU    LOCADDR=3,                FOR 3270 EMULATOR            X
               LOGAPPL=SYS1VSCS,                                      X
               DLOGMOD=D4C32XX3,                                      X
               ISTATUS=ACTIVE
DEP390L4 LU    LOCADDR=4,                FOR THE PC EMULATOR          X
               LOGAPPL=CICSP390,                                      X
               DLOGMOD=D4C32XX3,                                      X
               ISTATUS=ACTIVE
```

VTAM book: ADJ.B

```
*/* ADJ/SYS1                                                  USIBMPC
*/* ------------------------------------------------------------------
*/*
*/*  ADJACENT SSCP DEFINITION
*/*
*/* ------------------------------------------------------------------
*
         VBUILD  TYPE=ADJSSCP
*
* --------------------- USED WHEN NETWORK IS USIBMPC AND CDRM IS KNOWN
*
         NETWORK NETID=VTAM1
*
SSCP01   CDRM
SSCP02   ADJCDRM
```

VTAM book: CDRM.B

```
VBUILD TYPE=CDRM
*
          NETWORK NETID=VTAM1
*
SSCP01    CDRM SUBAREA=1,ELEMENT=1,                                   -
               ISTATUS=ACTIVE,CDRDYN=YES,CDRSC=OPT
SSCP02    CDRM SUBAREA=2,ELEMENT=1,                                   -
               ISTATUS=ACTIVE,CDRDYN=YES,CDRSC=OPT
```

APPC modetab entries

Your CICS CONNECTION definitions for LU 6.2 connections specify a *modename*. The modename identifies a modetab. For reference, we have included the JCL and source to build a modetab.

```
// JOB LOGMDE Assembly of MODETAB
// LIBDEF PHASE,CATALOG=PRD2.CONFIG
// OPTION CATAL
// EXEC ASSEMBLY
        PUNCH ' PHASE APPCTAB,*'
```

```
APPCTAB  MODETAB
*-------------------------------------------------------------------*
*                 LOGMODE for APPC                                  *
*-------------------------------------------------------------------*
NORMAL     MODEENT LOGMODE=NORMAL,      AGW  (SLU) TO AGW (PLU)        X
               PSNDPAC=X'00',           PRIMARY SEND PACING COUNT      X
               SRCVPAC=X'00',           SECONDARY RECEIVE PACING COUNT X
               SSNDPAC=X'00',           SECONDARY SEND PACING COUNT    X
               TYPE=0,                  NEGOTIATED BIND                X
               FMPROF=X'13',            FM PROFILE 19 LU 6.2           X
               TSPROF=X'07',            TS PROFILE 7 LU 6.2            X
               PRIPROT=X'B0',           PRIMARY NAU PROT               X
               SECPROT=X'B0',           SECONDARY NAU PROT             X
               RUSIZES=X'8B8B',         8 X 2**B                       X
               COMPROT=X'50B1',         COMMON NAU PROT                X
               PSERVIC=X'060200000000000000002C00'
#BATCH   MODEENT LOGMODE=#BATCH,FMPROF=X'13',TSPROF=X'07',      *@KFA* *
               ENCR=B'0000',SSNDPAC=3,RUSIZES=X'F7F7',      *@KFA**@04C*
               SRCVPAC=3,PSNDPAC=3 APPNCOS=#BATCH   *@KFA**@KGC**@T3C*
         TITLE '#INTER'                                         *@KFA*
*-------------------------------------------------------------------*
*                                                                   *
*        LOGMODE TABLE FOR INTERACTIVE SESSIONS ON RESOURCES        *
*        CAPABLE OF ACTING AS LU 6.2 DEVICES                        *
*                                                               @KFA*
*                                                               @KFA*
*-------------------------------------------------------------------*
#INTER   MODEENT LOGMODE=#INTER,FMPROF=X'13',TSPROF=X'07',      *@KFA* *
               ENCR=B'0000',SSNDPAC=7,RUSIZES=X'F7F7',      *@KFA**@04C*
               SRCVPAC=7,PSNDPAC=7 APPNCOS=#INTER   *@KFA**@KGC**@T3C*
         TITLE '#BATCHSC'                                       *@KFA*
*-------------------------------------------------------------------*
*                                                                   *
*        LOGMODE TABLE FOR BATCH SESSIONS REQUIRING SECURE          *
*        TRANSPORT ON RESOURCES CAPABLE OF ACTING AS LU 6.2         *
*        DEVICES                                                    *
*                                                               @KFA*
*-------------------------------------------------------------------*
#BATCHSC MODEENT LOGMODE=#BATCHSC,FMPROF=X'13',TSPROF=X'07',    *@KFA* *
               ENCR=B'0000',SSNDPAC=3,RUSIZES=X'F7F7',      *@KFA**@04C*
               SRCVPAC=3,PSNDPAC=3 APPNCOS=#BATCHSC *@KFA**@KGC**@T3C*
         TITLE '#INTERSC'                                       *@KFA*
*-------------------------------------------------------------------*
*                                                                   *
*        LOGMODE TABLE FOR INTERACTIVE SESSIONS REQUIRING           *
*        SECURE TRANSPORT ON RESOURCES CAPABLE OF ACTING AS         *
*        LU 6.2 DEVICES                                             *
*                                                               @KFA*
*-------------------------------------------------------------------*
#INTERSC MODEENT LOGMODE=#INTERSC,FMPROF=X'13',TSPROF=X'07',    *@KFA* *
               ENCR=B'0000',SSNDPAC=7,RUSIZES=X'F7F7',      *@KFA**@04C*
               SRCVPAC=7,PSNDPAC=7 APPNCOS=#INTERSC *@KFA**@KGC**@T3C*
         MODEEND
         END
/*
// EXEC LNKEDT
// OPTION CATAL
// EXEC ASSEMBLY
        PUNCH ' PHASE RDBMODES,*'
RDBMODES MODETAB
```

```
IBMRDB   MODEENT LOGMODE=IBMRDB,TYPE=0,PSNDPAC=X'00',SRCVPAC=X'00',    S
               SSNDPAC=X'02',RUSIZES=X'8989',                          S
               FMPROF=X'13',TSPROF=X'07',                              S
               PRIPROT=X'B0',SECPROT=X'B0',COMPROT=X'50A5',            S
               PSERVIC=X'060200000000000000122F00'
         MODEEND
         END
/*
// EXEC LNKEDT
/&
```

TCP/IP configuration

In this section, we provide a sample TCP/IP configuration table, related VTAM configuration, and CICS definitions for TCP/IP.

TCP/IP configuration table

The configuration table is read by TCP/IP during initialization. When you start TCP/IP, you specify a number that maps to IPINITnn, where nn is a unique number that identifies the appropriate configuration file.

```
*-------------------------------------------*
*                                           *
*        Define systems parameters          *
*                                           *
*-------------------------------------------*
*
*--- Get more console message for debugging purposes
SET DEBUG=OFF
*--- Set severity levels for messages to be displayed
SET MESSAGE CRITICAL=ON
SET MESSAGE IMPORTANT=ON
SET MESSAGE VITAL=ON
SET MESSAGE WARNING=OFF
SET MESSAGE INFORMATION=OFF
MESSAGE MSGID=IPI103,CONSOLE=NO
*---- my own IP address
SET IPADDR   = 1.2.3.4
*---- I define 8192 hosts for my subnet
SET MASK     = 000.255.224.000
*----  Maximum idle time before regain control. Set to 100 seconds
SET ALL_BOUND        = 30000
*----  Maximum data size sent before waiting for all ackknowledgments
SET WINDOW           = 4096
*----  Number of 32K Buffers per FTP Deamon
SET TRANSFER_BUFFERS = 20
*----  Time before retransmitting packets set to 300 msec.
SET RETRANSMIT       = 300
*----  Maximum time slice for pseudo tasks. Set to 100 msec.
SET DISPATCH_TIME    = 100
*----  Interval time before trying to redispatch pseudotasks if
*----  if the current one is running not interruptable code. Set to 30ms
SET REDISPATCH       = 30
*----  Don't ask for shutdown confirmation
SET DOWNCHECK        = OFF
*----  Remote FTP clients may initiate transfers
SET ISOLATION        = OFF
```

```
*----  No need to validate passwords
SET SECURITY         = OFF
*--- Enable CGI Written in REXX
* SET REXX_SUPPORT=ON
*----  I'm not a Gateway.
GATEWAY OFF
*----
*----
*-------------------------------------------*
*                                           *
*        Define the Communication Links     *
*                                           *
*-------------------------------------------*
DEFINE LINK,ID=RESEAU,TYPE=3172,DEV=600,MTU=1500
DEFINE ADAPTER,LINKID=RESEAU,NUMBER=0,TYPE=TOKEN_RING
*
*-------------------------------------------*
*                                           *
*        Define Routine Information         *
*                                           *
*-------------------------------------------*
DEFINE ROUTE,ID=IBMNET,LINKID=RESEAU,IPADDR=0.0.0.0,GATEWAY=1.2.3.5
*-------------------------------------------*
*                                           *
*        Define Telnet Daemons              *
*                                           *
*-------------------------------------------*
DEFINE TELNETD,ID=LU,TCPAPPL=TELNLU,TARGET=CICSP390,PORT=23,COUNT=4
*-------------------------------------------*
*                                           *
*        Define FTP Daemons                 *
*                                           *
*-------------------------------------------*
DEFINE FTPD,ID=FTP,PORT=21,COUNT=2,UNIX=NO
*-------------------------------------------*
*                                           *
*        Define WEB Daemons                 *
*                                           *
*-------------------------------------------*
DEFINE HTTPD,ID=WEB,PORT=80,ROOT='PRD2.WEB',CONFINE=NO
*                                           *
*        Line Printer Daemons               *
*                                           *
*-------------------------------------------*
DEFINE LPD,PRINTER=FAST,QUEUE='POWER.LST.A'
DEFINE LPD,PRINTER=FASTLIB,QUEUE='PRD2.STN'
DEFINE LPD,PRINTER=LOCAL,QUEUE='POWER.LST.A',LIB=PRD2,SUBLIB=STN
*-------------------------------------------*
*                                           *
*        Automated Line Printer Client      *
*                                           *
*-------------------------------------------*
DEFINE EVENT,ID=LST_LISTEN,TYPE=POWER,CLASS=X,QUEUE=LST,ACTION=LPR
*-------------------------------------------*
*                                           *
*        Setup the File System              *
*                                           *
*-------------------------------------------*
* DEFINE FILESYS,LOCATION=SYSTEM,TYPE=PERM
*
```

```
*
DEFINE FILE,PUBLIC='ICCF',DLBL=DTSFILE,TYPE=ICCF
DEFINE FILE,PUBLIC='SYSTEM',DLBL=IJSYSRS,TYPE=LIBRARY
DEFINE FILE,PUBLIC='PRD1',DLBL=PRD1,TYPE=LIBRARY
DEFINE FILE,PUBLIC='PRD2',DLBL=PRD2,TYPE=LIBRARY
DEFINE FILE,PUBLIC='POWER',DLBL=IJQFILE,TYPE=POWER
DEFINE FILE,PUBLIC='VSESPUC',DLBL=VSESPUC,TYPE=VSAMCAT
DEFINE FILE,PUBLIC='VSAM.FILE1',DLBL=TCPIP1,TYPE=KSDS,LRECL=80
DEFINE FILE,PUBLIC='VSECOM',DRIVER=VSECOM,TYPE=CGI-REXX
*
*-------------------------------------------*
*                                           *
*   Alias Definition                        *
*                                           *
*-------------------------------------------*
*
DEFINE NAME,NAME=DCTP3VSE,IPADDR=1.2.3.6
DEFINE NAME,NAME=DCTP3VM,IPADDR=1.2.3.5
DEFINE NAME,NAME=DCTP3OS2,IPADDR=1.2.3.7
*-------------------------------------------*
DEFINE USER,ID=GEGE,PASSWORD=XXXXXXXX
DEFINE USER,ID=USER1,PASSWORD=YYYYYYYY
/+
/*
```

VTAM Telnet configuration

The TCP/IP configuration file identifies the Telnet service as follows:

```
DEFINE TELNETD,ID=LU,TCPAPPL=TELNLU,TARGET=CICSP390,PORT=23,COUNT=4
```

The TCPAPPL value should match Telnet definitions in VTAM. In this example we use the name TELNLU and a count of 4. This means you should define VTAM APPL definitions for TELNLU00 to TELNLU04 as follows:

```
TCPAPPL  VBUILD TYPE=APPL
TELNLU00 APPL   AUTH=(ACQ)
TELNLU01 APPL   AUTH=(ACQ)
TELNLU02 APPL   AUTH=(ACQ)
TELNLU03 APPL   AUTH=(ACQ)
TELNLU04 APPL   AUTH=(ACQ)
```

CICS definitions for TCP/IP

TCP/IP provides a number of services executable from CICS. CSD definitions are required in CICS if you intend to exploit these services.

The following entries can be defined using the DFHCSDUP program:

```
DEFINE PROGRAM($EDCTCPV) GROUP(CEE)     LANGUAGE(C)         RSL(PUBLIC)
DEFINE PROGRAM(TELNET01) GROUP(TCPIP)   LANGUAGE(ASSEMBLER) RSL(PUBLIC)
DEFINE PROGRAM(FTP01)    GROUP(TCPIP)   LANGUAGE(ASSEMBLER) RSL(PUBLIC)
DEFINE PROGRAM(CLIENT01) GROUP(TCPIP)   LANGUAGE(ASSEMBLER) RSL(PUBLIC)
DEFINE TRANSACTION(TELN) GROUP(TCPIP)   PROGRAM(TELNET01)   RSL(PUBLIC)
DEFINE TRANSACTION(TELC) GROUP(TCPIP)   PROGRAM(TELNET01)   RSL(PUBLIC)
DEFINE TRANSACTION(TELW) GROUP(TCPIP)   PROGRAM(TELNET01)   RSL(PUBLIC)
DEFINE TRANSACTION(TELR) GROUP(TCPIP)   PROGRAM(TELNET01)   RSL(PUBLIC)
DEFINE TRANSACTION(FTP)  GROUP(TCPIP)   PROGRAM(FTP01)      RSL(PUBLIC)
DEFINE TRANSACTION(FTPC) GROUP(TCPIP)   PROGRAM(FTP01)      RSL(PUBLIC)
DEFINE TRANSACTION(FTPW) GROUP(TCPIP)   PROGRAM(FTP01)      RSL(PUBLIC)
DEFINE TRANSACTION(FTPR) GROUP(TCPIP)   PROGRAM(FTP01)      RSL(PUBLIC)
DEFINE TRANSACTION(LPR)  GROUP(TCPIP)   PROGRAM(CLIENT01)   RSL(PUBLIC)
DEFINE TRANSACTION(TCPC) GROUP(TCPIP)   PROGRAM(CLIENT01)   RSL(PUBLIC)
DEFINE TRANSACTION(TCPW) GROUP(TCPIP)   PROGRAM(CLIENT01)   RSL(PUBLIC)
DEFINE TRANSACTION(TCPR) GROUP(TCPIP)   PROGRAM(CLIENT01)   RSL(PUBLIC)
ADD GROUP(TCPIP) LIST(VSELIST)
```

The TCP/IP group (TCPIP) should be added to the appropriate group list for your CICS system. In our example, we use VSELIST.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 316. Note that some of the documents referenced here may be available in softcopy only.

- *TCP/IP Tutorial and Technical Overview*, GG24-3376
- *TCP/IP Solutions for VSE/ESA with OpenConnect Systems*, SG24-4270
- *The Native TCP/IP Solution for VSE*, SG24-2041

Other publications

These publications are also relevant as further information sources:

- *An Introduction to Messaging and Queuing*, GC33-0805
- *CICS for VSE/ESA Application Programming Guide*, SC33-0712
- *CICS for VSE/ESA Application Programming Reference*, SC33-0713
- *CICS for VSE/ESA CICS-Supplied Transactions*, SC33-0710
- *CICS for VSE/ESA Customization Guide*, SC33-0707
- *CICS for VSE/ESA Performance Guide*, SC33-0703
- *CICS for VSE/ESA Resource Definition (Macro)*, SC33-0709
- *CICS for VSE/ESA Resource Definition (Online)*, SC33-0708
- *CICS Transaction Server for VSE/ESA V1R1 Application Programming Guide*, SC33-1657
- *CICS Transaction Server for VSE/ESA V1R1 Application Programming Reference*, SC33-1658
- *CICS Transaction Server for VSE/ESA V1R1 CICS-Supplied Transactions*, SC33-1655
- *CICS Transaction Server for VSE/ESA V1R1 Performance Guide*, SC33-1667
- *CICS Transaction Server for VSE/ESA V1R1 Resource Definition Guide*, SC33-1653
- *Debug Tool for VSE/ESA Installation and Customization Guide*, SC26-8798
- *LE V1R4 C Run-Time Programming Guide*, SC33-6688
- *LE V1R4 Debugging Guide and Run-Time Messages*, SC33-6681
- *Language Environment V1R4 Installation and Customization*, SC33-6682
- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Clients*, GC33-1632
- *MQSeries Event Monitoring*, SC34-5760

- *MQSeries for VSE System Management Guide*, GC34-5364
- *MQSeries Intercommunication*, SC33-1369
- *MQSeries MQSC Command Reference*, SC33-1369
- *MQSeries Planning Guide*, GC33-1349
- *MQSeries Programmable System Management*, SC33-1482
- *MQSeries System Administration*, SC33-1873
- *TCP/IP for VSE/ESA V1R5 IBM Program Setup and Supplementary Information*, SC33-6601
- *VSE/ESA V2R4 Guide to System Functions*, SC33-6711
- *VSE/ESA V2R6 System Control Statements*, SC33-6713
- *VSE/VSAM V6R4 Commands*, SC33-6731
- *VSE/VSAM V6R4 User's Guide and Application Programming*, SC33-6732

Online resources

These Web sites and URLs are also relevant as further information sources:

- MQSeries for VSE/ESA home page

  http://www.ibm.com/software/ts/mqseries/platforms/vseesa/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Glossary

Abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with MQSeries for OS/390. A complete list of MQSeries for VSE abend reason codes and their explanations is contained in the *MQSeries for VSE System Management Guide*, GC34-5364.

Address space. The area of virtual storage available for a particular job.

Advanced Program-to-Program Communication (APPC). The general facility characterizing the LU 6.2 architecture and its various implementations in products.

Alias queue object. An MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

APAR. Authorized Program Analysis Report. A report of a problem caused by a suspected defect in a current, unaltered release of a program.

APPC. Advanced Program-to-Program Communication.

Asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *Synchronous messaging*.

Backout. An operation that reverses all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *Commit*.

Batch auto-start. A queue manager parameter used to indicate whether the MQSeries batch interface should be started automatically during system initialization.

Batch identifier. An XPCC identifier used to uniquely identify a queue manager to batch applications.

Browse. In message queuing, to use the MQGET call to copy a message without removing it from the queue. See also *Get*.

BSM. Basic Security Manager.

CCSID. Coded Character Set Identifier.

CEDF. CICS Execution Diagnostic Facility.

CGI. Common Gateway Interface.

Channel. See *Message channel*.

Channel event. An event indicating that a channel instance has become available or unavailable. Channel events are generated on the queue managers at both ends of the channel.

CI. Control Interval.

CICS bridge. See *MQSeries–CICS bridge*.

Cipher specification. A name or abbreviated code that identifies an encryption method used when processing SSL-enabled channels.

Client. A runtime component that provides access to queuing services on a server for local user applications. The queues used by the applications reside on the server. See also *MQSeries client*.

Client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

Client connection channel type. The type of MQI channel definition associated with an MQSeries client. See also *Server connection channel type*.

COA. Confirm-on-arrival. In reply queue processing, a reply message can be generated when a message is initially put to a queue by using the COA report option in the message descriptor of an object message.

COD. Confirm-on-delivery. In reply queue processing, a reply message can be generated when a message is initially read from a queue by using the COD report option in the message descriptor of an object message.

Coded Character Set Identifier (CCSID). The name of a coded set of characters and their code point assignments.

Command. In MQSeries, an instruction that can be carried out by the queue manager.

Command processor. An MQSeries program responsible for processing PCF messages. The command processor validates and executes PCF commands, and generates response messages to the issuer.

Command server. An MQSeries program responsible for processing the system command queue. The command server reads PCF message from the command queue and starts an instance of the MQSeries command processor to process the PCF message.

Command server auto-start. A queue manager parameter used to indicate whether the MQSeries command server should be started automatically during system initialization.

Commit. An operation that applies all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *Backout*.

Completion code. A return code indicating how an MQI call has ended.

Configuration file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, MQSeries for Windows XP, and MQSeries for VSE, a file that contains configuration information related to, for example, logs, communications, or installable services.

Connect. To provide a queue manager connection handle that an application uses on subsequent MQI calls. The connection is made either by the MQCONN call, or automatically by the MQOPEN call.

Connection handle. The identifier or token by which a program accesses the queue manager to which it is connected.

Control Interval (CI). A fixed-length area of direct access storage in which VSAM stores records and creates distributed free spaces. The control interval is the unit of information that VSAM transmits to or from direct access storage.

CSD. CICS System Definition.

DCT. Destination Control Table.

Dead Letter Queue (DLQ). A queue to which a queue manager or application sends messages that cannot be delivered to their correct destination.

Dead letter queue handler. An MQSeries-supplied utility that monitors a Dead Letter Queue (DLQ) and processes messages on the queue in accordance with a user-written rules table.

DISCINT. The disconnect interval parameter. This is a value associated with sender channels on other MQ platforms. The value is specified in seconds.

Distributed application. In message queuing, a set of application programs that can each be connected to a different queue manager but that collectively constitute a single application.

Distributed Queue Management (DQM). In message queuing, the setup and control of message channels to queue managers on other systems.

DLA. Disk Label Area.

DLQ. Dead Letter Queue.

DQM. Distributed Queue Management.

Dual logging. A method of recording MQSeries for VSE activity, where each change is recorded on two data sets, so that if a restart is necessary, and one data set is unreadable, the other can be used. Contrast with *Single logging*.

Event data. In an event message, the part of the message data that contains information about the event (such as the queue manager name, and the application that gave rise to the event). See also *Event header*.

Event header. In an event message, the part of the message data that identifies the event type of the reason code for the event.

Event log. See *Event queue*.

Event message. Contains information (such as the category of event, the name of the application that caused the event, and queue manager statistics) relating to the origin of an instrumentation event in a network of MQSeries systems.

Event queue. The queue onto which the queue manager puts an event message after it detects an event. Each category of event (queue manager, performance, or channel event) has its own event queue.

Exit. A program called at defined places in the processing carried out by the queue manager or MCA programs.

External Security Manager (ESM). A security product that is invoked by the System Authorization Facility (SAF) interface.

FCT. File Control Table.

FIQ. First in Queue.

Get. In message queuing, to use the MQGET call to remove a message from a queue. See also *Browse*.

GSD. Global System Definition.

Handle. See *Connection handle* and *Object handle*.

IBM. International Business Machines Corporation.

Instrumentation event. In MQSeries, an event is a logical combination of conditions that is detected by a queue manager or channel instance.

IP Address. Internet Protocol address. Usually a four-part dotted decimal value that uniquely identifies a remote host, for example, 1.20.33.444.

Interactive System Productivity Facility (ISPF). An IBM z/OS® licensed program that serves as a full-screen editor and dialog manager. It is used for writing application programs and provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user.

ISPF. Interactive System Productivity Facility.

Internet Protocol (IP). A protocol used to route data from its source to its destination in an Internet environment. This is the base layer on which other protocol layers, such as TCP and UDP, are built.

ITSO. International Technical Support Organization.

IUI. Interactive User Interface.

JCL. Job Control Language.

JCT. Journal Control Table.

LIQ. Last in Queue.

Listener. In MQSeries distributed queuing, a program that monitors for incoming network connections.

Local definition. An MQSeries object belonging to a local queue manager.

Local definition of a remote queue. An MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

Local queue. A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *Remote queue*.

Local queue manager. The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called remote queue managers, even if they are running on the same system as the program.

Logical Unit of Work (LUW). See *Unit of work*.

LU 6.2. A type of Logical Unit (LU) that supports general communication between programs in a distributed processing environment.

MCA. Message Channel Agent. The programs responsible for sending and receiving data to and from other queue managers.

Message. (1) In message queuing applications, a communication sent between programs. See also *Persistent message* and *Nonpersistent message*. (2) In system programming, information intended for the terminal operator or system administrator.

Message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a receiver at the other end) and a communication link. Contrast with *MQI channel*.

Message Channel Agent (MCA). A program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue. See also *Message queue interface*.

Message Channel Interface (MCI). The MQSeries interface to which customer or vendor-written programs that transmit messages between an MQSeries queue manager and another messaging system must conform. A part of the MQSeries Framework.

Message Descriptor (MD). Control information describing the message format and presentation that is carried as part of an MQSeries message. The format of the message descriptor is defined by the MQMD structure.

Message exit. An exit program called during channel operation following the retrieval of a message from a queue and prior to a message being placed on a queue.

Message expiry. Message attribute identifying a period of time expressed in tenths of a second. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

Message queue. Synonym for queue.

Message Queue Interface (MQI). The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

Message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

Message Sequence Number (MSN). A programming technique in which messages are given unique numbers during transmission over a communication link. This enables the receiving process to check whether all messages are received, to place them in a queue in the original order, and to discard duplicate messages.

Messaging. See *Synchronous messaging* and *Asynchronous messaging*.

MQI. Message Queue Interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system and transfers only MQI calls and responses in a bidirectional manner. Contrast with *Message channel*.

MQMT. MQSeries Master Terminal.

MQSC. See *MQSeries Commands*.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries–CICS bridge. A client/server facility that supports requests to run CICS DPL programs and 3270 transactions from applications running on remote, non-CICS, environments.

MQSeries Commands (MQSC). MQSC commands are verb-based text commands, uniform across all platforms, that are used to manipulate or display MQSeries objects.

MQSeries objects. Contrast with *Programmable Command Format (PCF)*.

MQSM. MQSeries System Monitor.

Nonpersistent message. A message that does not survive a restart of the queue manager. Contrast with *Persistent message*.

Object. In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

Object descriptor. A data structure that identifies a particular MQSeries object. Included in the descriptor are the name of the object and the object type.

Object handle. The identifier or token by which a program accesses the MQSeries object with which it is working.

PCF. See *Programmable Command Format (PCF)*.

PCT. Program Control Table.

Performance event. A category of event indicating that a limit condition has occurred.

Persistent message. A message that survives a restart of the queue manager. Contrast with *Nonpersistent message*.

Ping. In distributed queuing, a diagnostic aid that uses the exchange of a test message to confirm that a message channel or a TCP/IP connection is functioning.

PKI. Public Key Infrastructure. The PKI infrastructure includes X.509 digital certificates used by SSL services.

Platform. In MQSeries, the operating system under which a queue manager is running.

Port. A unique communications identifier used by TCP/IP programs to establish a conversation with a specific application. The target application binds a TCP/IP socket to the unique port number and then waits for connection requests for the port from remote hosts.

Programmable Command Format (PCF). A data message containing an MQSeries command and associated parameters. PCF messages are written to the system command queue.

Program Temporary Fix (PTF). A solution or bypass of a problem diagnosed by IBM field engineering as the result of a defect in a current, unaltered release of a program.

PSP. Product Service Planning.

Queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages that point to other queues or be used as models for dynamic queues.

Queue Manager (QM). (1) A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *Local queue manager* and *Remote queue manager*. (2) An MQSeries object that defines the attributes of a particular queue manager.

Queue manager event. An event that indicates:

- An error condition has occurred in relation to the resources used by a queue manager. For example, a queue is unavailable.
- A significant change has occurred in the queue manager. For example, a queue manager has stopped or started.

QSN. Queue Sequence Number.

RBA. Relative Byte Address.

Reason code. A return code that describes the reason for the failure or partial success of an MQI call.

Receive exit. An exit program called immediately following the receipt of data over an active channel.

Receiver channel. In message queuing, a channel that responds to a sender channel, takes messages from a communication link, and puts them on a local queue.

Relative Byte Address (RBA). The displacement in bytes of a stored record or control interval from the beginning of the storage space allocated to the data set to which it belongs.

Remote queue. A queue belonging to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *Local queue*.

Remote queue manager. To a program, a queue manager that is not the one to which the program is connected.

Remote queue object. See *Local definition of a remote queue*.

Remote queuing. In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

Reply message. A type of message used for replies to request messages. Contrast with *Request message* and *Report message*.

Reply-to queue. The name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

Report message. A type of message that gives information about another message. A report message can indicate that a message has been delivered, has arrived at its destination, has expired, or could not be processed for some reason. Contrast with *Reply message* and *Request message*.

Requester channel. In message queuing, a channel that may be started remotely by a sender channel. The requester channel accepts messages from the sender channel over a communication link and puts the messages on the local queue designated in the message. See also *Server channel*.

Request message. A type of message used to request a reply from another program. Contrast with *Reply message* and *Report message*.

Resynch. In MQSeries, an option to direct a channel to start up and resolve any in-doubt status messages, but without restarting message transfer.

Return codes. The collective name for completion codes and reason codes.

Rollback. Synonym for back out.

RSL. Resource Security Level.

SAF. See *System Authorization Facility*.

Security exit. An exit program called during the establishment of a channel.

Send exit. An exit program called prior to the transmission of data over an active channel.

Sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

Sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

Sequential number wrap value. In MQSeries, a method of ensuring that both ends of a communication link reset their current message sequence numbers at the same time. Transmitting messages with a sequence number ensures that the receiving channel can reestablish the message sequence when storing the messages.

Server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *Client*.

Server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

Server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *Client connection channel type*.

Single logging. A method of recording MQSeries for OS/390 activity where each change is recorded on one data set only. Contrast with *Dual logging*.

Single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

Single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *Two-phase commit*.

SIT. System Initialization Table.

SNA. Systems Network Architecture.

Socket. A communications handle used by TCP/IP programs to send data to, and receive data from, a remote host.

SSL. Secure Sockets Layer. A integrated feature of the TCP/IP product that provides a set of services to secure e-business transactions, including data encryption and X.509 certificate exchange.

Store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

Supervisor Call (SVC). An OS/390 instruction that interrupts a running program and passes control to the supervisor so that it can perform the specific service indicated by the instruction.

Synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *Asynchronous messaging*.

Syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

System Authorization Facility (SAF). An interface between the VSE operating system and external security managers. The SAF interface is used for security purposes.

System command queue. The system command queue is a communication parameter of the global system definition and identifies the target queue for PCF command messages.

System reply queue. The system reply queue is a communication parameter of the global system definition and identifies the target queue for MQSC response messages.

System Initialization Table (SIT). A table containing parameters used by CICS on startup.

System log. In MQSeries, a file recording the work done by queue managers while they receive, transmit, and deliver messages.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

TCP. Transmission Control Protocol.

TCP/IP. Transmission Control Protocol/Internet Protocol.

Thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

TP. Transaction Program.

Transaction. See *Unit of work* and *CICS transaction*.

Transaction identifier. In CICS, a name that is specified when the transaction is defined and that is used to invoke the transaction.

Transmission Control Protocol (TCP). Part of the TCP/IP protocol suite. A host-to-host protocol between hosts in packet-switched communications networks. TCP provides connection-oriented data stream delivery. Delivery is reliable and orderly.

Transmission Control Protocol/Internet Protocol (TCP/IP). A suite of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

Transmission program. See *Message channel agent*.

Transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

Trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

Triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

Trigger message. A message containing information about the program that a trigger monitor is to start.

Trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

Two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *Single-phase commit*.

UDP. User Datagram Protocol.

Undelivered message queue. See *Dead letter queue*.

Unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *Unit of work*.

Unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user requested syncpoint or at the end of a transaction. Contrast with *Unit of recovery*.

User Datagram Protocol (UDP). Part of the TCP/IP protocol suite. A packet-level protocol built directly on the Internet Protocol layer. UDP is a connectionless and less reliable alternative to TCP. It is used for application-to-application programs between TCP/IP host systems.

VCTCA. Virtual Channel-to-Channel Adapter.

XPCC. Cross Partition Communication Calls.

X.509. X.509 is the standard used for the generation and interpretation of PKI certificates.

Index

IBM

Redbooks

Using MQSeries for VSE

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Using MQSeries for VSE

Implementing MQSeries for VSE

Connecting MQSeries for VSE to WebSphere MQ

Building applications using MQSeries for VSE

This IBM Redbook will help you get started with MQSeries for VSE V2R1.2. It explores why and how to use MQSeries with the VSE operating system and shows why MQSeries is more than just another queuing system. This book is meant for those who know VSE but have only a slight understanding of MQSeries and how to use it.

This redbook contains an overview of the MQSeries installation and configuration, explains how to connect clients to servers and servers to servers, how to trigger applications, and the MQI programming interface. It describes the command interface, how the libraries are used, and how to edit, compile, test, and debug a program.

This redbook also provides information about performance and security for the MQSeries for VSE V2.1.2. It explains what has to be done to use MQSeries for communication between VSE and other platforms, such as OS/390 and Windows XP. Examples are given to guide the user through the administrative process of defining connections between different systems, and information regarding security, backup, and recovery is also provided.

SG24-5647-01 ISBN 0738491888