Envoy Message Queuing

version 1.3

Programmer's Guide

For use with Microsoft Message Queue services (MSMQ) software

ENVOY TECHNOLOGIES

Envoy Message Queuing version 1.3

Programmer's Guide

January, 2002

Envoy Technologies Inc.
Corporate Headquarters
120 Wood Avenue South
Iselin, NJ, 08830, USA
Phone: 732-549-6500
Fax: 732-549-3165
Web: ***http://www.envoytech.com***
Tech Support: **support@envoytech.com**

Envoy MQ Programmer's Guide

Contents

Chapter 1

Overview

Envoy MQ is an external gateway for the Microsoft Message Queue Server (MSMQ) environment. Envoy MQ extends MSMQ's capabilities from the Microsoft Windows operating system to other operating systems, and enables you to connect systems such as UNIX, IBM AS/400, IBM CICS, and Unisys ClearPath to an MSMQ enterprise network.

Envoy MQ provides a fast and reliable messaging interface between the Windows operating systems and other systems. Envoy MQ enables all your applications to use MSMQ store-and-forward messaging. This allows applications to communicate across a network even if the applications are not running at the same time.

Developed by Envoy Technologies in collaboration with Microsoft Corporation, Envoy MQ works together with MSMQ in the simplest and most efficient possible way.

Envoy MQ Client and Connector

Envoy MQ is a client/server system running in a distributed network environment. The Envoy MQ Connector (the server) component runs on Microsoft Windows platforms. To use Envoy MQ Connector, you must also install the appropriate Envoy MQ Client components, which interface your applications with Envoy MQ Connector and MSMQ.

Envoy MQ Clients are available for many operating systems and with APIs in several programming languages, for example:

Operating systems	API languages
UNIX flavors: Sun Solaris Tru64 UNIX HP UNIX IBM AIX Linux HP e3000 SCO UNIX DEC OpenVMS	C
IBM OS/400	C, RPG, COBOL

For technical and licensing information regarding the Envoy MQ Client components, please contact Envoy Technologies or see our web site, *http://www.envoytech.com*.

There is also a Envoy MQ Client for Unisys ClearPath systems. Please direct inquiries about this product and its support to your Unisys representative.

How you can use Envoy MQ

You can use Envoy MQ to create applications that run on non-Windows operating systems and that communicate with each other or with Windows systems by the MSMQ message queuing method.

If you are a C or C++ programmer, you can use MSMQ and Envoy MQ to create portable messaging applications that run on Windows, UNIX, and other systems in a network.

If you are an AS/400 or CICS programmer, you can use Envoy MQ to interface COBOL and RPG programs with MSMQ. The underlying functionality of Envoy MQ and MSMQ is independent of the platform or programming language. Applications can communicate in the same efficient way on all systems in your network.

MSMQ and Envoy MQ let you concentrate on application development without worrying about networking details. The simple—yet extraordinarily flexible—interface of MSMQ and Envoy MQ lets you design programs that communicate efficiently, with a minimum of effort.

How to use this book

This book is a learning and reference manual for Envoy MQ application programming. The chapters of this book explain:

- The basic concepts of MSMQ and Envoy MQ operation
- How to install and configure Envoy MQ Client
- How to program messaging applications using the Envoy MQ API

✓ *The API examples in this book are written in the C programming language. For information on API programming in other languages such as RPG and COBOL programming, please see the documentation for the Envoy MQ Clients running on specific operating systems.*

Other Envoy MQ documentation

The following Envoy MQ documentation is available:

Document	Description
Envoy MQ Connector Administrator's Guide	Explains how to set up and monitor the server component of the Envoy MQ system
Envoy MQ Programmer's Guide (this book)	Programming messaging applications using Envoy MQ
Envoy MQ Client Installation Guide	Installation Notes for each of the supported platforms
Envoy MQ Client for OS/400	Additional Platform notes for OS/400. Explains RPG and Cobol interface for EMQ on 0S/400 platform
Envoy MQ Client for HP e3000	Additional platform notes for HP e3000. Explains C and Cobol interface on HP e3000

MSMQ documentation

In addition to this book, you should consult the Microsoft MSMQ documentation and SDK online help.

Online help

On Windows systems, you can display online help about many aspects of Envoy MQ operation.

You can access the help information from the `Envoy MQ` folder on the Start menu, or within the Envoy MQ MMC.

Chapter 2

How Envoy MQ Works with MSMQ

MSMQ is Microsoft's message queuing system running under Microsoft Windows in a network environment.

Envoy MQ extends MSMQ functionality to other operating systems. In order to use Envoy MQ effectively it is important that you understand:

- The basic concepts of MSMQ
- The main features that MSMQ offers to application programmers
- The components of Envoy MQ
- The way in which Envoy MQ interfaces to MSMQ and the network

Introduction to MSMQ

MSMQ offers extraordinary capabilities for fast, reliable communication between Windows-based applications. The following paragraphs summarize some of the main MSMQ features.

Connection-less, asyn-chronous messaging

Applications can communicate using MSMQ without logging on to a remote system or establishing a session with each other. The computers on which the applications run do not even need to be connected at the instant when messages are written or read. Applications can continue running without waiting for transmission to be completed.

Guaranteed delivery and deliver once

MSMQ provides mechanisms by which an application can guarantee and confirm that messages are delivered and prevent duplicate delivery.

Message prioritization

MSMQ applications can specify how network resources should be allocated to messages of different types.

User-defined message structure

An MSMQ message may contain a single byte (or no message contents at all), a text string, or a long and complex data structure. The message may be encoded or encrypted in any syntax that the communicating applications understand.

Optional message properties An application can send or receive properties such as the message size, label, sender id, priority, delivery options, etc. along with the message body. The properties are stored in a compact, efficient data structure.

Independence, privacy, and security MSMQ can provide private, access-controlled queues for each application, or public queues accessed by many applications. Any number of queues can be active simultaneously.

Dynamic queue routing Administrators can change queue locations, protocols, and other queue properties without affecting messaging applications.

Transaction support A program can package send-message or receive-message operations in a transaction. The entire transaction is canceled and rolled back if any of the operations fail.

API MSMQ operates on the Application Layer of the *ISO Reference Model for Open System Interconnection*. MSMQ provides a simple interface between each application program and the network and frees the application programmer from concern about network or communication details.

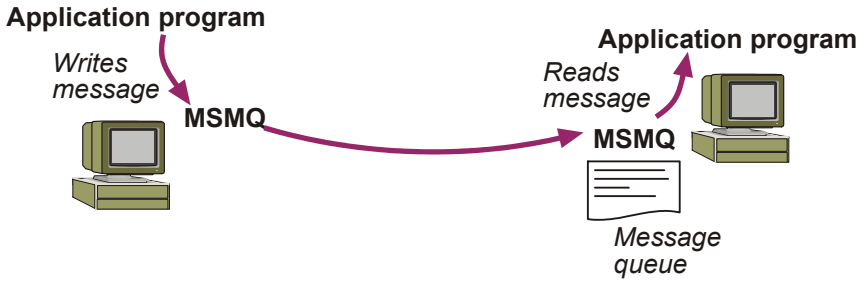
Principles of MSMQ operation

For complete information on MSMQ principles, see the Microsoft MSMQ documentation. The following discussion briefly introduces the concepts that are needed to program Envoy MQ applications.

Message queues The basic concepts of the MSMQ API are *message* and *message queue*.

Message A set of data that needs to be transmitted from an application to another application on the same or a different computer in a network.

Message queue A location where messages are stored, which can be written and read by applications.



Message properties A message may comprise one or more *message properties*.

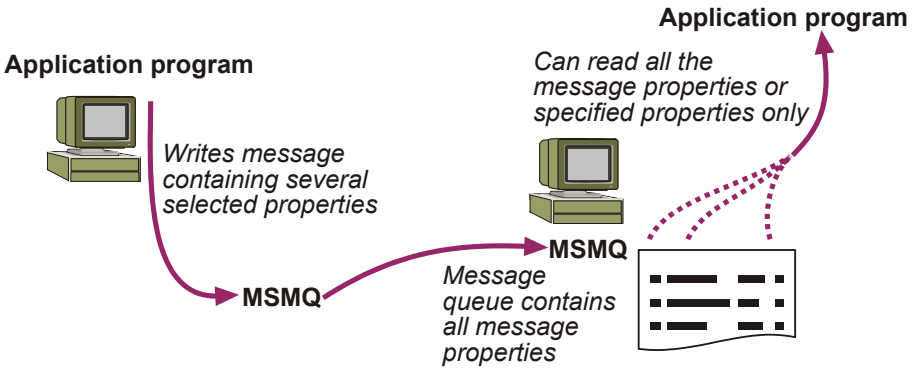
Message property A field of a message that is recognized by MSMQ.

Typical message properties are *message body*, *size of message body*, *label*, *priority*, *sender id*, etc. MSMQ represents each message in an efficient data structure that uses only enough memory for the included properties.

To create a message, an application specifies the message properties and supplies the property values. The application then issues an MSMQ API call to send the message.

To receive a message, an application issues an API call that reads the message from the queue. The application specifies which properties to read; any other properties in the message are ignored.

Of course, the message body may contain its own internal structure, which is recognized only by the sending and receiving applications and not by MSMQ. MSMQ does not restrict the content of the message body.



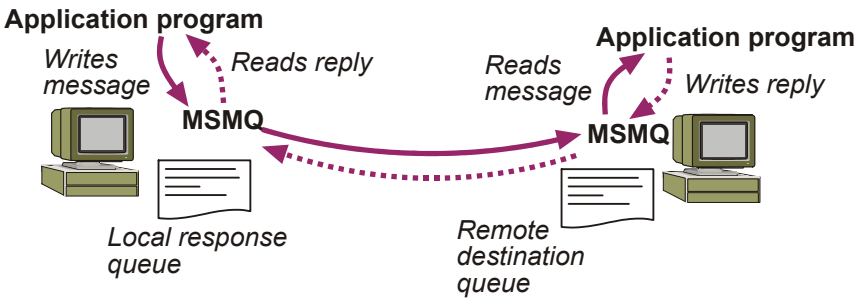
Queue locations and names

Applications interact with MSMQ via the MSMQ API. An application can create or open a message queue on any computer where MSMQ is running. The application can send (write) or receive (read) messages on queues it has opened.

An application creates a queue by its *path name*, for example `.\queue1.mq` (a *local queue* residing on the same computer as the application) or `machine2\queue1.mq` (a *remote queue* residing on a different computer). MSMQ assigns additional queue identifiers, such as a *format name* and a *GUID* (Global *U*nique *Id*entifier) code, which are used to identify queues uniquely throughout the network.

Destination and response queues

The sending application writes to a *destination queue*, which is typically *remote* to the sender and *local* to the receiving application. The receiving application reads the message from the destination queue. If a reply is required, the message may contain a *response queue* name, which is typically *local* to the sending application. The receiving application writes its reply to the response queue, which is then read by the sending application.



Connection-less messaging

This type of communication is called *connectionless* messaging because the sending and receiving applications do not need to establish a connection in order to transmit messages. The sending application can write a message to a queue even if the receiving application is not running. Later, the receiving application can read the queue and send a reply even if the sending application is not running.

MSMQ maintains temporary storage for messages that cannot be sent immediately. Thus an application can write a message to a queue on a computer that is not currently connected to the network. When a network connection is later established, MSMQ automatically forwards the message to the remote destination queue, without any further intervention by the user or an application program. This process is called *store and forward* messaging.

Example

To understand how MSMQ works, consider the following example.

The European sales manager of an American corporation needs to communicate with a database located at the corporate headquarters in Los Angeles.

During the day, the sales manager enters data and queries on a laptop computer offline. Her laptop application calls MSMQ API functions, which send messages to a remote destination queue in Los Angeles. The application is unaware that the laptop is not currently connected to Los Angeles. MSMQ simply stores the messages temporarily on her laptop disk until a connection is established.

In the afternoon, she dials into the London branch office. The London computer is not currently connected to Los Angeles. MSMQ automatically forwards the messages to a second temporary location on the London computer.

In the evening, the London computer connects to Los Angeles and MSMQ forwards the messages to the destination queue. The database application in Los Angeles calls MSMQ API functions to read messages from the queue, completing the transmission.

If a message requires a reply, the process is reversed. MSMQ in Los Angeles transmits to London, which stores the reply temporarily until the sales manager next dials in. The reply is then transmitted to a local queue on her laptop, where the application program can read it.

Purpose of Envoy MQ

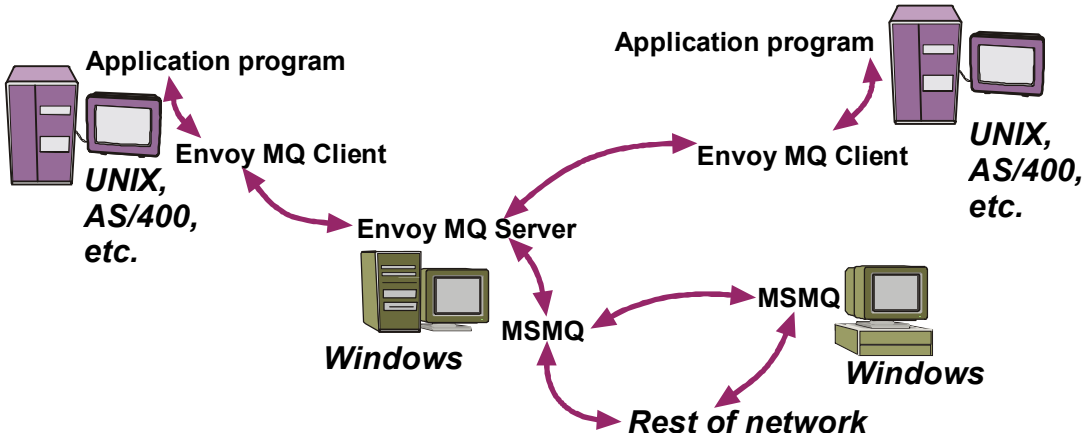
MSMQ is supported under the Microsoft Windows operating system. In a network containing both Windows and other operating systems, MSMQ provides messaging services only between the Windows systems on the network.

The Envoy MQ system is a supplemental product that extends MSMQ functionality to non-Windows platforms such as UNIX, AS/400, Open VMS and Java.

- Envoy MQ enables applications on non-Windows systems to use the MSMQ API and access MSMQ queues.

- Envoy MQ provides an interface between non-Windows systems and MSMQ.
- Communications between Windows systems on the network are provided by MSMQ.

Envoy MQ enables all your network applications to exploit the full power of MSMQ.



Envoy MQ components

A Envoy MQ installation comprises Connector and Client components.

Envoy MQ Clients

Envoy MQ Clients interface directly with your messaging applications. Envoy MQ Clients are installed on non-Windows systems and communicate with the Envoy MQ Connector (MQC).

Envoy MQ Connector

You must install Envoy MQ Connector (MQC) on at least one Windows system in the network. MQC provides the interface with MSMQ.

Optionally, you can install MQC on more than one Windows system in the network. Installation of several MQC is recommended for:

- Continuity of service
- Load balancing
- Scalability from a small site to a large multi-site enterprise

Application programming interface

Using the Envoy MQ Client API, you can write application programs for non-Windows platforms that access the MSMQ messaging capabilities.

The API is designed as similar as possible to the MSMQ API running in its native Windows environment. For a discussion of the main differences, see *Differences between Envoy MQ and MSMQ* below.

The native language of the API is C. Envoy MQ Clients with programming interfaces in RPG and COBOL are available for certain platforms such as AS/400 and HP e3000.

How Envoy MQ interacts with MSMQ

Envoy MQ interacts with MSMQ to form a fully integrated messaging network. MSMQ and Envoy MQ applications share the same message queues and exchange messages with each other freely. Thus:

- Envoy MQ and MSMQ applications can access the same queues, anywhere on the network.
- Envoy MQ applications can write, read, and delete queues created by MSMQ applications, and vice versa.
- Routing through Envoy MQ and MSMQ is completely automatic. An application does not specify whether a message is routed through Envoy MQ, MSMQ, or both to its final destination.

Differences between Envoy MQ and MSMQ

Envoy MQ implements the MSMQ API on external, non-Windows systems. In addition, Envoy MQ acts as an agent accessing MSMQ on behalf of the external systems. There are only a few minor differences between the Envoy MQ implementation and the native MSMQ implementation running under Windows (see *Principles of MSMQ operation* above).

Connectionless messaging

MSMQ communicates *asynchronously*. An application that calls the MSMQ API can continue processing without waiting for an acknowledgment or completion of transmission.

The Envoy MQ Connector (MQC) and Client communicate *synchronously*. The Envoy MQ Client initiates a *remote procedure call* to MQC, which must be completed before control is returned to the calling application. The Envoy MQ API functions return an error message if a live Connector-Client connection does not exist.

The limitation is unlikely to be important, provided that the Envoy MQ Client is connected to MQC by a fast, reliable communication link. Beyond the MQC, communication is managed by MSMQ and is completely asynchronous. MSMQ stores the messages temporarily on the Windows system if the destination system is not currently available.

Envoy MQ Server addressing

A Envoy MQ application can send and receive messages via any MQC on your network. You should configure a default MQC connection for each Envoy MQ Client. An application can connect to a non-default MQC by calling an API function.

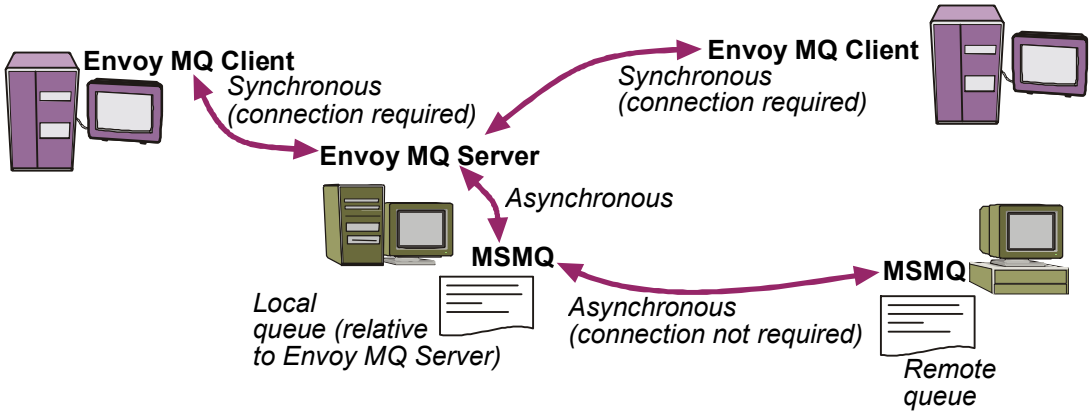
In multithreaded applications, each thread must establish a connection to the MQC before calling any other Envoy MQ API function. The connection can be closed afterwards and reopened to a different MQC.

If a thread loses its connection with the MQC, it can no longer issue Envoy MQ API calls. To continue accessing Envoy MQ, the thread must reopen an MQC connection.

Queue locations and names

MQC maintains all the queues on behalf of Envoy MQ Clients. The queues are physically located on the Windows network.

Local and *remote* queues are defined relative to the MQC location, not to the Envoy MQ Client. For example, the name `.\queue1.mq` refers to a local queue residing on the same computer as MQC. The name `machine2\queue1.mq` refers to a remote queue residing on a different Windows computer from MQC.



Conflicts between local queue names

In MSMQ, the scope of a local queue path name is a single computer. Two MSMQ applications running on different computers can use the same local name (e.g., `.\queue1.mq`) without any possibility of conflict.

The scope of a local name in Envoy MQ is the computer on which the MQC is installed. If several Envoy MQ applications (running on the same or different computers) create queues local to the same MQC, each application should check that the requested name is not already in use.

Scope of handles and cursors

In MSMQ, different threads of the same process can share a single MSMQ queue handle.

Each thread opens a separate channel from the Envoy MQ Client to the MQC. Although different threads can access the same queue, each must do so with its own independent queue handle.

Similar considerations apply to MSMQ locate handles and cursors and to transaction handles. The scope of handles and cursors in Envoy MQ is a single thread.

Asynchronous receive

Envoy MQ does not implement the asynchronous receive (callback) feature of MSMQ, in which MSMQ automatically activates a specified function in an application when a message is received on a queue.

A Envoy MQ application can simulate the callback by periodically checking the queue for a message and activating the desired function when the message is received.

Envoy MQ Connector security

Access to MQC is controlled by user name and password protection. The system administrator must register the authorized non-Windows users in the Windows system. For details, see the *Envoy MQ Connector Administrator's Guide*.

Message security

A Envoy MQ application can call the MSMQ authentication and encryption services to secure a message during transmission. There are two options for authentication:

- The user can previously register a certificate in MSMQ, on the Windows computer where MQC is installed (see the *Envoy MQ Connector Administrator's Guide*.

- A program running on a non-Windows system can register a certificate by calling the Envoy MQ `MQRegisterCertificate` API function.

The authentication and encryption apply only during the transmission between Windows systems. Authentication and encryption are not supported for the portion of the transmission route between the MQC and Clients. If you require authentication or encryption between non-Windows systems, then your application should access its own RSA (or similar) service.

Queue security

Envoy MQ does not extend the MSMQ API for queue access control to non-Windows platforms. You can control the access to individual queues manually using the MSMQ Explorer.

Queue manager properties

The MSMQ property `PROPID_QM_CONNECTION` is not supported.

Transaction support

Envoy MQ Clients provide API functions to begin, commit, or abort an MSMQ transaction. If the transaction is aborted, the Envoy MQ operations are rolled back as are MSMQ operations in the Windows system.

The Envoy MQ transaction functions are similar to those provided by native MSMQ, with certain differences in implementation. In multithreaded applications, the scope of a transaction handle is a single thread. A transaction handle cannot be shared by more than one thread.

Return codes

In general, the Envoy MQ Client API functions return the same error codes as the corresponding MSMQ functions.

Programming

The data structures, API functions, and other aspects of the Envoy MQ API are described in Chapter 4, *Programming Messaging Applications*. A few slight differences in the implementation on different operating systems are described in the documentation for the specific Envoy MQ Clients.

Chapter 3

Installation

This chapter explains how to install, configure, and test Envoy MQ Client under most operating systems.

✓ *For possible minor differences regarding particular systems, please see the documentation for the specific Envoy MQ Client.*

System and network requirements

Operating systems

There are currently Envoy MQ Clients for the following operating systems:

UNIX operating systems

- Sun Solaris 2.5 or higher
- IBM AIX 4.X or higher
- HP-UX 11.x
- SCO-Unix
- Compaq Tru64 Unix 5.X
- Linux

Non-UNIX systems

- IBM OS/400 V3R2 or higher
- Java
- HP e3000 MPE/iX 5.5 or higher
- DEC Alpha OpenVMS 6.2
- DEC VAX OpenVMS 6.2

✓ *Please contact Envoy Technologies or see our Web site (*http://www.envoytech.com*) for information about other operating systems and versions, not listed above.*

Unisys ClearPath systems

There is a version of Envoy MQ Client for Unisys ClearPath systems. Please direct any inquiries about this version or its support to your Unisys representative.

- Unisys HMP NX, NX42XX through NX48XX, using operating system MCP Level 44.1 and higher
- Unisys HMP IX, IX44XX through IX48XX, using operating system OS2200 Level SBR6 or higher

Additional requirements

- TCP/IP communication link to at least one Windows system on which Envoy MQ Server (version 1.2) is installed (for CICS, SNA communication is also supported)
- Free disk space for the Envoy MQ Client software

Where to install

You should install Envoy MQ Client on each non-Windows computer that you want to connect to the MSMQ messaging network.

Installation procedure

Insert the Envoy MQ Client CD-ROM in a Windows system on your network. On the CD-ROM, locate the folder for the non-Windows platform (for example Sun Solaris or AS/400) where you want to install Envoy MQ Client.

Copy all the files in the CD-ROM folder, via FTP or a network drive, to an empty directory on the hard disk of the non-Windows computer. The suggested directory name is `EMQDC`, but you may choose any other name if preferred. Place the directory on the path of users whose programs will make Envoy MQ API calls.

Configuration

Before you can use Envoy MQ Client, you must configure it with parameters such as:

- The connection and logon information for MQC
- A code page that Envoy MQ uses to translate string-valued message properties to UNICODE

Configuration files

FMQROOT
FMQOVERRIDE

To configure Envoy MQ Client, you need to create one or more configuration files on the Envoy MQ Client computer. The following environment variables define the location of the files:

FMQROOT	(Required) The directory location of the root configuration file, which is called FMQ.ENV. The value of FMQROOT should be the Envoy MQ Client directory path, not including a filename.
FMQOVERRIDE	(Optional) The location of an optional, secondary file that supplements and overrides the settings in FMQROOT. Set FMQOVERRIDE to the directory path, including a filename.

For example, on a UNIX system you might define:

```
setenv FMQROOT /home/FMQDC
setenv FMQOVERRIDE /usr/users/jdoe/myenv.env
```

The FMQROOT file contains global default settings for all Envoy MQ applications on the computer. The FMQOVERRIDE file can contain supplementary settings for a particular user or application. For example, if FMQOVERRIDE contains additional Envoy MQ Server connections, an application can connect to any of the Servers defined in either FMQROOT or FMQOVERRIDE. In case of conflict between the settings in the files, the FMQOVERRIDE settings override FMQROOT.

The FMQOVERRIDE file is not required. If it is missing, the system takes all settings from FMQROOT. Likewise, if a particular setting is missing from FMQOVERRIDE, the system takes the setting from FMQROOT. You can create any number of configuration files and switch between them by changing the value of FMQOVERRIDE.

Command line utility to set configuration parameters

You can use the fmqdccfg utility, which is supplied with Envoy MQ Client, to edit the configuration files.

On the command line, enter the fmqdccfg utility with the following command:

```
fmqdccfg -<switch><value> [-<switch><value>] ...
```

✓ *The switches are described in the following paragraphs. For the -srv, -srvd, -cp, and -cpd switches, insert a space character between the switch and the value. All other switches must be followed immediately by a value with no intervening space.*

Configuration filename switch

Use the following switch to specify the configuration filename:

-env	Name and path of the configuration file to edit (default FMQROOT).

Server connection switches

The following switches define a connection to a Envoy MQ Server. You can call fmqdccfg repeatedly to define multiple connections. Later, an application can connect to a Server by specifying the connection name in the FMQConnect API function (see Chapter 4, *Programming Messaging Applications*).

-srv	The connection name.
-node	IP address of the Server.

`-port`	TCP/IP port of the Server.
`-to`	TCP/IP timeout of the Client/Server connection, in seconds (default 30 seconds).

✓ *This timeout is independent of the `MQReceiveMessage()` timeout. Envoy MQ honors an `MQReceiveMessage()` timeout that is longer than the TCP/IP timeout, or even infinite.*

`-dmn, -user, -pwd`	Windows logon information for the MQC: domain name, user name, and password. The password is stored in an encrypted form.

✓ *You should enter the Windows logon information only if you connect to the MQC using the explicit logon method. See* Logon methods *below for an explanation.*

`-def`	Set the default connection for the Envoy MQ Client (specify the connection name, which you should previously define using the `-srv` switch).
`-srvd`	Delete a MQC connection definition from the configuration file.
`-uses`	Where a parameter has not been explicitly defined for a connection, use the parameters of another connection as defaults (specify the second connection name).

Code page switches

Envoy MQ automatically translates string-valued message properties (for example queue names) to UNICODE. For this to work, you need a UNICODE translation table for the code page that your application uses. Use the following switches to download code-page tables from MQC and to manage the tables.

Before you download code-page tables, you should define a default MQC connection using the `-def` switch (see above) or using the `FMQCONNECT` environment variable (see *Environment variables* below). The desired code-page tables must be installed on the Envoy MQ Server (see the *Installation* chapter of the *Envoy MQ Connector Administrator's Guide* for instructions).

You can call `fmqdccfg` repeatedly, using the `-cp` switch, to download and store multiple code-page tables. When an application runs, Envoy MQ Client uses the table for the code page that is currently in effect.

`-cp`	Download a code-page table from the default Envoy MQ Server (specify the code page number and a path on the local machine to store the table).
`-cpd`	Delete a code-page table (specify the code page number).
`-scp`	Select the code-page table that Envoy MQ Client should use to translate to and from UNICODE (specify one of the code page numbers that you downloaded).

Envoy MQ Client uses the system default code page if you do not specify the `-scp` switch.

Switches to display information

The following switches display information:

`-l`	Lists the current settings in the configuration file.
`-?`	Displays help for the `fmqdccfg` utility.

Examples

Define a Envoy MQ Server in a particular configuration file:

```
fmqdccfg –env/usr/users/jdoe/myenv.env -srv newserver2 –
node192.1.1.2 –port1100
```

Define a connection called `newserver`, at IP address 192.1.1.1, port 1100:

```
fmqdccfg -srv newserver –node192.1.1.1 -port1100
```

Set `newserver` as the default connection:

```
fmqdccfg -defnewserver
```

Set the default connection to a timeout of 20 seconds:

```
fmqdccfg -defnewserver -to20
```

Set the Windows logon parameters for a connection:

```
fmqdccfg -srv newserver –dmnEarth –userJDoe –pwdTopSecret
```

Define a connection called `server2`, whose parameters are identical to those of `newserver` except for the IP address.

```
fmqdccfg -srv server2 -usesnewserver -node192.1.1.2
```

Download a translation table for code page 737 and store the table in a specified file:

```
fmqdccfg -cp 737 /usr/users/fmq/codepages/cp737.tbl
```

Set the default code page of `newserver` to 737.

```
fmqdccfg -srv newserver -scp737
```

Define a connection called `server3`, make it the default, download a code-page table, and define the code page as the default, all in a single command:

```
fmqdccfg –srv server3 –node192.1.1.3 –port1100 –defserver3 –
cp 850 /usr/users/fmq/codepages/cp850.tbl –scp850
```

Deleting configuration parameters

The `-srv` and `-cp` switches have a corresponding delete switch. For example, to delete the `newserver` connection:

```
fmqdccfg -srvd newserver
```

Delete the translation table for code page 1249:

```
fmqdccfg -cpd 1249
```

To delete other settings, specify the switch without a value. For example, to delete the port setting of the `newserver` connection:

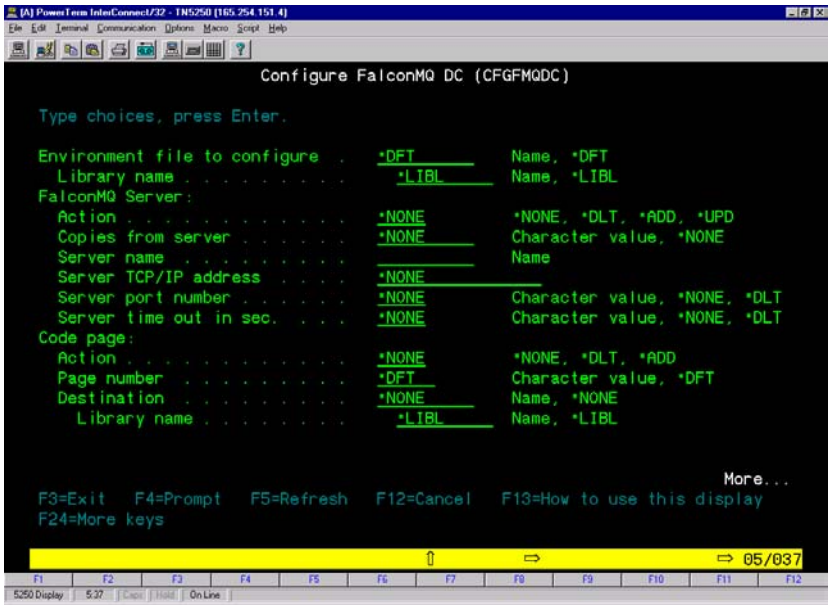
```
fmqdccfg -srv newserver -port
```

To unassign a default connection, specify the `-def` switch without a value. If you do this, and you do not set a default using the `FMQCONNECT` environment variable, then your applications must call the `FMQConnect` function to connect to MQC (see *Connecting to Envoy MQ Connector* in Chapter 4, *Programming Messaging Applications*).

```
fmqdccfg -def
```

Full-screen configuration editor

On some platforms, such as the IBM AS/400, Envoy MQ Client has a full-screen configuration editor. To set the parameters, follow the instructions on the screen or in the documentation for the specific Envoy MQ Client. The parameters are identical to those of the command line `fmqdccfg` utility.



Editing the configuration file manually

You can edit a configuration file manually using any text editor. A connection password must be encrypted, however, so you can enter a password only using the Envoy MQ Client utilities.

Environment variables

Envoy MQ Client uses the `FMQROOT` and `FMQOVERRIDE` environment variables to define the location of configuration files (see *Configuration files* above.)

Optionally, you can define several additional environment variables to configure Envoy MQ Client.

`FMQLOGPATH`	The directory path of the Envoy MQ Client debug log, or `CONSOLE` to send the log to `stdout` (see *Debug logging* in Chapter 4, *Programming Messaging Applications*).
`FMQDEBUG`	Enable debug logging. To enable logging, define `FMQDEBUG` with any value (for example `true`). To disable logging, delete the `FMQDEBUG` definition.
`FMQCONNECT`	The name of the default MQC connection (overrides the default connection in the configuration files).

For example, on a UNIX system you might start debug logging and define a connection as follows:

```
setenv FMQLOGPATH /usr/users/jdoe
setenv FMQDEBUG true
setenv FMQCONNECT newserver
```

You can stop debug logging by deleting the `FMQDEBUG` definition:

```
unsetenv FMQDEBUG
```

Security of Envoy MQ Client applications

Before a Envoy MQ Client application can access an MQC, you must register a Windows user name and password. To register new users or delete existing authorizations, see the *Installation* chapter of the *Envoy MQ Connector Administrator's Guide*.

✓ ***IMPORTANT*** *Register your own user name immediately so you can run the* Installation test*, below.*

Logon methods

There are two ways that a Envoy MQ Client can connect to MQC:

Default logon method — Access is controlled by the user name under which a Envoy MQ application runs on a non-Windows system. The application does not need to send a user name and password explicitly.

Explicit logon method — Access is controlled by a Windows logon. A Envoy MQ application must send a domain name, a user name, and a password that are valid in Windows. MQC verifies the user and password in the specified domain.

If you use the default method, you need to register the user name, together with the MQC prefix, in Windows (see the *Installation* chapter of the *Envoy MQ Connector Administrator's Guide* for instructions). You do not need to specify a domain name, user name, or password (`-dmn`, `-user`, and `-pwd` switches) when you define the MQC connection (see *Configuration* above).

If you use the explicit method, you need to register the user name and password, together with MQC password, in Windows. You must specify a domain name, user name, and password in the Client connection definition.

Installation test

To test the operation of Envoy MQ Client, run the `gwping` and `gwpong` programs supplied with the Envoy MQ software. These programs conduct a *ping-pong* test of the messaging system:

- The `gwping` program sends a *ping* message via Envoy MQ Client and MQC to a message queue.
- The `gwpong` program sends a *pong* reply to a second message queue, where it is read by `gwping`.

✓ *Executable files and the C source code for `gwping` and `gwpong` are supplied in your Envoy MQ samples directory (see Chapter 5,* Sample Application*).*

Default test To run a default test of communication from Envoy MQ Client to MQC and back, run the `gwpong` and `gwping` batch or script files, which are supplied in the Envoy MQ Client samples/pingpong directory for your platform.

✓ *The names of the files may differ on some platforms. On some platforms, such as AS/400, there is a full-screen interface to run the test programs. For details, see the Envoy MQ Client documentation for your platform.*

Open a command window and make your Envoy MQ Client samples/pingpong directory the working directory. To start the `gwpong` program, enter the command:

```
gwpong
```

Open a new command window, and again make your Envoy MQ samples/pingpong directory the working directory. To start the `gwping` program, enter the command:

```
gwping -n 10
```

The `gwping` program sends a sequence of ten test messages, each containing the text `"PING"`, to a queue called `.\PongQ`. The `gwpong` program waits to receive the message, and then sends it back to a queue called `.\PingQ`. The `gwping` program reads the reply from `.\PingQ` and signals you when it is received.

Results

For each of the ten test messages, `gwping` should display *Ping sent* and *Received reply* together with the elapsed time.

In the event of an error, review the installation and configuration of the Envoy MQ Client and MQC.

Optional tests

Optionally, you may enter `gwping` with any of the following command-line switches:

`-n` <*number of iterations*>	Number of ping-pong cycles tested (default = 1).
`-q` <*pong queue*>	An MSMQ queue where `gwping` sends the ping message to be read by `gwpong` (default = `".\PongQ"`).
`-p` <*ping queue*>	An MSMQ queue where `gwping` reads the pong reply (default = `".\PingQ"`).
`-s` <*string to send*>	The text of the test message (default = `"PING"`).
`-l` <*message length*>	Length of the message text. By default, this is the length of the message string (`-s` switch). You may specify a larger value to test memory limitations, etc.
`-r`	Sends messages of random length up to the maximum specified by `-l`.
`-v`	Abbreviated display of test results (default = verbose).

You may enter `gwpong` with the following switches:

`-q` <*pong queue*>	Destination queue for the ping message (must match the `-q` switch of `gwping`, default = `".\PongQ"`).
`-v`	Abbreviated display of test results (default = verbose).
`-c` <*sample size*>	Display results every <*sample size*> messages (default = 1).

For example, to send a single `"PING"` message back and forth between `gwping` and `gwpong`, enter:

(In first command window) `gwpong`

(In second window) `gwping`

To send the message "`hello, world`" five times, enter:

(In first command window) `gwpong`
(In second window) `gwping -n 5 -s "hello, world"`

To test the response of `gwpong` to several `gwping` programs sending messages concurrently:

(In first command window) `gwpong`
(In second window) `gwping -n 10 -p ".\queue1" -s "ping 1"`
(In third window) `gwping -n 10 -p ".\queue2" -s "ping 2"`
(In fourth window) `gwping -n 10 -p ".\queue3" -s "ping 3"`

The `gwpong` program replies to the appropriate response queue for each message that it receives.

```
% gwping -n 100 -s "hello, world"
Attempting to create Pong queue .\PongQ
Pong Queue .\PongQ Exists: 0XC0090006
Pong queue .\PongQ Format name = PUBLIC=798831c1-2b74-11d0-9760-00a024804bc1

Attempting to open Pong queue for send
Pong Queue .\PongQ Opened
Attempting to create ping queue .\PingQ
Ping queue .\PingQ Format name = PUBLIC=798831cc-2b74-11d0-9760-00a024804bc1

Ping Queue .\PingQ Opened
Ping #: 1 "hello, world" sent to queue .\PongQ
Received reply 1: "hello, world"
Ping #: 2 "hello, world" sent to queue .\PongQ
Received reply 2: "hello, world"
...
Total time 17 seconds
press the <Enter> Key...
```

Testing communication to another computer

You can test communication from Envoy MQ Client, via MQC, to another computer.

For example, suppose your network contains a Envoy MQ Client installation, connected by default to a MQC installed on a Windows system called `GATEWAY1`. The network also contains another system running Envoy MQ Client, connected to an MQC on Windows system `GATEWAY2`.

To test the communication, enter the following command on the second computer:

```
gwpong -q ".\PongQ"
```

On the first computer, enter the command:

```
gwping -n 10 -q "GATEWAY2\PongQ"
```

The `-n 10` switch repeats the ping-pong cycle 10 times.

Chapter 4

Programming Messaging Applications

The Envoy MQ Client implements a subset of the MSMQ API on non-Windows operating systems. With a few exceptions, the implementation is identical to MSMQ. Source code written for MSMQ should run on the Envoy MQ Client with little or no adaptation.

Before programming Envoy MQ Client applications, you should be familiar with the Microsoft MSMQ API. Please refer to the Microsoft documentation and SDK online help for information on the data structures, message properties, and API functions defined in MSMQ. The Microsoft documentation provides complete and definitive documentation of the API, which is beyond the scope of this chapter.

In addition, you should review Chapter 2, *How Envoy MQ Works with MSMQ*, especially the section *Differences between Envoy MQ and MSMQ*. That section explains general considerations for programming Envoy MQ applications and for porting MSMQ applications to Envoy MQ.

This chapter documents the specific differences of Envoy MQ Client from MSMQ, including:

- Header files and libraries needed to compile and link Envoy MQ Client applications
- Code page translation
- Error codes
- Compiling and linking
- Minor differences in implementation of the MSMQ API functions

Envoy MQ Client provides several types of API functions that have no direct equivalent in MSMQ:

- Opening and closing MQC connections
- Transaction support

Header files

wintypes.h
mq.h
fmqpubd.h

Include the following header files and definition in your Envoy MQ Client applications. The files are found in your Envoy MQ include directory.

```
#include <wintypes.h>
#include <mq.h>
```

Header file	Description
wintypes.h	Windows data types and other definitions used in the Envoy MQ API.
mq.h	Main API header file. Declares data structures and API functions similar to those of the native MSMQ API.

The Envoy MQ Client directory contains a third header file, called fmqpubd.h, which provides additional Envoy MQ declarations. The mq.h file includes fmqpubd.h, so you don't need to reference fmqpubd.h explicitly in your program code.

Header file	Description
fmqpubd.h	Additional declarations for Envoy MQ API functions that have no analog in MSMQ, such as FMQConnect, FMQDisconnect, FMQCommit, and FMQAbort.

Data structures

MQVAL

The data structures defined in Envoy MQ Client are identical to those of MSMQ.

The most important structures are:

MQMSGPROPS	Contains a set of message properties and their values.
MQQUEUEPROPS	Contains a set of queue properties and their values.
MQQMPROPS	Contains a set of queue manager properties and their values.

`MQPROPVARIANT` Stores a property value.

✓ *For a complete list of the MSMQ data structures and their declarations, see the Microsoft MSMQ documentation. The Microsoft documentation refers to a `PROPVARIANT` structure. `PROPVARIANT` and `MQPROPVARIANT` are equivalent and can be used interchangeably.*

Notation of property value fields

MQPROPVARIANT MQVAL

Named union in the MQPROP VARIANT structure

In MSMQ, the `MQPROPVARIANT` structure uses an unnamed union to store a property value. Unnamed unions are not supported by some C compilers that are commonly used on non-Windows platforms. The Envoy MQ version of `MQPROPVARIANT` therefore uses a named union (called `val`) for the above purpose.

The following table shows the difference between the MSMQ and Envoy MQ declarations of `MQPROPVARIANT`.

In MSMQ	In Envoy MQ

`struct tagMQPROPVARIANT {`
`  /* nonunion fields */`
`  union`
`  { /* union fields */ };`
`  };`
 	
`typedef struct tagMQPROPVARIANT`
`MQPROPVARIANT;`
	`struct tagMQPROPVARIANT {`
	`  /* nonunion fields */`
	`  union`
	`  { /* union fields */ }`
	`val;`
	`  };`
	
	`typedef struct tagMQPROPVARIANT`
	`MQPROPVARIANT;`

Union notation

The use of a named union means that the notation for property value fields differs in MSMQ applications and in Envoy MQ Client applications.

For example, suppose that `PropVar1` is an `MQPROPVARIANT` structure storing the body of a message. The union field for the message body property is called `caub`. In an MSMQ application, you can specify the message body using the following notation:

```
PropVar1.caub
```

In a Envoy MQ application, however, you must specify the name of the union. The Envoy MQ notation is therefore:

```
PropVar1.val.caub
```

Portable MQVAL notation

To write code that runs with both MSMQ and Envoy MQ, use the `MQVAL` macro to specify the name of the union field.

The `MQVAL` macro is defined in the Envoy MQ header file `wintypes.h`, in the following way:

```
#ifdef _WIN32
#  define MQVAL(x) x
#else
#  define MQVAL(x) val.x
#endif
```

For example, you can write the message body field using the following notation:

```
PropVar1.MQVAL(caub)
```

On a Windows system, the compiler converts this notation to `PropVar1.caub`, as required to run with MSMQ. On a non-Windows system, the compiler converts the notation to `PropVar1.val.caub`, as required to run with Envoy MQ.

Code page translation

Envoy MQ and MSMQ communicate using the UNICODE (2-byte) character set. Envoy MQ automatically translates string properties and parameters (for example queue names) from many different code pages to UNICODE, and vice versa.

To enable the translation, you must download the appropriate code-page translation table from MQC. For instructions, see *Configuration* in Chapter 3, *Installation*. Envoy MQ uses the translation table for the code page that is active when your application is running.

Message body property

Envoy MQ Client translates the message body property (`PROPID_M_BODY`) if the message body type property (`PROPID_M_BODY_TYPE`) is `VT_LPWSTR` or `VT_BSTR`.

Envoy MQ Client does not translate a message body of any other type, or a message that does not have a body type property, because it doesn't know whether the body contains text or binary data. Instead, you should program whatever conversions are needed.

Sending a message body with translation

When you send a message body having one of the translated types, Envoy MQ Client converts the contents of the message buffer (`pElems` field) to UNICODE and adjusts the message size indicator (`cElems` field) accordingly.

For example, suppose your Envoy MQ Client application sends a `"hello"` message of 6 bytes (counting the null terminator). The message should have a message body property with the following characteristics:

- A `pElems` buffer of at least 6 bytes, storing the `"hello"` string
- A `cElems` value of 6

The message should also have a body type property with the value `VT_LPWSTR`.

When the application calls `MQSendMessage()` with this message, Envoy MQ Client converts `"hello"` to a 12-byte UNICODE representation. Thus message is stored on the destination queue in the UNICODE representation, with a `cElems` value of 12.

Receiving a message body with translation

When you receive a message body having one of the translated types, Envoy MQ Client converts the `pElems` buffer from UNICODE. Envoy MQ Client does not adjust the `cElems` buffer size indicator.

For example, suppose your application receives a 12-byte UNICODE "hello" message having a body type property of `VT_LPWSTR`. The application must allocate a buffer of at least 12 bytes to receive the message. The application should then call `MQReceiveMessage()` with this buffer. Envoy MQ Client translates the message from UNICODE, so the application actually receives 6 meaningful bytes in the buffer. The received message body still has a `cElems` value of 12.

To determine the required receive buffer size before you receive a message, you can call `MQReceiveMessage()` to peek at the `PROPID_M_BODY_SIZE` property. Allocate the buffer and call `MQReceiveMessage()` again to receive `PROPID_M_BODY`.

Message extension property

Envoy MQ does not translate the message extension property (`PROPID_M_EXTENSION`) because it does not know the internal structure of the extension.

Examples

For source code examples illustrating how to send and receive messages with code-page translation, see Chapter 5, *Sample Application*.

Error handling

Envoy MQ Client returns two types of error or status values identical to MSMQ:

- *Function return values*—returned by API functions
- *Property status values*—reported in the property status fields of messages

In addition, Envoy MQ Client provides an error handling mechanism for problems in Envoy MQ communication or operation.

- *Envoy MQ errors*—returned by API functions as a generic `MQ_ERROR` value

Function return values

`MQ_OK`
`MQ_ERROR_...`
`MQ_INFORMATION_...`

The Envoy MQ Client API functions return the same values as MSMQ API functions, for example:

```
HRESULT hr;

/* Create a queue */
hr = MQCreateQueue(NULL, &qprops, wsFormat, &dwSize);
```

```
/* Check if creation failed, not because queue already exists
*/
if (FAILED(hr))
{ if (hr != MQ_ERROR_QUEUE_EXISTS)
    Error("Cannot create queue.", hr);
}
```

The `MQ_...` return codes are defined in the `mq.h` header file. For a complete list of the return codes and their interpretation, see the Microsoft MSMQ documentation.

Illegal message property error

In MSMQ, if a message property structure contains an invalid property, an API function may succeed and return `MQ_INFORMATION_ILLEGAL_PROPERTY`. In Envoy MQ Client, the function fails and returns `MQ_ERROR_ILLEGAL_PROPID`.

Property status values `aStatus[]`

Errors in setting or reading specific message properties are recorded in the *property status* (`aStatus[]`) fields of the message data structure. The interpretation of these fields is identical in MSMQ and Envoy MQ.

For complete information, see the Microsoft MSMQ documentation.

Envoy MQ errors

Certain errors may occur in Envoy MQ that do not occur in MSMQ. An example is a communication failure between the Envoy MQ Client and MQC.

In the event of such an error, the Envoy MQ Client API functions return the generic MSMQ error `MQ_ERROR`. This error value is almost never returned by MSMQ itself.

Error logging

Envoy MQ Client places error, warning, and information messages in an error log.

On most Envoy MQ platforms, the log file is `log/fmqsys.log`, in the Envoy MQ Client directory. Check the Envoy MQ documentation for your platform to confirm the exact log location.

Debug logging

Optionally, you can enable *debug logging*. If you do this, Envoy MQ Client stores a copy of all error, warning, and information messages in a special debug log.

To enable debug logging, set the `FMQDEBUG` environment variable (see *Environment variables* in Chapter 3, *Installation*) or call the `FMQDebug()` function (see *Enabling debug logging* below in this chapter).

Each process has a separate debug log, named `dc_xxxx.log`, where `xxxx` is the process id (for example `dc_1234.log`). On most Envoy MQ platforms, the default location of debug logs is the `log/user` subdirectory of the Envoy MQ Client directory (check the Envoy MQ documentation for your platform to confirm the exact location). To set a different location, set the `FMQLOGPATH` environment variable or call the `FMQSetLogPath()` function.

Link libraries

Link your Envoy MQ Client application to the Envoy MQ library for your operating system:

Operating system	Link libraries
UNIX	`libfmqdc.a` (referenced as `–lfmqdc`)
Other	Please consult the Envoy MQ Client documentation for your system, supplied with the Envoy MQ software

Sample makefile

The following is a sample makefile for the Sun Solaris version of the `gwping` and `gwpong` programs.

```
CC=cc
CC_FLAGS=-c -I../../include
LINK_FLAGS=-L../../lib -lthread -lfmqdc -lsocket -lnsl
TARGET=gwpong gwping
PONGDEP=gwpong.o
PINGDEP=gwping.o

all: $(TARGET)

clean:
    rm $(PONGDEP) $(PINGDEP) $(TARGET)

.c.o:
    $(CC) $(CC_FLAGS) $< -o $@

gwpong: $(PONGDEP)
    $(CC) -o $@ $(PONGDEP) $(LINK_FLAGS)

gwping: $(PINGDEP)
    $(CC) -o $@ $(PINGDEP) $(LINK_FLAGS)
```

MSMQ API functions

The following is a complete list of MSMQ API functions and their implementation in the Envoy MQ Client. Implementations marked with an asterisk (Yes*) differ in some way from their MSMQ equivalents. For details, see the discussions of specific functions.

For full information on the API functions, see the Microsoft MSMQ documentation.

MSMQ API function	Description	Implemented in Envoy MQ Client?
`MQCreateQueue()`	Create a message queue	Yes*
`MQDeleteQueue()`	Delete a queue	Yes

`MQLocateBegin()`
`MQLocateNext()`
`MQLocateEnd()`
	Search for queues with specified properties	Yes*

`MQOpenQueue()` `MQCloseQueue()`	Access a queue for reading or writing	Yes
`MQSendMessage()`	Write a message to a queue	Yes
`MQReceiveMessage()`	Read a message from a queue	Yes*
`MQCreateCursor()` `MQCloseCursor()`	Enumerate messages on a queue (e.g., to peek at successive messages without deleting)	Yes
`MQSetQueueProperties()` `MQGetQueueProperties()`	Set or retrieve properties of a queue	Yes
`MQSetQueueSecurity()` `MQGetQueueSecurity()`	Set or retrieve the read/write/delete access privileges of individual queues	No
`MQRegisterCertificate`	Register a certificate used to send authenticated messages	Yes*
`MQGetSecurityContext()` `MQFreeSecurityContext()`	Obtain or free a security context handle used to authenticate messages	Yes*
`MQFreeMemory()`	Free allocated memory	Yes

`MQHandleToFormatName()`
`MQPathNameToFormatName()`
`MQInstanceToFormatName()`
	Convert queue identifiers to MSMQ format names	Yes

`MQGetMachineProperties()`	Retrieve properties of the MSMQ Queue Manager	Yes
`MQBeginTransaction()`	Create a new transaction	Yes

Creating a queue — `MQCreateQueue()`

The `pSecurityDescriptor` argument must be either `NULL` (default security policy) or `PSD_SPECIALACCESS_ALL` (defined in `fmqpubd.h`, providing full queue access rights to everyone).

Searching for queues — `MQLocateBegin()`

If this function is called with the `pColumns` argument set to `NULL`, MSMQ returns an `MQ_ERROR_INSUFFICIENT_RESOURCES` error. Envoy MQ Client returns an `MQ_ERROR_ILLEGAL_MQCOLUMNS` error instead.

Receiving a message — `MQReceiveMessage()`

To receive the message body property (`PROPID_M_BODY`), you must supply a receive buffer that is large enough for the entire body. Even if Envoy MQ Client translates the body from UNICODE to the local code page, you must allocate a buffer that is large enough to contain the UNICODE representation (usually twice the size of the code-page representation, see *Code page translation* above).

Envoy MQ Client does not support the asynchronous receive (callback) feature of MSMQ. Therefore the `lpOverlapped` and `fnReceiveCallback` arguments of `MQReceiveMessage()` must be `NULL`.

Queue access privileges — `MQSetQueueSecurity()` `MQGetQueueSecurity()`

These functions are not implemented in Envoy MQ Client. Calls to these functions return `MQ_ERROR`.

Retrieving security context — `MQRegisterCertificate()` `MQGetSecurityContext()` `MQFreeSecurityContext()`

Before you can send authenticated messages, you need to register an certificate on the Windows system where MQC is installed. There are two options for doing this:

- You can register a certificate manually on the Windows system. For the registration procedure, see the *Installation* chapter in the *Envoy MQ Connector Administrator's Guide*.
- A program can register a certificate by calling the Envoy MQ API function `MQRegisterCertificate`. The function operates identically to the MSMQ `MQRegisterCertificate` function, except that the registration is on the remote Windows system and not on the local system.

If you store a certificate on the external system, send it in the `lpCertBuffer` argument of `MQGetSecurityContext()`. Otherwise, the function seeks an internal MSMQ certificate for the Envoy MQ user.

✓ *Authentication is provided for the portion of the transmission route between Windows systems, not for the portion between Windows and non-Windows systems (see* Differences between Envoy MQ and MSMQ *in* Chapter 2, How Envoy MQ Works with MSMQ*).*

API functions for connecting to Envoy MQ Connector

Envoy MQ Client provides the following API functions for connecting to an MQC:

Envoy MQ Client API function	Description
`FMQConnect()`	Open a session with a specified MQC through which Envoy MQ Client routes messages
`FMQDisconnect()`	Close the session to the MQC

Connecting to Envoy MQ Connector `FMQConnect()`

The first time your application calls a Envoy MQ API function, Envoy MQ Client automatically connects to the default MQC.

You can call the `FMQConnect()` function to establish a non-default connection. The connection must be defined in your Client configuration.

For each thread of a process, you must call `FMQConnect()` before any other call to the Envoy MQ API. Otherwise Envoy MQ Client automatically connects to the default MQC.

You cannot call `FMQConnect()` if a connection is already open. You must close the connection by calling `FMQDisconnect()` before calling `FMQConnect()` (see *Disconnecting from Envoy MQ Connector* below). If a connection is lost, you must call `FMQDisconnect()` before re-establishing a connection.

The prototype of `FMQConnect()` is declared in `fmqpubd.h`. The function has no equivalent in MSMQ.

Prototype

```
HRESULT APIENTRY FMQConnect(PCSTR pszQMName, HANDLE* phConnect);
```

Arguments

`pszQMName`	The name of an MQC connection. To define connections, see *Configuration* in Chapter 3, *Installation*. If `NULL`, `FMQConnect` uses the default connection.
`phConnect`	(Output) The connection handle.

Return value

`MQ_OK` for success.

Example

```
HANDLE h;
FMQConnect("newserver", &h);
```

✓ *In Envoy MQ version 1.0, the `FMQConnect()` function accepted Windows logon arguments directly (host, domain, user, and password). This function still exists, but has been renamed `FMQV1Connect()`.*

Although you may substitute `FMQV1Connect()` temporarily in your existing Envoy MQ applications, we suggest that you change to the new `FMQConnect()` syntax. This lets you maintain connection information more easily and ensures compatibility with future versions of Envoy MQ.

Disconnecting from Envoy MQ Connector `FMQDisconnect()`

Use `FMQDisconnect()` to close the session with an MQC. You can call `FMQConnect()` for the same or a different MQC later on (see *Connecting to Envoy MQ Connector* above).

The operating system automatically closes the session with an MQC when your application process ends. In multithreaded applications, however, terminating a thread does not automatically disconnect the MQC You should call the `FMQDisconnect()` function to close the session before you end the thread. Otherwise, subsequent Envoy MQ API calls from another thread may give unexpected results.

The prototype of `FMQDisconnect()` is declared in `fmqpubd.h`. The function has no equivalent in MSMQ.

Prototype

```
HRESULT APIENTRY FMQDisconnect (HANDLE hConnect);
```

Argument

`hConnect`	`NULL`. Optionally (for compatibility with future versions supporting multiple simultaneous connections), you may specify the connection handle returned by `FMQConnect()`.

Return value

`MQ_OK` for success.

Example

```
FMQDisconnect(NULL);
```

API functions supporting MSMQ transactions

Envoy MQ Client provides API functions supporting the MSMQ internal transaction coordinator.

Before calling these functions, you must obtain a transaction handle by calling `MQBeginTransaction()`.

The MQC contains a list of all transactions involving its open clients. If a client terminates without committing one or more of its transactions, those transactions are automatically aborted. The transaction commit coordinator is the Windows system where the MQC is installed.

Envoy MQ Client API function	Similar to MSMQ	Description
`FMQCommit()`	`ITransaction::Commit()`	Commit a transaction
`FMQAbort()`	`ITransaction::Abort()`	Abort a transaction

Committing a transaction — `FMQCommit()`

This function commits and releases an MSMQ transaction.

Prototype

```
HRESULT FMQCommit (Itransaction** ppTransaction, BOOL
fRetaining, DWORD grfTC, DWORD grfRM);
```

Arguments

`ppTransaction` (Input/output) The transaction to commit. On output, `ppTransaction` is set to `NULL` and Envoy MQ frees `*ppTransaction`.

The other arguments are the same as in the method `ITransaction::Commit()` of an MSMQ transaction object.

Return value

`MQ_OK` for success.

Aborting a transaction — `FMQAbort()`

This function aborts and releases an MSMQ transaction.

Prototype

```
HRESULT FMQAbort (Itransaction** ppTransaction, BOID*
pboidReason, BOOL fRetaining, BOOL fAsync);
```

Arguments

`ppTransaction` (Input/output) The transaction to abort. On output, `ppTransaction` is set to `NULL` and Envoy MQ frees `*ppTransaction`.

The other arguments are the same as in the method `ITransaction::Abort()` of an MSMQ transaction object.

Return value

`MQ_OK` for success.

API functions for information and debugging

Envoy MQ Client provides the following API functions for outputting debugging and version information.

Envoy MQ Client API function	Description
`FMQGetLogPath`	Retrieve the location of the debug log
`FMQSetLogPath()`	Set the location of the debug log
`FMQDebug()`	Enable or disable debug logging
`FMQVersion()`	Retrieve Envoy MQ Client version information

Setting the location of the debug log

`FMQSetLogPath()`
`FMQGetLogPath()`

The `FMQSetLogPath()` function sets the directory path of the Envoy MQ Client debug log for your application (see *Debug logging* above). `FMQGetLogPath()` retrieves the current path.

You can also define the log path by setting the `FMQLOGPATH` environment variable (see *Environment variables* in Chapter 3, *Installation*). Calling `FMQSetLogPath()` overrides the `FMQLOGPATH` setting. If you do not define the log path by either of these methods, Envoy MQ Client places the log in the default location (on most platforms, this is the `log/user` subdirectory of the Envoy MQ Client directory).

The name of the log file is `dc_xxxx.log`, where `xxxx` is the process id of the application that generated the log, for example `dc_1234.log`.

To use the log file, you must enable debug logging by calling the `FMQDebug` function or by setting the `FMQDEBUG` environment variable.

The prototypes of `FMQSetLogPath()` and `FMQGetLogPath()` are declared in `fmqpubd.h`. The function has no equivalent in MSMQ.

Prototypes

```
HRESULT APIENTRY FMQSetLogPath(PCSTR pszLogPath);

HRESULT APIENTRY FMQGetLogPath(PSTR pszLogPath, DWORD
dwCount);
```

Arguments

`pszLogPath`	The directory path of the log (not including a filename). If you specify `"CONSOLE"`, Envoy MQ Client sends the log to `stdout`.
`dwCount`	The size of the `pszLogPath` buffer.

Return value

`MQ_OK` for success.

Enabling debug logging FMQDebug()

The FMQDebug() function enables or disables debug logging (see *Debug logging* above).

By default, debug logging is disabled. If you enable debug logging, Envoy MQ Client places a copy of error and information messages in the debug log file.

You can also enable debug logging by setting the FMQDEBUG environment variable (see *Environment variables* in Chapter 3, *Installation*). Calling FMQDebug() overrides the FMQDEBUG setting.

The prototype of FMQDebug() is declared in fmqpubd.h. The function has no equivalent in MSMQ.

Prototype

```
HRESULT APIENTRY FMQDebug(FMQ_DEBUG_OP Debug);
```

Argument

Debug	FMQ_DEBUG_ON to enable debug logging, FMQ_DEBUG_OFF to disable.

Return value

MQ_OK for success.

Retrieving the Envoy MQ version FMQVersion()

The FMQVersion() function returns the version of Envoy MQ Client.

The function outputs an FMQVERSION structure, which contains the version information.

The prototype of FMQVersion() and the FMQVERSION structure are declared in fmqpubd.h. The function has no equivalent in MSMQ.

✓ *You can also run the fmqver utility to display version information. The utility is supplied in the Envoy MQ Client bin subdirectory.*

Prototype

```
HRESULT APIENTRY FMQVersion(FMQVERSION *FmqVersion);
```

Argument

FmqVersion	(In/out) On input, supply a pointer to an FMQVERSION structure. On output, FmqVersion->StringSummary contains the Envoy MQ Client version. FmqVersion->OperatingSystem contains the name of the operating system. (FVersion also has several other fields that store version information. See the declaration of FMQVERSION in fmqpubd.h for details).

Return value

MQ_OK for success.

Chapter 5

Sample Application

This chapter presents the complete source code of the `gwping` program. The program calls Envoy MQ Client API functions for messaging operations such as:

- Creating and opening a queue
- Sending a message
- Defining a response queue for a reply to the message
- Receiving a message

The program is supplied with Envoy MQ Client to test the Envoy MQ operation. For operating instructions, see the *Installation test* in Chapter 3, *Installation*.

The explanations in this chapter are necessarily brief. For complete information on the data structures and the calling syntax of API functions, see Chapter 4, *Programming Messaging Applications*.

Online sample programs

C-language source code and executable versions of the following sample applications are supplied with the Envoy MQ Client software:

`gwping`	Ping-pong test of Envoy MQ operation (sends *ping* test messages and receives the *pong* replies, see *Ping-pong messaging programs* below).
`gwpong`	Companion program to `gwping` (generates the *pong* replies).

Ping-pong messaging programs

`gwping`
`gwpong`

The `gwping` and `gwpong` programs are designed to test the operation of a Envoy MQ installation. You can run the programs on any system where the Envoy MQ Client is installed. For operating instructions, see *Installation test* in Chapter 3, *Installation*.

The programs conduct a *ping-pong* test of the Envoy MQ messaging system:

- `gwping` writes a *ping* message (by default containing the text `"PING"`) to a queue, along with the name of response queue where a reply should be sent
- `gwpong` (running in the background) reads the *ping* message and the name of the response queue
- `gwpong` writes a *pong* response message (containing the same text as the *ping*) to the response queue
- `gwping` reads the *pong* message from the response queue and compares it to the *ping*

✓ *The source code of `gwping` is presented below. You can examine the source code of `gwpong` online in your Envoy MQ directory.*

Source code — `gwping`

Headers and definitions

The Envoy MQ header files `wintypes.h` and `mq.h` are included (see *Header files* in Chapter 4, *Programming Messaging Applications*).

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "wintypes.h"
#include "mq.h"

#define ARRAY_SIZE(array) (sizeof(array)/sizeof(*(array)))
```

Main function

```
int main (unsigned int argc, char *argv[]) {

  unsigned int i;
  WCHAR * pszPingQName = ".\\PingQ";
  WCHAR * pszPongQName = ".\\PongQ";
  unsigned int bVerbose = 1, cbSend = 0, numTimes = 1,
    dwLen = 0, bRandSize = 0, bRandTime = 0;
  char * pszMessage = "PING";
  char * pszSend = 0;
  char * pszRecv = 0;
  char * pszRand = 0;
```

Message and queue property structures

Structures are declared and initialized for the destination and response queues and for the *ping* and *pong* messages.

For an explanation of the message property structure, see *Set up ping message* below.

```
  QUEUEPROPID aPIQueue[4];
  MQPROPVARIANT aPVQueue[4];
  HRESULT aHRQueue[4];
  MQQUEUEPROPS mqQProps = {0,NULL,NULL,NULL};
  WCHAR wszPongFName[256], wszPingFName[256];
  DWORD cbPongFName = ARRAY_SIZE(wszPongFName),
    cbPingFName = ARRAY_SIZE(wszPingFName);
  HRESULT hRes;
  QUEUEHANDLE hPingQueue, hPongQueue;
  ULONG dwCount = (ULONG)(-1);
  clock_t startTime;
  DWORD dwTotalTime;

  MSGPROPID aPIMsgPing[] =
    {PROPID_M_BODY, PROPID_M_APPSPECIFIC,
    PROPID_M_RESP_QUEUE, PROPID_M_BODY_TYPE};

  MQPROPVARIANT aPVMsgPing[ARRAY_SIZE(aPIMsgPing)] =
    {{VT_VECTOR | VT_UI1}, {VT_UI4}, {VT_LPWSTR}, {VT_UI4}};
  HRESULT aHRMsgPing[ARRAY_SIZE(aPIMsgPing)] = {0};
  MQMSGPROPS mqMPropsPing =
    {ARRAY_SIZE(aPIMsgPing),NULL,NULL,NULL};

  MSGPROPID aPIMsgPong[] =
{PROPID_M_BODY,PROPID_M_APPSPECIFIC};
  MQPROPVARIANT aPVMsgPong[ARRAY_SIZE(aPIMsgPong)] =
    {{VT_VECTOR | VT_UI1},{VT_UI4}};
  HRESULT aHRMsgPong[ARRAY_SIZE(aPIMsgPong)] = {0};
  MQMSGPROPS mqMPropsPong =
    {ARRAY_SIZE(aPIMsgPong),NULL,NULL,NULL};

  mqQProps.aPropID = aPIQueue;
  mqQProps.aPropVar = aPVQueue;
  mqQProps.aStatus = aHRQueue;

  mqMPropsPing.aPropID = aPIMsgPing;
  mqMPropsPing.aPropVar = aPVMsgPing;
  mqMPropsPing.aStatus = aHRMsgPing;

  mqMPropsPong.aPropID = aPIMsgPong;
  mqMPropsPong.aPropVar = aPVMsgPong;
  mqMPropsPong.aStatus = aHRMsgPong;
```

Interpret user input

For an explanation of the command-line syntax, see *Installation test* in Chapter 3, *Installation*.

```
  for (i=1; i<argc; i++) {
    if (strcmp(argv[i], "-q") == 0) {
      pszPongQName = argv[++i];
      continue;
    }
```

```
  if (strcmp(argv[i], "-p") == 0) {
    pszPingQName = argv[++i];
    continue;
  }

  if (strcmp(argv[i], "-n") == 0) {
    numTimes = atoi(argv[++i]);
    continue;
  }

  if (strcmp(argv[i], "-s") == 0) {
    pszMessage = argv[++i];
    continue;
  }

  if (strcmp(argv[i], "-l") == 0) {
    cbSend = atoi(argv[++i]);
    continue;
  }

  if (strcmp(argv[i], "-v") == 0) {
    bVerbose = 0;
    continue;
  }
  if (strcmp(argv[i], "-r") == 0) {
    bRandSize = 1;
    continue;
  }

  printf("Usage: \n"
    "-p: Ping queue\n"
    "-q: Pong queue\n"
    "-n: Number of iterations\n"
    "-s: String to send\n"
    "-l: Message length\n"
    "-r: Random length up to message length\n"
    "-v: Short output\n");
  return 0;
}

if (!cbSend) cbSend = strlen(pszMessage) + 1;
```

Allocate buffers

The program allocates buffers to send the *ping* message and to receive the *pong* reply. Both buffers are allocated with twice the size that is actually needed for the message string.

The double-sized buffer is required for receiving because `MQReceiveMessage()` receives the UNICODE (2-byte character) representation of the string, which it then converts to the local code page (see *Code page translation* in Chapter 4, *Programming Messaging Applications*).

The double-sized buffer is not required for sending. A single-sized buffer would suffice, but the double size is allocated for coding consistency.

```
pszSend = (char *) malloc(cbSend * 2);
```

```
  pszRecv = (char *) malloc(cbSend * 2);
  pszRand = (char *) malloc(cbSend * 2);
  if (!pszSend || !pszRecv || !pszRand) {
    printf("Failed to allocate memory buffers for
messages\n");
    exit(1);
  }

  dwLen = strlen(pszMessage);
  for (i=0; i+dwLen < cbSend; i+=dwLen) memmove(pszSend+i,
    pszMessage,dwLen);
  memmove(pszSend+i, pszMessage,(cbSend-1) % dwLen);
  pszSend[cbSend-1] = 0;
```

Create destination queue

The program calls the Envoy MQ API function `MQCreateQueue()` to create a destination queue for the *ping* if it doesn't already exist. By default, the queue is called `.\PongQ` because it is local to the `gwpong` program.

The `gwpong` program independently attempts to create the same queue. When `gwping` and `gwpong` run concurrently, at least one of them receives `MQ_ERROR_QUEUE_EXISTS` indicating that the queue already exists.

The program assigns one property to the queue: the queue path name. The code uses the `MQVAL` macro to assign the property value. `MQVAL` is used to make the code portable between Envoy MQ and MSMQ (see *Notation of property value fields* in Chapter 4*, Programming Messaging Applications*).

```
  aPIQueue[0] = PROPID_Q_PATHNAME;
  aPVQueue[0].vt = VT_LPWSTR;
  aPVQueue[0].MQVAL(pwszVal) = pszPongQName;
  mqQProps.cProp = 1;
  if (bVerbose) printf ("Attempting to create Pong queue
%s\n",
    pszPongQName);
  hRes = MQCreateQueue(PSD_SPECIALACCESS_ALL, &mqQProps,
    wszPongFName, &cbPongFName);
  if (hRes == MQ_ERROR_QUEUE_EXISTS) {
    if (bVerbose) printf("Pong Queue %s Exists: %#8.8X\n",
      pszPongQName, hRes);
  } else if (hRes != MQ_OK) {
    printf("Attempt to create pong queue %s failed:
%#8.8X\n",
      pszPongQName, hRes);
    exit(1);
  }
```

Determine format name of destination queue

`MQPathNameToFormatName()` determines the format name, which is needed to open the queue. (The format name is returned by `MQCreateQueue()` only if a new queue is created.)

```
  cbPongFName = ARRAY_SIZE(wszPongFName);
  hRes = MQPathNameToFormatName(pszPongQName, wszPongFName,
    &cbPongFName);
  if (hRes == MQ_OK){
```

```
    if (bVerbose) printf("Pong queue %s Format name =
%s\n\n",
      pszPongQName, wszPongFName);
  } else {
    printf("MQPathNameToFormatName %s failed: %#8.8X\n",
      pszPongQName, hRes);
    exit(1);
  }
```

Open destination queue

The program calls `MQOpenQueue()` to open the destination queue for sending (`gwpong` opens it for receiving).

```
  if (bVerbose) printf("Attempting to open Pong queue for
send\n");
  hRes = MQOpenQueue(wszPongFName, MQ_SEND_ACCESS, 0,
&hPongQueue);
  if (hRes != MQ_OK) {
    printf( "Attempt to open pong queue %s failed: %#8.8X\n",
      pszPongQName, hRes);
    exit(1);
  } else {
    if (bVerbose) printf("Pong Queue %s Opened\n",
pszPongQName);
  }
```

Create response queue

The program creates a second queue for the *pong* response (by default called `.\PingQ` because it is local to the `gwping` program).

```
  aPIQueue[0] = PROPID_Q_PATHNAME;
  aPVQueue[0].vt = VT_LPWSTR;
  aPVQueue[0].MQVAL(pwszVal) = pszPingQName;
  mqQProps.cProp = 1;
  if (bVerbose)  printf ("Attempting to create ping queue
%s\n",
    pszPingQName);
  hRes = MQCreateQueue(PSD_SPECIALACCESS_ALL, &mqQProps,
    wszPingFName, &cbPingFName);
  if (hRes == MQ_ERROR_QUEUE_EXISTS) {
    if (bVerbose) printf("Ping Queue %s Exists: %#8.8X\n",
      pszPingQName, hRes);
  } else if (hRes != MQ_OK) {
    printf("Attempt to create ping queue %s failed:
%#8.8X\n",
      pszPingQName, hRes);
    exit(1);
  }
```

Determine format name of response queue

```
  cbPingFName = ARRAY_SIZE(wszPingFName);
  hRes =

MQPathNameToFormatName(pszPingQName,wszPingFName,&cbPingFName
);
  if (hRes == MQ_OK){
```

```
    if (bVerbose) printf("Ping queue %s Format name =
%s\n\n",
      pszPingQName, wszPingFName);
  } else {
    printf("MQPathNameToFormatName %s failed: %#8.8X\n",
      pszPingQName, hRes);
    exit(1);
  }
```

Open response queue

The response queue is opened for receiving (`gwpong` opens it for sending).

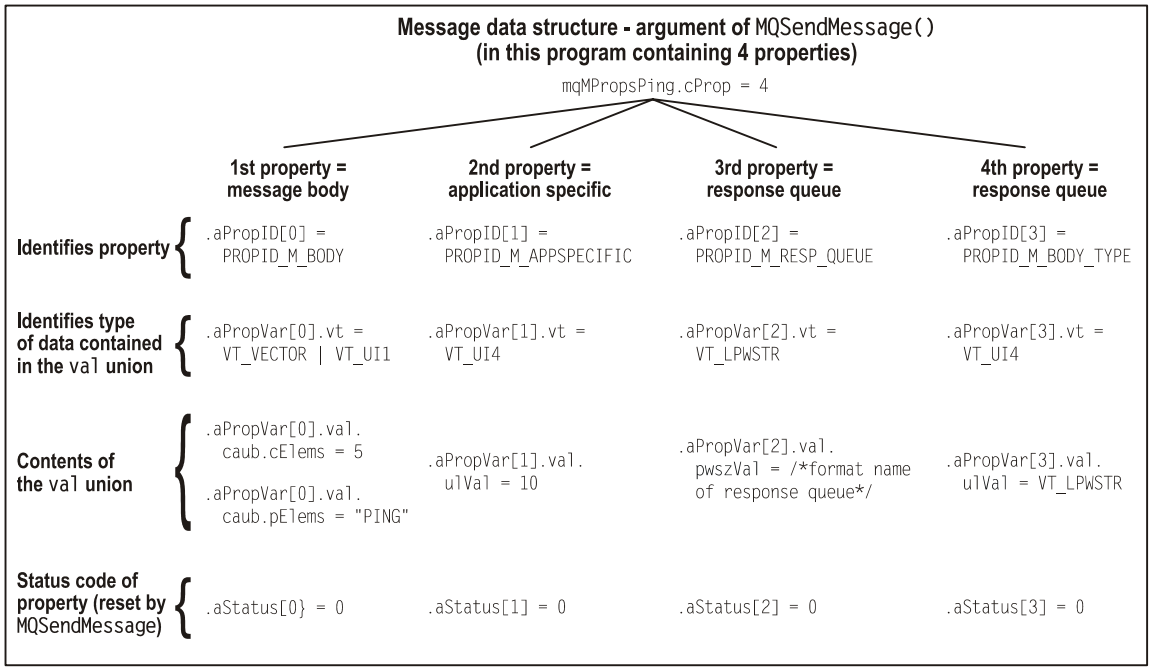
```
  hRes = MQOpenQueue(wszPingFName, MQ_RECEIVE_ACCESS,
    MQ_DENY_RECEIVE_SHARE, &hPingQueue);
  if (hRes != MQ_OK) {
    printf( "Attempt to open ping queue %s failed: %#8.8X\n",
      pszPongQName, hRes);
    exit(1);
  } else {
    if (bVerbose) printf( "Ping Queue %s Opened\n",
pszPongQName);
  }
```

Set up ping message

The message data structure includes four properties:

- The *message body*—containing the message string (e.g., `"PING"`)
- *An application-specific integer*—containing the sequence number of the current ping-pong iteration (e.g., 10)
- *The format name of the response queue*—obtained from `MQPathNameToFormatName()` above
- The *message body type*—which has a value of `VT_LPWSTR`, indicating that the message body contains a UNICODE string. This body type causes Envoy MQ Client to translate the message body from the local code page to UNICODE (see *Code page translation* in Chapter 4, *Programming Messaging Applications*).

The `cProp`, `aPropID` and `vt` fields were previously initialized (see *Message and queue property structures* above).



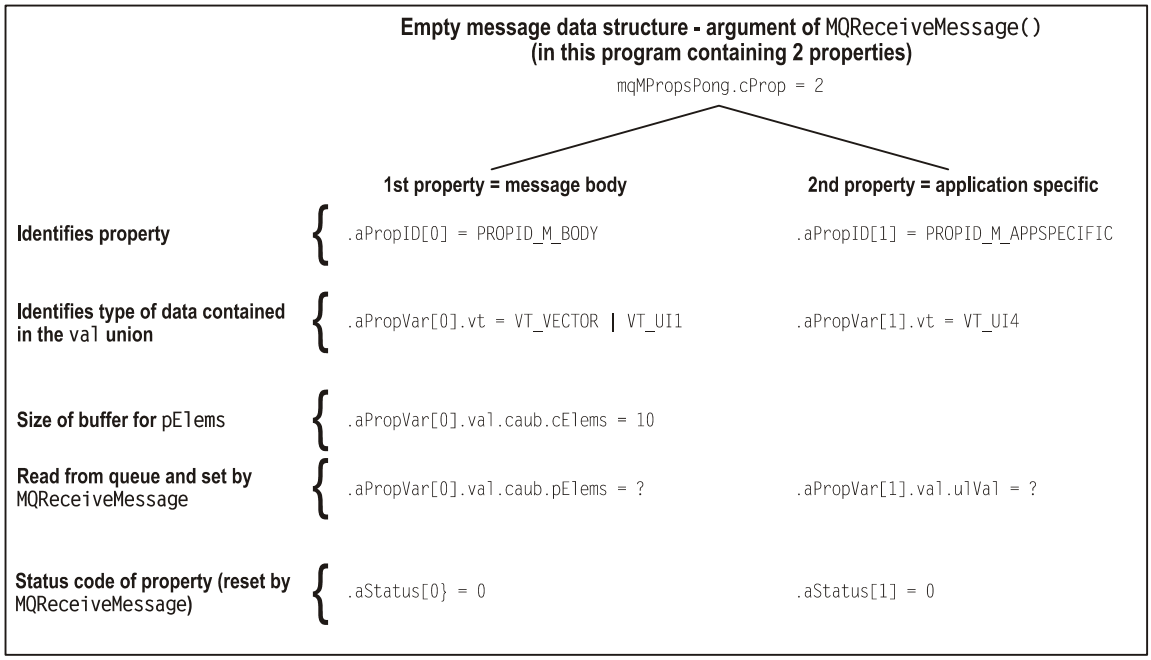

```
  aPVMsgPing[0].MQVAL(caub.cElems) = cbSend;
  aPVMsgPing[0].MQVAL(caub.pElems) = (unsigned char
*)pszSend;
  aPVMsgPing[2].MQVAL(pwszVal) = wszPingFName;
  aPVMsgPing[3].MQVAL(ulVal) = VT_LPWSTR;
```

Set up empty structure for pong message

The *pong* message is received into an empty message structure containing two message properties:

- The message body
- The application specific property (message sequence number)

For the declarations of the message structure, see *Message and queue property structures* above. The program sets the size indicator of the receive buffer (`cElems` field) to twice the actual string length of the message because `MQReceiveMessage()` needs to receive the string in a UNICODE representation (see *Allocate buffers* above).




```
  aPVMsgPong[0].MQVAL(caub.pElems) = (unsigned char
*)pszRecv;
  aPVMsgPong[0].MQVAL(caub.cElems) = cbSend * 2;
```

Send ping message

The program initializes a timer and sends the sequence of *ping* messages. The arguments of `MQSendMessage()` are:

Argument	Meaning
`hPongQueue`	Handle of the destination queue, obtained from `MQOpenQueue()` above
`&mqMPropsPing`	Pointer to the message data structure
`NULL`	Pointer to a transaction object

```
  startTime = time(NULL);

  for (i=1; i<=numTimes; i++) {
    aPVMsgPing[1].MQVAL(ulVal) = i;
    if (bRandSize) {
      int len =(ULONG)(((1.0 * rand()) / RAND_MAX) * cbSend +
1);
      aPVMsgPing[0].MQVAL(caub.cElems) = len;
      strcpy(pszRand,pszSend);
      pszRand[len-1]='\0';
      aPVMsgPing[0].MQVAL(caub.pElems) = (unsigned char
*)pszRand;
    }

   hRes = MQSendMessage(hPongQueue, &mqMPropsPing, NULL);
```

```
    if (hRes != MQ_OK) {
      printf("Failed to send ping to queue %s, status =
%#8.8X\n",
        pszPongQName, hRes);
    } else {
      if (bVerbose) printf(
        "Ping #: %d \"%.20s\" sent to queue %s\n", i,
        aPVMsgPing[0].MQVAL(caub.pElems),
        pszPongQName);
    }

    if (hRes == MQ_ERROR ) break;

    if (bRandSize) {
      aPVMsgPong[0].MQVAL(caub.cElems) =
        aPVMsgPing[0].MQVAL(caub.cElems) * 2;
    }
```

Receive pong response

The arguments of `MQReceiveMessage()` are:

Argument	Meaning
`hPingQueue`	Handle of the response queue, obtained from `MQOpenQueue()` above
`INFINITE`	Time to wait for the response (could alternatively be set to a finite time, e.g., 10,000 msec; if no response were received within the specified time, `MQReceiveMessage()` would return an error)
`MQ_ACTION_RECEIVE`	The desired action (`RECEIVE` and delete the message from the queue, as opposed to `PEEK` which reads but does not delete)
`&mqMPropsPong`	Pointer to the empty message data structure
`NULL` `NULL`	MSMQ callback function (not supported by Envoy MQ)
`0`	Cursor handle (not used by `gwping`)
`NULL`	Pointer to a transaction object

✓ *For an example of receiving a message together with the name of a response queue, examine the `gwpong` source code provided online with Envoy MQ.*

```
    hRes = MQReceiveMessage(hPingQueue, INFINITE,
MQ_ACTION_RECEIVE,
      &mqMPropsPong, NULL, NULL, 0, NULL);
```

Compare contents of ping and pong messages

The program compares the sequence number, message size, and message contents of the *ping* message and the *pong* reply.

The message buffer size (`cElems` field) of the pong message is twice the actual length of the received string because `MQReceiveMessage()` initially received the message in a UNICODE representation. The program therefore divides `cElems` by 2 to determine the actual length of the *pong* string.

```
    if (hRes == MQ_OK) {
      if (aPVMsgPong[1].MQVAL(ulVal) ==
aPVMsgPing[1].MQVAL(ulVal))
      {
        if (bVerbose)
          printf("Received reply %d: \"%.20s\"\n",
          aPVMsgPong[1].MQVAL(ulVal),
          aPVMsgPong[0].MQVAL(caub.pElems));
        if ((aPVMsgPing[0].MQVAL(caub.cElems) !=
          aPVMsgPong[0].MQVAL(caub.cElems))/2 ||
          memcmp(aPVMsgPong[0].MQVAL(caub.pElems),
          aPVMsgPing[0].MQVAL(caub.pElems),
          aPVMsgPing[0].MQVAL(caub.cElems))) {
          printf("Reply buffer different from sent
buffer\n");
        }
      } else {
        printf("Wrong reply sequence %d expected %d\n",

aPVMsgPong[1].MQVAL(ulVal),aPVMsgPing[1].MQVAL(ulVal));
      }
    } else {
      printf("Failed to recv reply for ping %d, status =
%#8.8X\n",
        i, hRes);
    }
  }
```

Report elapsed time

The program reports the time for sending the sequence of *ping* messages and receiving the *pong* replies.

```
  dwTotalTime = (time(NULL) - startTime);

  printf("Total time %lu seconds\n",dwTotalTime);
```

Close the queues

```
  hRes = MQCloseQueue(hPingQueue);
  hRes = MQCloseQueue(hPongQueue);
```

Delete the response queue

The program deletes the response queue that it created. It does not delete the destination queue, because that queue is considered to belong to the `gwpong` program.

```
  hRes = MQDeleteQueue(wszPingFName);

  if (bVerbose) {
    printf("press the <Enter> Key...\n");
    getchar();
```

```
    }
    if (pszSend) free(pszSend);
    if (pszRecv) free(pszRecv);
    if (pszRand) free(pszRand);
    return 0;
}
```

Appendix A

Glossary

Abort	Cancel and roll back all operations in a transaction.
API	*A*pplication *p*rogramming *i*nterface.
Application	See *Envoy MQ application*.
Asynchronous	Not needing to wait until operation is completed.
Authentication	A digital signature method confirming that a received message comes from the purported sender and has not been altered.
Certificate	A unique user identifier for message authentication.
Code page	A mapping of characters and symbols (usually of a specific language) to single-byte numerical values.
Commit	Confirm and irrevocably execute all operations in a transaction.
Connectionless messaging	A method for sending messages between applications that do not need to be connected in a communication session.
Default logon	Access to MQC controlled by the user name under which an application runs on an external system.
Destination queue	The queue to which a message is sent.
Encryption	A digital encoding that prevents unauthorized persons from reading a message.
Envoy MQ	Envoy Technologies messaging interface software products.
Envoy MQ application	A program that calls the Envoy MQ API.
Envoy MQ Client	Envoy MQ component running on non-Windows operating systems and providing a programming interface for messaging applications.

Envoy MQ Connector	Envoy MQ component running under the Windows operating systems and interfacing MSMQ with Envoy MQ Clients.
Explicit logon	Access to Envoy MQ Connector controlled by a user name and password, sent by an application.
Extension field	A field of the message extension property containing a GUID plus any data.
External certificate	A certificate obtained from a certificate authority, used to verify who sent a message.
Format name	A code assigned by MSMQ to uniquely identify a queue throughout a network.
GUID	*G*lobally *U*nique *Id*entifier code identifying software objects.
Internal certificate	A public key written in the form of an X.509 certificate, used to verify that the sender identifier attached to a message is valid.
Local queue	A message queue stored on the same computer as an application or messaging software installation.
LU	*L*ogical *u*nit.
Message	A set of data transmitted from an application to another application on the same or a different computer.
Message body	The message property whose value is the main message content (text or binary).
Message extension	A structured message property containing any number of extension fields.
Message property	A data field of a message that is predefined in the messaging software.
Message queue	A location where messages are stored, which can be written and read by applications.
MSMQ	*M*icrosoft *M*essage *Q*ueue Server.
Path name	A queue name including a network path, e.g., `.\queue1.mq` or `machine2\queue1.mq`.
Property structure	A data structure representing a set of message, queue, or queue manager properties.
PROPVARIANT	A data structure containing the value of a message, queue, or queue manager property.
Query structure	A data structure used to represent a queue query.
Queue name	See *Format name* and *Path name*.
Read	Retrieve a message from a queue; receive.

Receiving application	A program that calls the messaging software API to receive a message.
Remote queue	A message queue stored on a different computer from an application or messaging software installation.
Response	A message sent in reply to another message.
Response queue	A queue specified by a sending application, to which a receiving application should send a reply.
Roll back	Undo all operations in a transaction.
RPC	*R*emote *p*rocedure *c*all. A function call on one computer, acting on data in that computer, but executed on another computer.
SDK	*S*oftware *D*evelopment *K*it.
Sending application	A program that calls the messaging software API to send a message.
SNA	*S*ystem *N*etwork *A*rchitecture. A set of IBM protocols for network communication.
Store and forward	Save a message at a sequence of one or more intermediate locations until it can be sent to its final destination.
Synchronous	Needing to wait until operation is completed.
TCP/IP	A standard protocol for network communication.
Trace file	A file recording calls to the Envoy MQ API and a hexadecimal dump of transmissions.
Transaction	A set of operations packaged together and committed or aborted as a group.
UNICODE	An international standard that maps the characters and symbols of many languages to a unique set of double-byte numerical values.
Write	Place a message on a queue; send.

Envoy MQ Programmer's Guide

Index