[MS-MQRR]:
Message Queuing (MSMQ):
Queue Manager Remote Read Protocol Specification

Intellectual Property Rights Notice for Protocol Documentation

- This protocol documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the protocols, and may distribute portions of it in your implementations of the protocols or your documentation as necessary to properly document the implementation. This permission also applies to any documents that are referenced in the protocol documentation.
- Microsoft does not claim any trade secret rights in this documentation.
- Microsoft has patents that may cover your implementations of the protocols. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. If you are interested in obtaining a patent license, please contact protocol@microsoft.com.
- The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

This protocol documentation is intended for use in conjunction with publicly available standard specifications, network programming art, and Microsoft Windows distributed systems concepts, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

A protocol specification does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them.

Revision Summary

Date	Revision History	Revision Class	Comments
02/22/2007	0.01		MCPP Milestone 3 Initial Availability
06/01/2007	1.0	Major	Updated and revised the technical content.
07/03/2007	1.0.1	Editorial	Revised and edited the technical content.
07/20/2007	1.0.2	Editorial	Revised and edited the technical content.

Copyright © 2008 Microsoft Corporation.

Date	Revision History	Revision Class	Comments
08/10/2007	2.0	Major	Updated and revised the technical content.
09/28/2007	2.0.1	Editorial	Revised and edited the technical content.
10/23/2007	2.0.2	Editorial	Revised and edited the technical content.
11/30/2007	2.0.3	Editorial	Revised and edited the technical content.
01/25/2008	2.0.4	Editorial	Revised and edited the technical content.

Copyright © 2008 Microsoft Corporation.

Table of Contents

Copyright © 2008 Microsoft Corporation.

Copyright © 2008 Microsoft Corporation.

Copyright © 2008 Microsoft Corporation.

1 Introduction

This document specifies the Message Queuing (MSMQ): Queue Manager Remote Read Protocol, an **RPC**-based protocol that is used by **Message Queuing (MSMQ)** clients to read or reject a **message** from a **queue**, move a message between queues, and purge all messages from a queue.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Authentication Level
Dynamic Endpoint
Endpoint
Globally Unique Identifier (GUID)
Interface Definition Language (IDL)
Network Data Representation (NDR)
Opnum
Remote Procedure Call (RPC)
RPC Protocol Sequence
RPC Transport
Universally Unique Identifier (UUID)

The following terms are defined in [MS-MQMA]:

Transaction

The following terms are defined in [MS-MQMQ]:

Connector Queue
Cursor
Dead-Letter Queue
Distribution List (DL)
Message
Message Body
Message Packet
Message Packet Header
Message Packet Trailer
Message Property
Microsoft Message Queuing (MSMQ)
Private Queue
Public Queue
Queue
Queue Journal
Queue Manager
Subqueue

The following terms are specific to this document:

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

Copyright © 2008 Microsoft Corporation.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[MC-MQSRM] Microsoft Corporation, "Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP)", October 2007.

[MS-DTCO] Microsoft Corporation, "MSDTC Connection Manager: OleTx Transaction Protocol Specification", July 2007.

[MS-DTYP] Microsoft Corporation, "Windows Data Types", January 2007.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-MQBR] Microsoft Corporation, "Message Queuing (MSMQ): Binary Reliable Messaging Algorithm", September 2007.

[MS-MQMA] Microsoft Corporation, "Message Queuing (MSMQ): Architecture Protocol Specification", August 2007.

[MS-MQMQ] Microsoft Corporation, "Message Queuing (MSMQ): Data Structures", August 2007.

[MS-MQQB] Microsoft Corporation, "Message Queuing (MSMQ): Message Queuing Binary Protocol Specification", August 2007.

[MS-MQQP] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol Specification", August 2007.

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", January 2007.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC2553] Gilligan, R., Thomson, S., Bound, J., and Stevens, W., "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999, http://www.ietf.org/rfc/rfc2553.txt

1.2.2 Informative References

There are no informative references for this protocol.

1.3 Protocol Overview (Synopsis)

Message queuing is a communications service that provides asynchronous and reliable message passing between client applications running on different hosts. In message queuing, clients send application messages to a queue and/or consume application messages from a queue. The queue provides persistence of the messages, enabling them to survive across application restarts, and

Copyright © 2008 Microsoft Corporation.

allowing the sending and receiving client applications to send and receive messages asynchronously from each other.

Queues are typically hosted by a communications service called a **queue manager**. By hosting the queue manager in a separate service from the client applications, applications can communicate even if they never execute at the same time by exchanging messages via a queue hosted by the queue manager.

The queue manager may execute on a different node than the client applications. When this scenario occurs, a protocol is required to insert messages into the queue, and another protocol is needed to consume messages from the queue. The Message Queuing (MSMQ): Queue Manager Remote Read Protocol provides a protocol for consuming messages from a queue.

1.3.1 Messages

Each message exchanged in a message queuing system typically has a set of **message properties** that contain metadata about the message and a distinguished property, called a **message body**, that contains the application payload. Message properties that are serialized in front of the message body are referred to as message headers, and message properties serialized after the message body property are referred to as message trailers.

Messages carried by this protocol are treated as payload. The format and structure of the application messages is generally opaque to the protocol. However, the protocol does assume that such messages map to the abstractions of message header, message body, and message trailer mentioned above. This mapping enables a consumer to request that a subset of the message body be returned while allowing all the message headers and message trailers to be returned. For more details, see the **SectionBuffer (section 2.2.6)** structure.

The protocol also assumes that each message has a lookup identifier that is unique within the queue. This identifier is not part of the message but is instead assigned by the server.

1.3.2 Queues

A queue is a logical data structure that contains an ordered list of zero or more messages. Queues, like files, have names. This protocol uses the QUEUE_FORMAT (section 2.2.3) structure to identify queues.

This protocol provides a mechanism to open a queue. Opening provides an opportunity to check for the existence of the queue and to perform authorization checks. The protocol provides for the return of an RPC context handle that is used by the client to specify the queue to operate on in subsequent requests. The use of an RPC context handle provides a mechanism to ensure that server state is cleaned up if the connection between the client and server is lost.

When opening a queue, the client can specify an access mode that determines the operations (Peek, Receive, Move, Reject, and Purge) for which the returned handle can subsequently be used. The client can specify a sharing mode that either allows other clients to concurrently access the queue, or ensures that the client has exclusive access to the queue. The latter can be used to avoid race conditions caused by other clients operating on the queue at the same time.

1.3.3 Queue Operations

The protocol provides mechanisms for the following operations against an open queue.

A message can be consumed from an open queue through a destructive read operation referred to as a **Receive** operation. This operation atomically reads the message and removes it from the queue. Because this operation removes a message from a queue, a loss of network connection

Copyright © 2008 Microsoft Corporation.

during this operation could result in permanent loss of the message. To guard against this situation, the protocol provides a mechanism for the client to positively or negatively acknowledge receipt of the message. Upon receipt of positive acknowledgment from the client, the server can remove the message from the queue. While the server is awaiting acknowledgment from the client, access to the message by other clients is prevented.

A message can be read from an open queue through a non-destructive read operation referred to as a **Peek** operation. This operation reads the message but does not remove it from the queue.

For both **Receive** and **Peek** operations, the client can limit the amount of the message body payload returned. This enables efficient use of network resources when the client requires only a portion of the message body, or when the client needs just the message properties.

All the messages can be removed from a queue through a **Purge** mechanism. The messages removed through this mechanism are not returned to the client.

A message can be moved from one queue to another queue hosted at the same server through an atomic **Move** mechanism.

A client can inform the server that it has no need for a message via a **Reject** operation. The server can use this indication to inform the sender that the client did not consume the message. How a server does this is not addressed in this specification.

1.3.4 Access Patterns

Messages in a queue can be consumed in a first-in-first-out (FIFO) access pattern. Because messages in a queue are ordered, there is a head that represents the front of the queue and a tail that represents the end of the queue.

The protocol provides mechanisms to **Peek** or **Receive** the first message or the last message in the queue.

The protocol also allows the client to specify exactly which message to **Peek** or **Receive**, regardless of its position in the queue, through a unique lookup identifier assigned to each message by the server. A message can also be specified relative to the message identified by the lookup identifier, that is, the message immediately preceding or following the message identified by the lookup identifier.

Finally, the protocol provides a mechanism, referred to as a **cursor**, for sequential forward access through the queue. A cursor logically represents a current pointer that lies between the head and tail of the queue. A cursor can be specified to the **Peek** or **Receive** operation, which **Peeks** or **Receives** the message at the current pointer represented by the cursor. The cursor current pointer can be moved forward through a modified **Peek** operation called **PeekNext**. A **Receive** operation intrinsically moves the cursor forward.

Because cursors are stateful, the protocol provides mechanisms to create a cursor, to return a cursor handle to the client, and to close a cursor. Because a cursor represents a position within a queue, the protocol logically relates the cursor to the context handle associated with an open queue. The protocol places no limit on the number of concurrent cursor associated with a queue context handle.

1.3.5 Transactions

The protocol allows the queue operations **Receive** or **Move** to be performed within the context of a distributed atomic transaction, as specified in [MS-DTCO]. When this is done, the state changes that are related to the queue associated with the operation are performed provisionally, awaiting

Copyright © 2008 Microsoft Corporation.

asynchronous notification of the outcome of the transaction. If the transaction outcome is **Commit**, the state changes become permanent. If the transaction outcome is **Abort**, the state changes are rolled back.

The protocol does not require that all queues support this atomic transaction behavior. A queue that supports transactional **Receive** must also support non-transactional **Receive**. The protocol returns an error if a transacted operation is attempted against a non-transactional queue. The protocol does not provide any other mechanism for determining whether a queue supports transactional behavior.

1.4 Relationship to Other Protocols

The Message Queuing (MSMQ): Queue Manager Remote Read Protocol is dependent upon RPC for its **transport**. This protocol uses RPC as specified in section 2.1.

The protocol functionality is a superset of the functionality as specified in [MS-MQQP]. Implementers SHOULD choose this protocol over Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol except where compatibility necessitates using MSMQ: Queue Manager to Queue Manager Protocol, as specified in [MS-MQQP].<1>

To orchestrate the transactional scenarios of this protocol, this protocol carries Propagation Tokens, as specified in [MS-DTCO] section 2.

This protocol is capable of carrying the layout and internal structure of the message in the queue, as specified in [MS-MQQP].

1.5 Prerequisites/Preconditions

The Message Queuing (MSMQ): Queue Manager Remote Read Protocol is an RPC interface, and as a result, has prerequisites, as specified in [MS-RPCE], that are common to RPC interfaces.

It is assumed that the protocol client has obtained the name of a remote computer that supports this protocol before this protocol is invoked.

This protocol uses authentication through RPC. The client must be in possession of valid credentials recognized by the server. The server must be started and fully initialized before the protocol can start.

1.6 Applicability Statement

This protocol provides functionality related to consumption of messages from a queue hosted at a queue manager running on a remote computer. It does not provide functionality related to inserting messages into a queue.

The server side of the Message Queuing (MSMQ): Queue Manager Remote Read Protocol is applicable for implementation by a queue manager that provides message queuing communication services to clients. The client side of this protocol is applicable for implementation by client libraries that provide message queuing services to applications, or by a client queue manager that delegates requests on behalf of a client application.

This protocol could be used to reliably transfer messages from a queue hosted at one queue manager (the server) to a queue hosted at another queue manager (the client). However, there are other protocols that may be more suited to providing such reliable message transfer between queues. Message Queuing (MSMQ): Binary Reliable Messaging Protocol Specification, as specified in [MS-MQBR], is one such protocol that may provide the message transfer functionality more efficiently and in a manner that provides end-to-end reliability through intermediate store-and-forward hops.

Copyright © 2008 Microsoft Corporation.

1.7 Versioning and Capability Negotiation

Supported transports: This protocol uses the RPC over TCP/IP protocol sequence. However, it supports a mechanism for explicitly negotiating the RPC **endpoint** to be used. Details are specified in section 3.1.4.1.

Protocol versions: This protocol uses a single version of the RPC interface, but that interface has been extended by adding the following additional methods at the end.

- **R_MoveMessage (Opnum 10) (section 3.1.4.10)**
- **R_OpenQueueForMove (Opnum 11) (section 3.1.4.11)**
- **R_QMEnlistRemoteTransaction (Opnum 12) (section 3.1.4.12)**
- **R_StartTransactionalReceive (Opnum 13) (section 3.1.4.13)**
- **R_SetUserAcknowledgementClass (Opnum 14) (section 3.1.4.14)**
- **R_EndTransactionalReceive (Opnum 15) (section 3.1.4.15)**

Security and authentication methods: This protocol supports the NTLM and Kerberos authentication methods.<2>

1.8 Vendor-Extensible Fields

This protocol uses HRESULTs as specified in [MS-ERREF]. Vendors can choose their own values for this field, as long as the C bit (0x20000000) is set, indicating that it is a customer code.

1.9 Standards Assignments

Parameter	Value	Reference
RPC interface **UUID**	1A9134DD-7B39-45BA-AD88-44D01CA47F28	[C706]
Interface version	1.0	[C706]

Copyright © 2008 Microsoft Corporation.

2 Messages

The following sections specify how Message Queuing (MSMQ): Queue Manager Remote Read Protocol messages are transported and common MSMQ: Queue Manager Remote Read Protocol data types.

2.1 Transport

This protocol MUST use the following **RPC protocol sequence**: RPC over TCP/IP (ncacn_ip_tcp), as specified in [MS-RPCE]. This protocol SHOULD use RPC **dynamic endpoints** as specified in [C706] section 4. This protocol MAY use an RPC static endpoint as specified in **R_GetServerPort (section 3.1.4.1)**.<3>

This protocol allows any user to establish a connection to the RPC server. For each connection, the server uses the underlying RPC protocol to retrieve the identity of the invoking client, as specified in [MS-RPCE] section 3.3.3.4.3. The server SHOULD use this identity to perform method-specific access checks.

2.2 Common Data Types

This protocol MUST indicate to the RPC runtime that it is to support both the **NDR** and NDR64 transfer syntaxes and MUST provide a negotiation mechanism for determining which transfer syntax will be used, as specified in [MS-RPCE] section 3.

In addition to the RPC base types and definitions, as specified in [C706] and [MS-DTYP], additional data types are defined in the following list that summarizes the types defined in this specification.

- HRESULT
- GUID
- **QUEUE_FORMAT**
- **QueueContextHandle**
- Message Packet Structure
- **SectionBuffer**
- **SectionType**

2.2.1 HRESULT

This specification uses the HRESULT type as specified in [MS-ERREF].

2.2.2 GUID

This specification uses a **globally unique identifier** as specified in [MS-DTYP].

2.2.3 QUEUE_FORMAT

This structure is used to identify a queue. This structure is common to many message queuing (MSMQ) protocols. Not every **QUEUE_FORMAT_TYPE** enumeration and field of this structure are supported by this protocol. However, they are included here for completeness. Where an aspect of this structure is not supported by this protocol, it is identified as such.

Copyright © 2008 Microsoft Corporation.

2.2.3.1 QUEUE_FORMAT_TYPE

The **QUEUE_FORMAT_TYPE** enumeration identifies the type of name format being used.

```
typedef  enum
{
  QUEUE_FORMAT_TYPE_UNKNOWN = 0,
  QUEUE_FORMAT_TYPE_PUBLIC = 1,
  QUEUE_FORMAT_TYPE_PRIVATE = 2,
  QUEUE_FORMAT_TYPE_DIRECT = 3,
  QUEUE_FORMAT_TYPE_MACHINE = 4,
  QUEUE_FORMAT_TYPE_CONNECTOR = 5,
  QUEUE_FORMAT_TYPE_DL = 6,
  QUEUE_FORMAT_TYPE_MULTICAST = 7,
  QUEUE_FORMAT_TYPE_SUBQUEUE = 8
} QUEUE_FORMAT_TYPE;
```

QUEUE_FORMAT_TYPE_UNKNOWN: The format type is unknown.

QUEUE_FORMAT_TYPE_PUBLIC: The **QUEUE_FORMAT** structure contains a GUID identifying a queue.

QUEUE_FORMAT_TYPE_PRIVATE: The **QUEUE_FORMAT** structure contains an **OBJECTID (section 2.2.3.3)** identifying a queue.

QUEUE_FORMAT_TYPE_DIRECT: The **QUEUE_FORMAT** structure contains a direct name string identifying a queue.

QUEUE_FORMAT_TYPE_MACHINE: The **QUEUE_FORMAT** structure contains a GUID identifying a queue.

QUEUE_FORMAT_TYPE_CONNECTOR: The **QUEUE_FORMAT** structure contains a GUID identifying a **connector queue**. Not supported by this protocol.

QUEUE_FORMAT_TYPE_DL: The **QUEUE_FORMAT** structure contains a GUID identifying a **distribution list (DL)**. Not supported by this protocol.

QUEUE_FORMAT_TYPE_MULTICAST: The **QUEUE_FORMAT** structure contains a GUID identifying a multicast address. Not supported by this protocol.

QUEUE_FORMAT_TYPE_SUBQUEUE: The **QUEUE_FORMAT** structure contains a direct name string identifying a **subqueue**.<4>

2.2.3.2 QUEUE_SUFFIX_TYPE

This enumeration is used in the **m_SuffixAndFlags** field of the **QUEUE_FORMAT** structure to identify a subqueue of the queue identified by the unnamed union of the **QUEUE_FORMAT** structure, or to identify a system queue of the machine identified by the unnamed union of the **QUEUE_FORMAT** structure.

```
typedef  enum
{
  QUEUE_SUFFIX_TYPE_NONE = 0,
  QUEUE_SUFFIX_TYPE_JOURNAL = 1,
  QUEUE_SUFFIX_TYPE_DEADLETTER = 2,
  QUEUE_SUFFIX_TYPE_DEADXACT = 3,
```


Copyright © 2008 Microsoft Corporation.

```
  QUEUE_SUFFIX_TYPE_XACTONLY = 4,
  QUEUE_SUFFIX_TYPE_SUBQUEUE = 5
} QUEUE_SUFFIX_TYPE;
```

QUEUE_SUFFIX_TYPE_NONE: There is no suffix.

QUEUE_SUFFIX_TYPE_JOURNAL: Refers to the **queue journal** of the queue identified by the unnamed union of the **QUEUE_FORMAT** structure.

QUEUE_SUFFIX_TYPE_DEADLETTER: Refers to the non-transacted **dead letter queue** of the machine identified by the unnamed union of the **QUEUE_FORMAT** structure.

QUEUE_SUFFIX_TYPE_DEADXACT: Refers to the transacted dead letter queue of the machine identified by the unnamed union of the **QUEUE_FORMAT** structure.

QUEUE_SUFFIX_TYPE_XACTONLY: Refers to the transacted connected queue of the connector queue identified by the unnamed union of the **QUEUE_FORMAT** structure.

QUEUE_SUFFIX_TYPE_SUBQUEUE: Refers to the subqueue that is the direct name identified by the unnamed union of the **QUEUE_FORMAT** structure.<5>

2.2.3.3 OBJECTID

```
typedef struct _OBJECTID {
  GUID Lineage;
  unsigned long Uniquifier;
} OBJECTID;
```

Lineage: The globally unique identifier (GUID) of the queue manager where the queue is hosted.

Uniquifier: The unique identifier for the **private queue**. Uniquely identifies a queue hosted at the queue manager identified by **Lineage**.

2.2.3.4 DL_ID

The **DL_ID** structure defines a distribution list identifier. Not supported by this protocol.

```
typedef struct _DL_ID {
  GUID m_DlGuid;
  wchar_t* m_pwzDomain;
} DL_ID;
```

m_DlGuid: The globally unique identifier (GUID) of the distribution list.

m_pwzDomain: The domain of the distribution list.

2.2.3.5 MULTICAST_ID

The **MULTICAST_ID** structure defines a multicast identifier. Not supported by this protocol.

Copyright © 2008 Microsoft Corporation.

```
typedef struct _MULTICAST_ID {
  unsigned long m_address;
  unsigned long m_port;
} MULTICAST_ID;
```

m_address: The IP multicast address.

m_port: The port to which the queue is attached.

2.2.3.6 QUEUE_FORMAT Structure

The **QUEUE_FORMAT** structure describes the type of queue being managed and an identifier for that queue.

```
typedef struct QUEUE_FORMAT {
  unsigned char m_qft;
  unsigned char m_SuffixAndFlags;
  unsigned short m_reserved;
  [switch_is(m_qft)] union {
    [case(QUEUE_FORMAT_TYPE_PUBLIC)]
      GUID m_gPublicID;
    [case(QUEUE_FORMAT_TYPE_PRIVATE)]
      OBJECTID m_oPrivateID;
    [case(QUEUE_FORMAT_TYPE_DIRECT)]
      wchar_t* m_pDirectID;
    [case(QUEUE_FORMAT_TYPE_MACHINE)]
      GUID m_gMachineID;
    [case(QUEUE_FORMAT_TYPE_CONNECTOR)]
      GUID m_GConnectorID;
    [case(QUEUE_FORMAT_TYPE_DL)]
      DL_ID m_DlID;
    [case(QUEUE_FORMAT_TYPE_MULTICAST)]
      MULTICAST_ID m_MulticastID;
    [case(QUEUE_FORMAT_TYPE_SUBQUEUE)]
      wchar_t* m_pDirectSubqueueID;
  };
} __QUEUE_FORMAT;
```

m_qft: The type of queue format name.

m_SuffixAndFlags: This field is broken into two subfields: **Suffix Type** is located in the four least significant bits, and **Flags** is located in the four most significant bits.

0	1	2	3	4	5	6	7
Flags				Suffix Type			

Copyright © 2008 Microsoft Corporation.

Flags	Meaning
QUEUE_FORMAT_FLAG_NOT_SYSTEM 0x00	Specified queue is not a system queue.
QUEUE_FORMAT_FLAG_SYSTEM 0x80	Specified queue is a system queue.

Suffix Type:	Meaning
QUEUE_SUFFIX_TYPE_NONE 0x00	No suffix specified. The **Flags** subfield MUST be set to 0x00. The **m_qft** member MUST NOT be set to 0x04.
QUEUE_SUFFIX_TYPE_JOURNAL 0x01	Journal suffix. The **Flags** subfield MUST be set to 0x80. The **m_qft** member MUST NOT be set to 0x05,0x06, or 0x07.
QUEUE_SUFFIX_TYPE_DEADLETTER 0x02	Dead-letter suffix. The **Flags** subfield MUST be set to 0x80. The **m_qft** member MUST NOT be set to 0x01, 0x02, 0x05,0x06 or 0x07.
QUEUE_SUFFIX_TYPE_DEADXACT 0x03	Transacted dead letter suffix. The **Flags** subfield MUST be set to 0x80. The **m_qft** member MUST be set to 0x03 or 0x04.
QUEUE_SUFFIX_TYPE_XACTONLY 0x04	Transaction-only suffix. The **m_qft** member MUST be set to 0x05.
QUEUE_SUFFIX_TYPE_SUBQUEUE 0x05	Subqueue suffix. The **Flags** subfield MUST be 0x00. The **m_qft** member MUST be set to 0x08.<6>

m_reserved: The integer value used for padding. The client SHOULD set this value to 0. The server MUST not use it.

: Based on the value of **m_qft**.

m_gPublicID: A globally unique identifier (GUID) of a **public queue**. Selected when **m_qft** is set to 0x01.

m_oPrivateID: An **OBJECTID (section 2.2.3.3)** of a private queue; members MUST be used as specified in **OBJECTID**. Selected when **m_qft** is set to 0x02.

m_pDirectID: Name identifier of a direct queue. Selected when **m_qft** is set to 0x03. The name MUST conform to one of the following formats in ABNF notation:

QueueDirectName = PrivateQueuePath / PublicQueuePath / MachineQueuePath

PrivateQueuePath = "DIRECT=" Protocol ":" ProtocolAddressSpecification "\PRIVATE$\" QueueName [";JOURNAL"]

PublicQueuePath = "DIRECT=" Protocol ":" ProtocolAddressSpecification "\" QueueName [";JOURNAL"]

MachineQueuePath = "DIRECT=" Protocol ":" ProtocolAddressSpecification "\SYSTEM$;" ("JOURNAL" / "DEADLETTER" / "DEADXACT")

Protocol = "TCP" / "OS" / "HTTP" / "HTTPS" / "IPX"

Copyright © 2008 Microsoft Corporation.

QueueName = 1*124VCHAR

VCHAR = "!" / %x23-26 / %x28-2A / %x2C-3A / %x3C-5B / %x5D-7E

< ProtocolAddressSpecification > is the protocol specific address format as defined in the following table.<7>

Protocol	Description	Protocol address specification
TCP	Connection-oriented TCP over IP.	Internet address notation (IP address).
IPX	Connection-oriented SPX over IPX.	Network number and host number (separated by the ":" character). Not supported by this protocol.
OS	Connection using the native computer-naming convention.	Any computer name supported by the underlying operating system.
HTTP	HTTP transport	IP address or full DNS name (computer name within an enterprise) followed by the virtual directory name, separated by a forward slash. Not supported by this protocol.
HTTPS	Secure HTTP transport through a Secure Sockets Layer (SSL) connection.	IP address or full DNS name (computer name within an enterprise) followed by the virtual directory name separated by a forward slash. Not supported by this protocol.

m_gMachineID: The GUID of a machine. Selected when **m_qft** is set to 0x04.

m_GConnectorID: The GUID of a connector queue. Selected when **m_qft** is set to 0x05. Not supported by this protocol.

m_DlID: The identifier of a distribution list. Selected when **m_qft** is set to 0x06. Not supported by this protocol.

m_MulticastID: The identifier of a multicast queue. Selected when **m_qft** is set to 0x07. Not supported by this protocol.

m_pDirectSubqueueID: The identifier of a subqueue. Selected when **m_qft** is set to 0x08. The format for the subqueue ID is the same as that of m_pDirectID.<8>

The direct name subqueue ID MUST NOT be associated with a multicast ID, a system queue (dead-letter and journal) or another subqueue.

The following ABNF notation defines the format of a subqueue identifier :

SubQueuePath = QueueDirectName ";" SubqueueName

SubqueueName = 1*64VCHAR

; QueueDirect Name as specified in m_pDirectID

; VCHAR as specified in m_pDirectID

Copyright © 2008 Microsoft Corporation.

2.2.4 QueueContextHandle

The **QueueContextHandle** is a data type that defines an RPC context handle corresponding to an open queue. A client MUST call **R_OpenQueue (section 3.1.4.2)** or **R_OpenQueueForMove (section 3.1.4.11)** to create a **QueueContextHandle** and **R_CloseQueue (section 3.1.4.3)** to delete a **QueueContextHandle**. More details are specified in section 3.1.4.

This type is declared as follows:

```
typedef [context_handle] void* QueueContextHandle;
```

2.2.5 Message Packet Structure

The Message Packet Structure is the data structure that contains the **UserMessage** and other headers that represent the payload that is transferred across the wire as a result of a remote read operation. More details are specified in **R_StartReceive (section 3.1.4.7)** and **R_StartTransactionalReceive (section 3.1.4.13)**.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
UserMessage (variable)																															
...																															
ExtensionHeader (optional)																															
...																															
...																															
SubqueueHeader (optional)																															
...																															
...																															
...																															
...																															
...																															

Copyright © 2008 Microsoft Corporation.

...
...
(SubqueueHeader (optional) cont'd for 29 rows)
DeadLetterHeader (variable)
...
ExtendedAddressHeader (optional)
...
...
...
...
...
...

UserMessage (variable): A **UserMessage** structure, as specified in section 2.2.5.1.

ExtensionHeader (12 bytes): An **ExtensionHeader** structure, as specified in section 2.2.5.2.

SubqueueHeader (148 bytes): A **SubqueueHeader** structure, as specified in section 2.2.5.3.

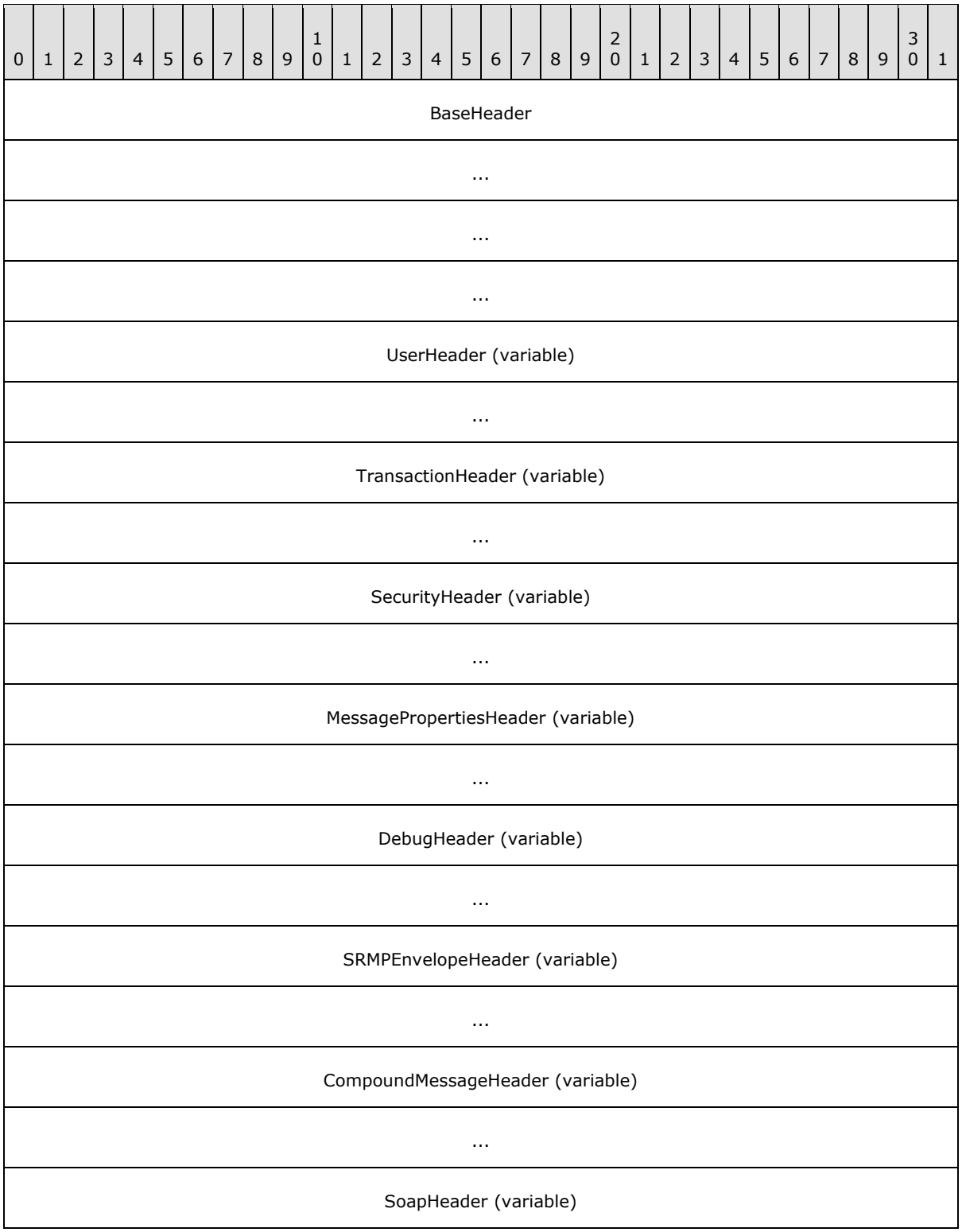
DeadLetterHeader (variable): A **DeadLetterHeader** structure, as specified in section 2.2.5.4.

ExtendedAddressHeader (28 bytes): An **ExtendedAddressHeader** structure, as specified in section 2.2.5.5.

2.2.5.1 UserMessage

The UserMessage structure can be either a Binary Message (section 2.2.5.1.1) or an SRMP Message (section 2.2.5.1.2).

Copyright © 2008 Microsoft Corporation.



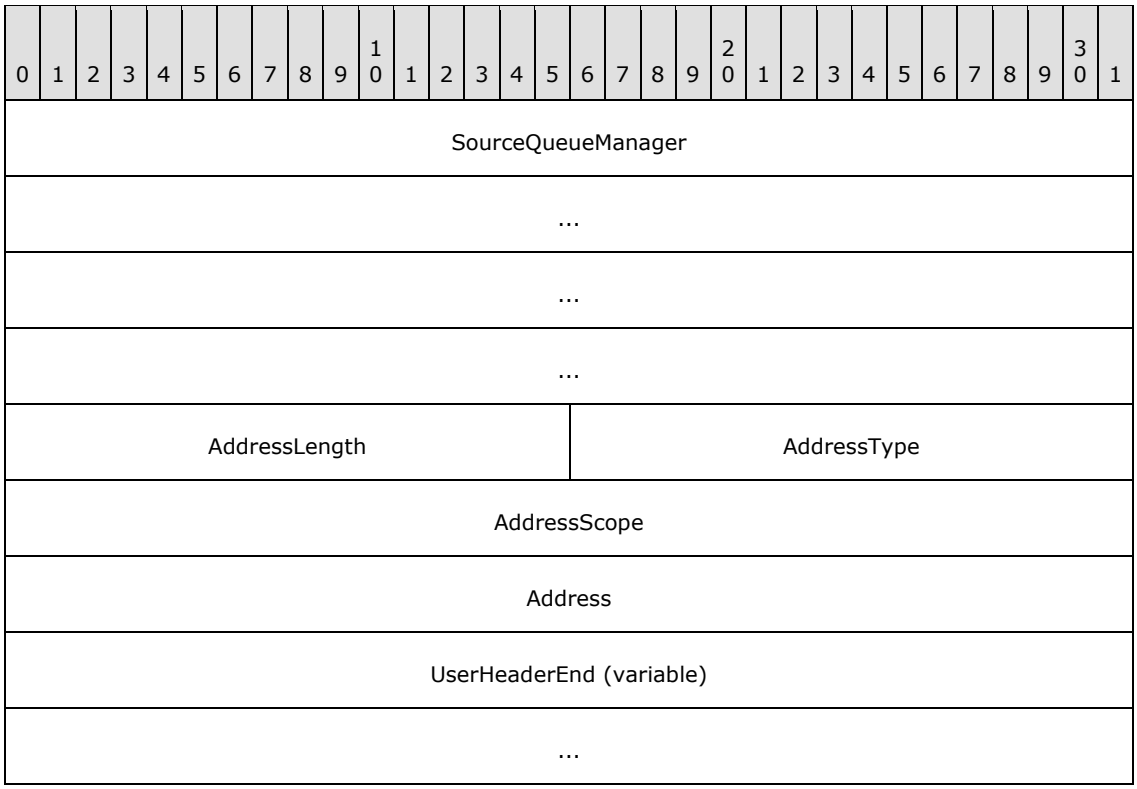
Copyright © 2008 Microsoft Corporation.

...
MultiQueueFormatHeader (variable)
...
SessionHeader (optional)
...
...
...

BaseHeader (16 bytes): A BaseHeader as specified in [MS-MQQB] section 2.2.2.1. The **TimeToReachQueue** field has the same length and format as that specified in [MS-MQQB], but differs in that it represents the absolute expiration time of the message as the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time).

UserHeader (variable): A UserHeader (as specified in [MS-MQQB] section 2.2.2.3) with the following field overlays, which pertain when the UserHeader specifies that the destination queue is a direct format name. In this case, the **QueueManagerAddress** specifies the host address from which a message was received.

Copyright © 2008 Microsoft Corporation.



SourceQueueManager (16 bytes): A **GUID**, as specified in [MS-DTYP], that identifies the sender of the message.

AddressLength (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be the actual address length in the **Address** field.

AddressType (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be set to one of the following values:

Value	Meaning
IP_ADDRESS_TYPE 0x0001	The address specified in the **Address** field is an IPv4 address.
IPV6_ADDRESS_TYPE 0x0006	The address specified in the **Address** field is an IPv6 address.

AddressScope (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be set either to the IPv6 address scope if **AddressType** is IPV6_ADDRESS_TYPE, or otherwise to 0. More details are specified in [RFC2553] section 3.3.

Address (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST contain the address of the host from which the message was received. The field MUST contain as much of the address as can fit in the field. More details are specified in [RFC2553] section 3.3.

UserHeaderEnd (variable): A variable-length buffer mapped by UserHeader (as specified in [MS-MQQB] section 2.2.2.3) beginning with the **TimeToBeReceived** field.

Copyright © 2008 Microsoft Corporation.

TransactionHeader (variable): A TransactionHeader, as specified in [MS-MQQB] section 2.2.8.5.

SecurityHeader (variable): A SecurityHeader as specified in [MS-MQQB] section 2.2.8.6.

MessagePropertiesHeader (variable): A MessagePropertiesHeader as specified in [MS-MQQB] section 2.2.2.4.

DebugHeader (variable): A DebugHeader as specified in [MS-MQQB] section 2.2.8.8.

SRMPEnvelopeHeader (variable): An SRMPEnvelopeHeader (section 2.2.5.1.2.1).

CompoundMessageHeader (variable): A CompoundMessageHeader (section 2.2.5.1.2.2).

SoapHeader (variable): A SoapHeader as specified in [MS-MQQB] section 2.2.8.7.

MultiQueueFormatHeader (variable): A MultiQueueFormatHeader as specified in [MS-MQQB] section 2.2.8.1.

SessionHeader (16 bytes): A SessionHeader as specified in [MS-MQQB] section 2.2.8.4.

More details about the following individual headers, with the exceptions of SRMPEnvelopeHeader (section 2.2.5.1.2.1) and CompoundMessageHeader (section 2.2.5.1.2.2), are specified in [MS-MQQB] section 2.2.8.

In addition, the following exceptions also exist on the field attributes as specified in [MS-MQQB]. The overall structure of the data is the same; however, particular fields have been overridden or have different meaning in this protocol. The size of each overridden field is the same size as the original field.

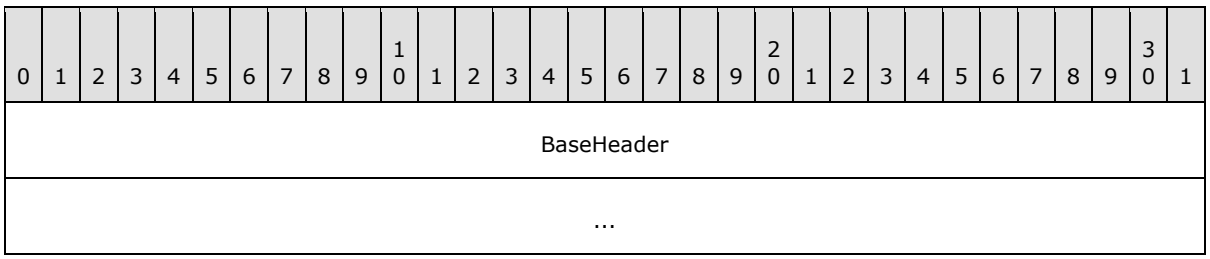
UserMessage.BaseHeader.TimeToReachQueue

The definition for **TimeToReachQueue** differs from what is specified in [MS-MQQB] section 2.2.8 in the following manner:

- In [MS-MQQB], this field indicates the length of time, in seconds, that a UserMessage Packet has to reach its destination queue manager.
- In [MS-MQRR], this field indicates the absolute expiration time of the message defined as the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time).

2.2.5.1.1 Binary Message

When the **message packet** is in the binary format, the UserMessage (section 2.2.5.1) is as specified below. The SRMPEnvelopeHeader (section 2.2.5.1.2.1) and the CompoundMessageHeader (section 2.2.5.1.2.2) are excluded.



Copyright © 2008 Microsoft Corporation.

...
...
UserHeader (variable)
...
TransactionHeader (variable)
...
SecurityHeader (variable)
...
MessagePropertiesHeader (variable)
...
DebugHeader (variable)
...
SoapHeader (variable)
...
MultiQueueFormatHeader (variable)
...
SessionHeader (optional)
...
...
...

BaseHeader (16 bytes): A BaseHeader as specified in section 2.2.5.1.

Copyright © 2008 Microsoft Corporation.

UserHeader (variable): A UserHeader as specified in section 2.2.5.1.

TransactionHeader (variable): A TransactionHeader as specified in section 2.2.5.1.

SecurityHeader (variable): A SecurityHeader as specified in section 2.2.5.1.

MessagePropertiesHeader (variable): A MessagePropertiesHeader as specified in section 2.2.5.1.

DebugHeader (variable): A DebugHeader as specified in section 2.2.5.1.

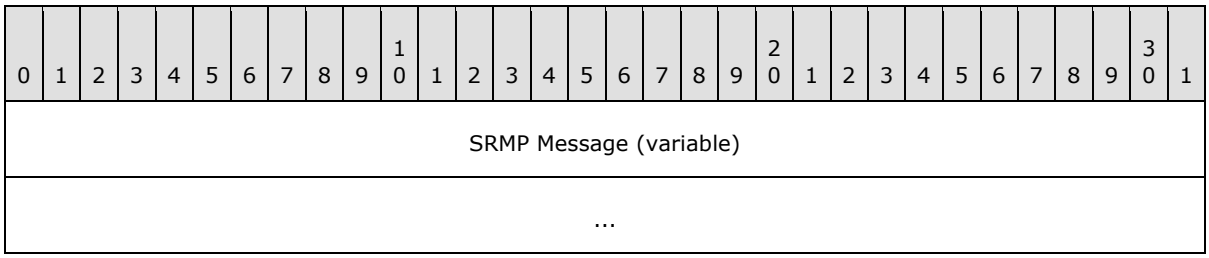
SoapHeader (variable): A SoapHeader as specified in section 2.2.5.1.

MultiQueueFormatHeader (variable): A MultiQueueFormatHeader as specified in section 2.2.5.1.

SessionHeader (16 bytes): A SessionHeader as specified in section 2.2.5.1.

2.2.5.1.2 SRMP Message

When the message packet is in the SRMP format, the full UserMessage (section 2.2.5.1) is sent, including both the SRMPEnvelopeHeader (section 2.2.5.1.2.1) and the CompoundMessageHeader (section 2.2.5.1.2.2).

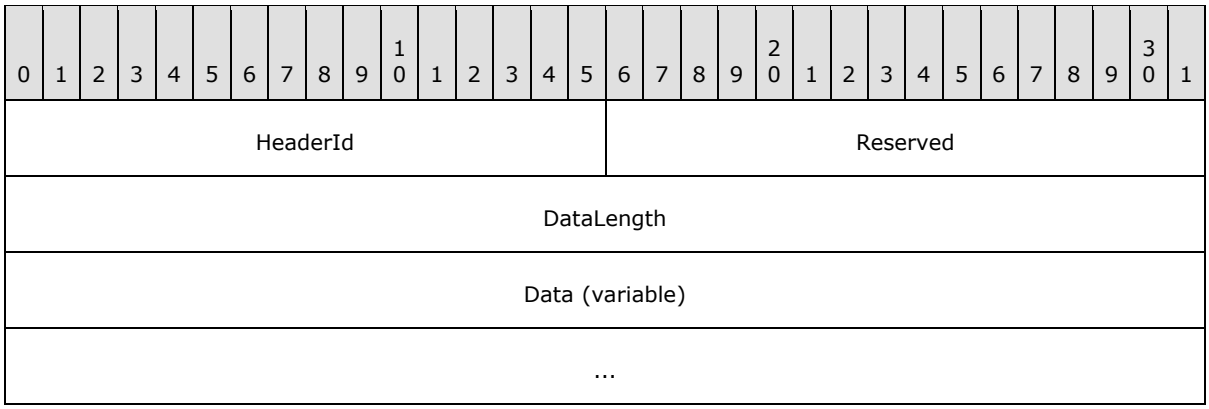


SRMP Message (variable): A UserMessage (section 2.2.5.1).

2.2.5.1.2.1 SRMPEnvelopeHeader

The SRMPEnvelopeHeader contains information about the SOAP envelope. This header MUST be present if the packet is in SOAP Reliable Messaging Protocol (SRMP) format.

Copyright © 2008 Microsoft Corporation.



HeaderId (2 bytes): A **USHORT**, as specified in [MS-DTYP], that specifies the identification number of the header.

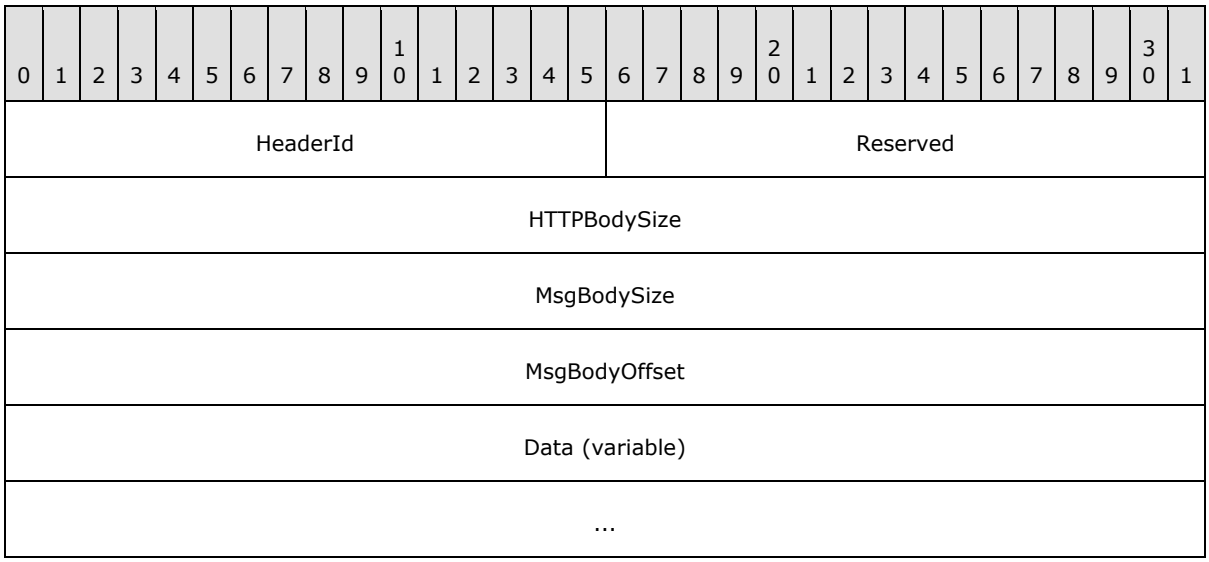
Reserved (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be ignored.

DataLength (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be the length of the data in the **Data** field.

Data (variable): Specifies the data in **WCHAR**, as specified in [MS-DTYP], including the NULL terminator. The data is formatted as an SRMP Message structure, as specified in [MC-MQSRM] section 2.2.2.

2.2.5.1.2.2 CompoundMessageHeader

The CompoundMessageHeader contains information about the SRMP compound message as specified in [MC-MQSRM] section 2.2.2. This header MUST be present if the packet is in SRMP format.



HeaderId (2 bytes): A **USHORT**, as specified in [MS-DTYP], that specifies the identification number of the header.

Copyright © 2008 Microsoft Corporation.

Reserved (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be ignored.

HTTPBodySize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be the size of the **Data** field in bytes.

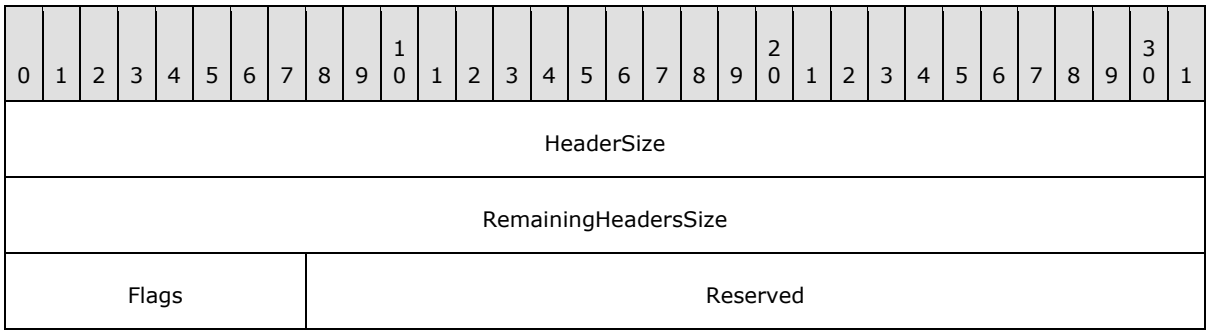
MsgBodySize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be the size, in bytes, of the message body within the **Data** field.

MsgBodyOffset (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be set to the offset of the message body within the **Data** field.

Data (variable): Specifies an array of bytes that contains the SRMP message, including the HTTP POST message that carried the SRMP message. More details are specified in [MC-MQSRM] section 4.1.

2.2.5.2 ExtensionHeader

The ExtensionHeader contains information about the presence and size of other headers in the Message Packet structure (section 2.2.5). This header MUST be included in the Message Packet structure. <9>



HeaderSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the size in bytes of the ExtensionHeader.

RemainingHeadersSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be the sum of sizes in bytes of all headers that follow ExtensionHeader.

Flags (1 byte): Indicates the presence or absence of other headers in the Message Packet structure. Any combination of the following values is acceptable.

0	1	2	3	4	5	6	7
DL	SQ	X2	DI	EA	X5	X6	X7

Where the bits are defined as:

Value	Description
DL	MUST be set to 1 if the Message Packet structure contains the DeadLetterHeader (section 2.2.5.4). MUST be set to 0 otherwise.

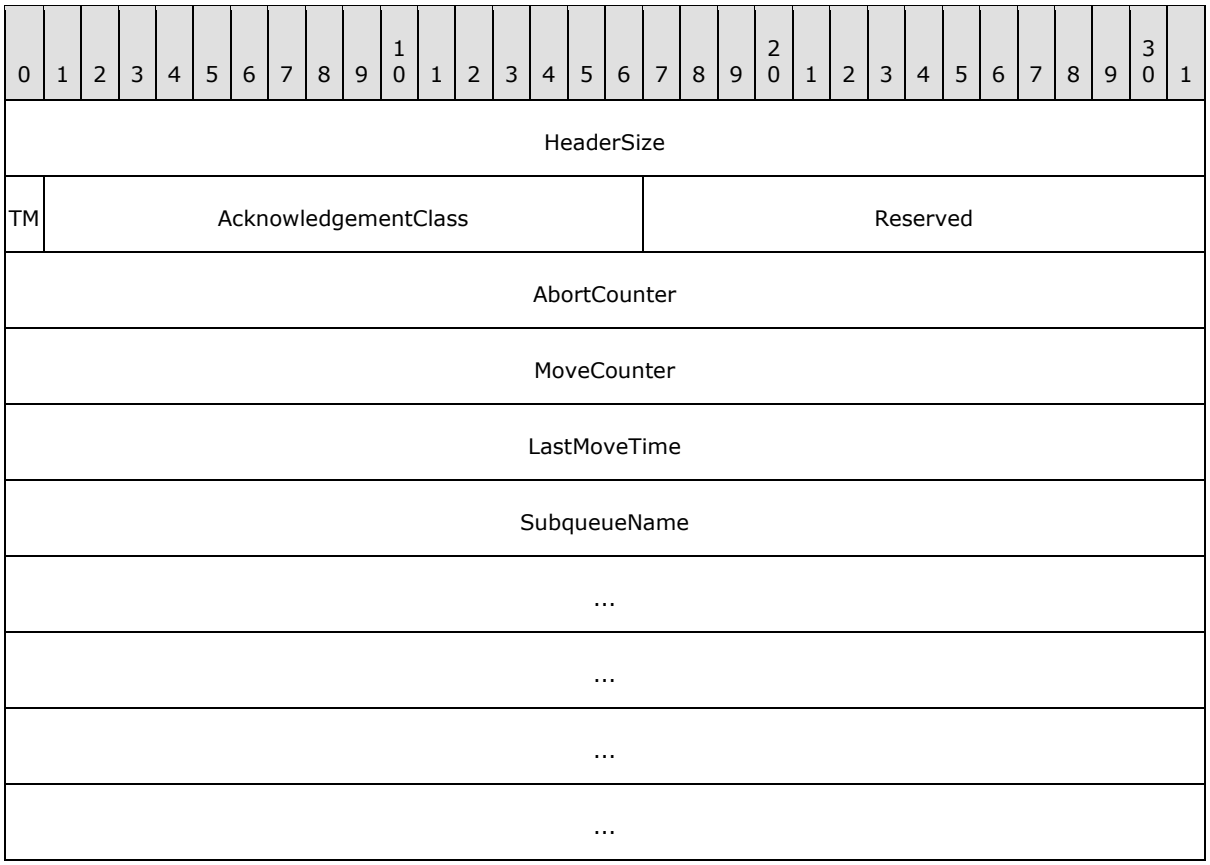
Copyright © 2008 Microsoft Corporation.

Value	Description
SQ	MUST be set to 1 if the Message Packet structure contains the SubqueueHeader (section 2.2.5.3). MUST be set to 0 otherwise.
X2	Unused bit field. MUST be ignored.
DI	MUST be set to 1 if the dead-letter queue as specified by the DeadLetterHeader is invalid. MUST be set to 0 otherwise. If the DeadLetterHeader is not included, this field MUST be ignored when reading the message packet.
EA	MUST be set to 1 if the Message Packet Structure contains the ExtendedAddressHeader (section 2.2.5.5). MUST be set to 0 otherwise.
X5	Unused bit field. MUST be ignored.
X6	Unused bit field. MUST be ignored.
X7	Unused bit field. MUST be ignored.

Reserved (3 bytes): MUST be ignored when reading the Message Packet structure.

2.2.5.3 SubqueueHeader

The SubqueueHeader encapsulates information about the message as specified below. This header SHOULD be present if the message belongs to a subqueue. <10>



Copyright © 2008 Microsoft Corporation.

...
...
...
(SubqueueName cont'd for 8 rows)
TargetSubqueueName
...
...
...
...
...
...
...
(TargetSubqueueName cont'd for 8 rows)

HeaderSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the size in bytes of the SubqueueHeader.

TM (1 bit): A one-**BIT** flag, as specified in [MS-DTYP], that MUST be 0.

AcknowledgementClass (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST specify the acknowledgment class of the message. See [MS-MQQB] section **2.2.1.5**.

Reserved (15 bits): MUST be ignored.

AbortCounter (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the number of sequentially failed attempts to read the message or to move the message. See sections 3.1.4.13, 3.1.4.10, and 3.1.6.2.

MoveCounter (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the number of times that the message has been moved. See section 3.1.4.10.

LastMoveTime (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the local time of the most recent move of the message. The time is specified as the number of milliseconds elapsed since midnight of January 1, 1970. If the message has never been moved, this value is 0. See section 3.1.4.10.

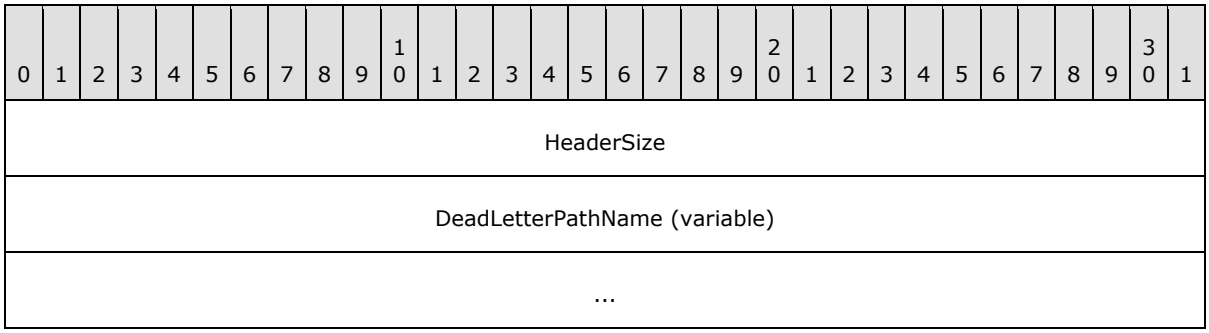
Copyright © 2008 Microsoft Corporation.

SubqueueName (64 bytes): If the message belongs to a subqueue, the value MUST contain the null-terminated Unicode string that specifies the subqueue name. If the subqueue name is shorter than the field size, the remaining bytes MUST be set to 0. If the message does not belong to the subqueue, all bytes MUST be set to 0.

TargetSubqueueName (64 bytes): If the message is participating in the transacted Move operation that is not yet committed or aborted, this field MUST contain the null-terminated Unicode string that specifies the target subqueue name. If the subqueue name is shorter than the field size, the remaining bytes MUST be set to 0. If the message is not part of a transacted Move operation, all bytes MUST be set 0.

2.2.5.4 DeadLetterHeader

The DeadLetterHeader specifies the path of an application-specified dead-letter queue. This header SHOULD be present if the message belongs to a dead-letter queue.<11>



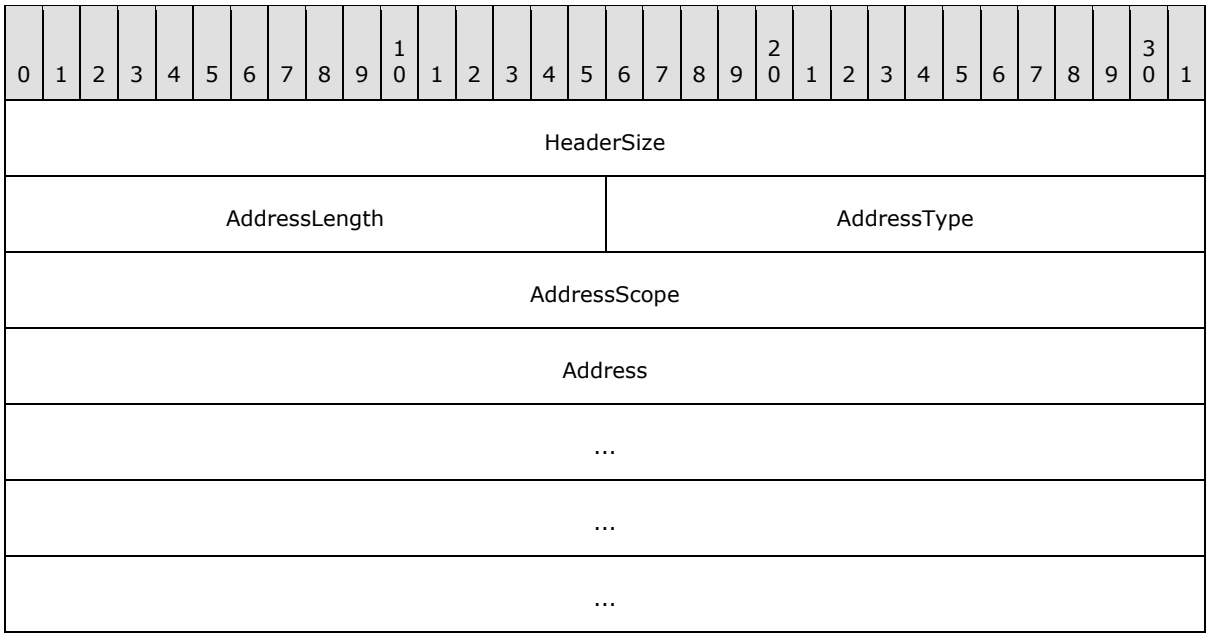
HeaderSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be set to the total size in bytes of the DeadLetterHeader.

DeadLetterPathName (variable): MUST contain a null-terminated Unicode string that specifies the application-specified dead-letter queue. The array MUST be aligned up to the next 4-byte boundary by adding padding zeros if necessary.

2.2.5.5 ExtendedAddressHeader

The ExtendedAddressHeader specifies the host address from which a message was received using a direct format name.<12>

Copyright © 2008 Microsoft Corporation.



HeaderSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the size, in bytes, of the ExtendedAddressHeader.

AddressLength (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be the actual address length in the **Address** field.

AddressType (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be set to one of the following values:

Value	Meaning
IP_ADDRESS_TYPE 0x0001	The address specified in the **Address** field is an IPv4 address.
IPV6_ADDRESS_TYPE 0x0006	The address specified in the **Address** field is an IPv6 address.

AddressScope (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be set either to the IPv6 address scope if **AddressType** is IPV6_ADDRESS_TYPE, or otherwise to 0. See [RFC2553] section 3.3.

Address (16 bytes): An array of **UCHAR**, as specified in [MS-DTYP], that MUST contain the host address from which the message was received. If the **AddressType** is IP_ADDRESS_TYPE, the address MUST be in IPv4 address format. If the **AddressType** is IPV6_ADDRESS_TYPE, the address MUST be in IPv6 address format. See [RFC2553] section 3.3.

2.2.6 SectionBuffer

A **SectionBuffer** represents a fragment or section of a Message Packet. Operations **R_StartReceive (section 3.1.4.7)** and **R_StartTransactionalReceive (section 3.1.4.13)** fragment a Message Packet into an array of one or more **SectionBuffer** structures. The client concatenates these fragments to reconstruct a valid Message Packet. There may be up to two

Copyright © 2008 Microsoft Corporation.

sections per message. A Message Packet is split into two sections only when a subset of the distinguished message body property is returned. The first section always contains the message body property up to the size requested.

```
typedef struct _SectionBuffer {
  SectionType SectionBufferType;
  DWORD SectionSizeAlloc;
  DWORD SectionSize;
  [unique, size_is(SectionSize)]
    byte* pSectionBuffer;
} SectionBuffer;
```

SectionBufferType: MUST specify a type for the **SectionBuffer** structure that indicates whether the **pSectionBuffer** member contains the whole Message Packet, or if not, then which of the two sections it does contain. The SectionType (section 2.2.7) enumeration lists possible values. More details are specified in 2.2.7.

SectionSizeAlloc: MUST specify the original size (in bytes) of the part of the Message Packet that this **SectionBuffer** represents. When the **SectionBuffer** represents the first section of the message, this field specifies the size that the **SectionBuffer** would have been if the entire message body property were included. The difference between **SectionSizeAlloc** and **SectionSize** represents the size of the message body that was not transferred.

If the **SectionBufferType** is stFullPacket, stBinarySecondSection, or stSrmpSecondSection, then **SectionSizeAlloc** MUST be equal to **SectionSize**.

If the **SectionBufferType** is stBinaryFirstSection or stSrmpFirstSection, then **SectionSizeAlloc** MUST be equal to or greater than **SectionSize**.

SectionSize: MUST be the size (in bytes) of the buffer pointed to by **pSectionBuffer**. **SectionSize** specifies the size of the part of the Message Packet contained in **pSectionBuffer**.

pSectionBuffer: MUST be a pointer to an array of bytes containing a section of the Message Packet.

2.2.7 SectionType

The **SectionType** enumeration defines the available **SectionBuffer** types.

```
typedef  enum
{
  stFullPacket = 0,
  stBinaryFirstSection = 1,
  stBinarySecondSection = 2,
  stSrmpFirstSection = 3,
  stSrmpSecondSection = 4
} SectionType;
```

stFullPacket: The **pSectionBuffer** element of the **SectionBuffer** structure contains a complete Message Packet. The UserMessage is either that specified in section 2.2.5.1.1 or in section 2.2.5.1.2.

Copyright © 2008 Microsoft Corporation.

stBinaryFirstSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the first section of the Message Packet up to, but not beyond the MessagePropertiesHeader in the UserMessage as specified in section 2.2.5.1.1.

stBinarySecondSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the second section of the Message Packet from beyond the end of the MessagePropertiesHeader in the UserMessage (as specified in section 2.2.5.1.1) to the end of the packet.

stSrmpFirstSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the first section of the Message Packet up to, but not beyond the CompoundMessageHeader in the UserMessage, as specified in section 2.2.5.1.2.

stSrmpSecondSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the second section of the Message Packet from beyond the end of the CompoundMessageHeader in the UserMessage (as specified in section 2.2.5.1.2) to the end of the packet.

Copyright © 2008 Microsoft Corporation.

3 Protocol Details

The following sections specify details of the Message Queuing (MSMQ): Queue Manager Remote Read Protocol, including abstract data models, interface method syntax, and message processing rules.

3.1 RemoteRead Server Details

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to explain how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

Servers MUST maintain the following data elements.

Queue table: A table of queues deployed at the server, keyed by name. Each entry MUST contain the following:

- The name of the queue.
- The current queue state, as defined by the states in the Queue State Diagram (section 3.1.1.1).
- The queue shared open count.
- A Boolean transaction flag that indicates whether the queue supports transacted operations.
- A Message List.

QueueContextHandle table: A table of QueueContextHandles, keyed by **QueueContextHandle**. Each entry MUST contain the following:

- The **QueueContextHandle**.
- The Access Mode specified when the queue was opened.
- A reference to the queue table entry.

Cursor table: A table of cursor handles, keyed by a {cursor handle, **QueueContextHandle**} key pair. Each entry MUST contain the following:

- The cursor handle.
- The **QueueContextHandle** specified when the queue was created.
- The current position of the cursor within the queue.
- The current cursor state, as defined by the states in the Cursor State Diagram (section 3.1.1.2).

Pending Request table: A table of pending requests, keyed by a {request ID and **QueueContextHandle**} key pair. Each entry MUST contain the following:

- The type of pending request set to one of the following values:
  - ReceiveWaitForMessage
  - PeekWaitForMessage

Copyright © 2008 Microsoft Corporation.

- ReceiveWaitForEndOrCancel

- A request ID, supplied by the client.
- The **QueueContextHandle** specified with the request.
- The expiration time of the request.
- The *LookupId* of a message associated with the request.
- The cursor handle associated with the request.

Transactional Request table: A table of request lists, keyed by a transaction identifier. Each entry of the list MUST contain the following:

- The type of operation (**R_StartTransactionalReceive (section 3.1.4.13)** or **R_MoveMessage (section 3.1.4.10)**).
- The *LookupId* of a message associated with the request.
- The **QueueContextHandle** of a source queue.
- The **QueueContextHandle** of the destination queue, when the type of operation is **R_MoveMessage (Opnum 10)**.

Message List: A linked list of messages in a queue, in ascending order by arrival time within message priority. Each entry MUST contain the following:

- A server-assigned *LookupId* for the message, unique across all messages in the queue.
- A message locked flag indicating whether the message is locked by a pending request.
- The transaction identifier of the transaction the message is currently locked by (if any).
- The offset and byte size of the message headers, message body, and message trailers.
- The type of the message packet, either binary or SRMP.
- An acknowledgment class of the message.
- The time when the message arrived in the queue.

3.1.1.1 Queue State Diagram

The following diagram specifies the state transitions for a queue. In particular, it specifies the queue state transitions resulting from Open and Close events, and how the sharing mode parameter specified on Open affects these state transitions.

Copyright © 2008 Microsoft Corporation.

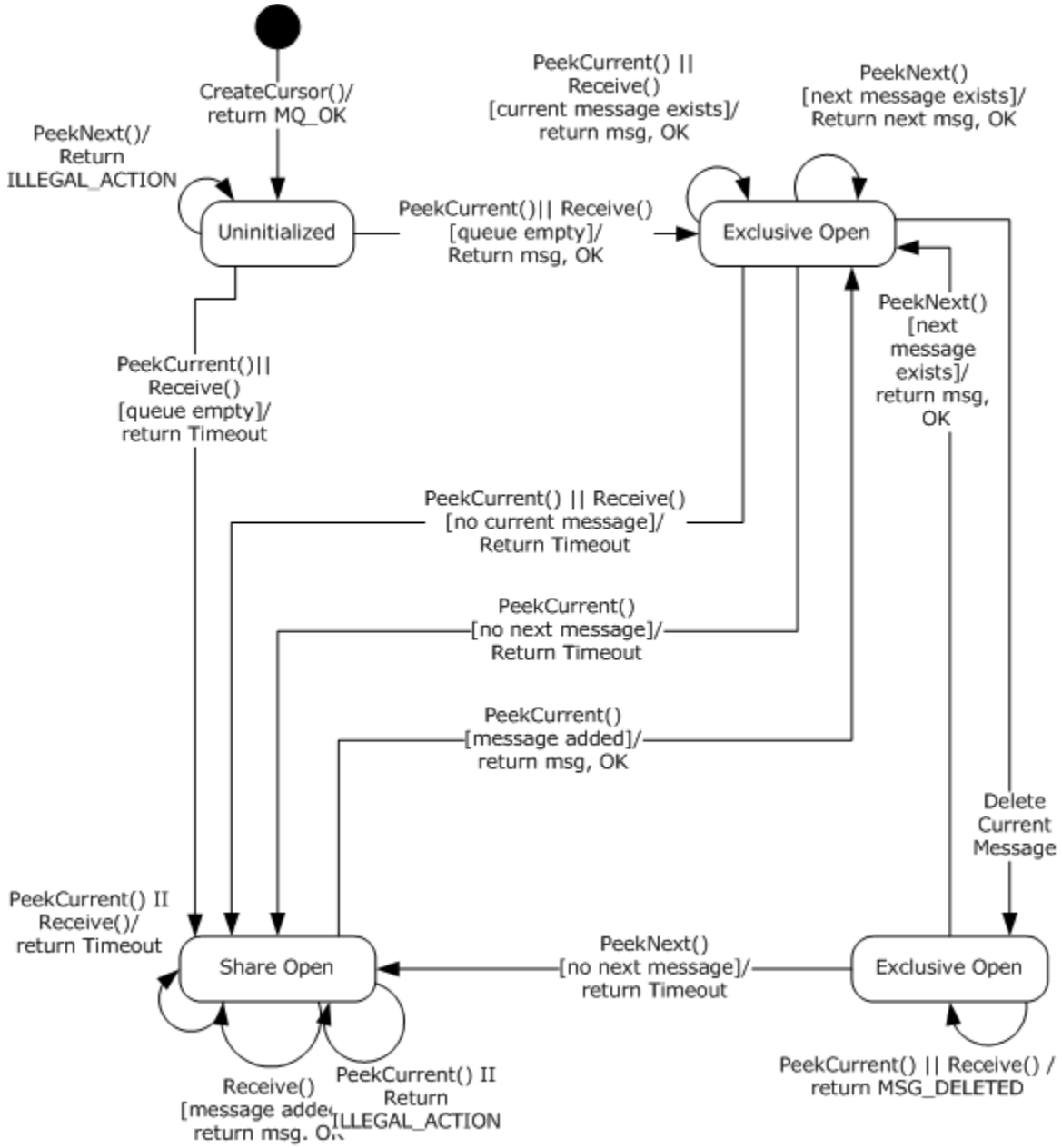


Figure 1: Queue state

3.1.1.2 Cursor State Diagram

The following diagram specifies the state transitions for a cursor. A cursor represents a position in a queue. The diagram specifies how cursor state transitions are affected by PeekCurrent, PeekNext, Receive, Message Added, and Message Deleted events, as well as by the presence or absence of messages in the queue

Copyright © 2008 Microsoft Corporation.

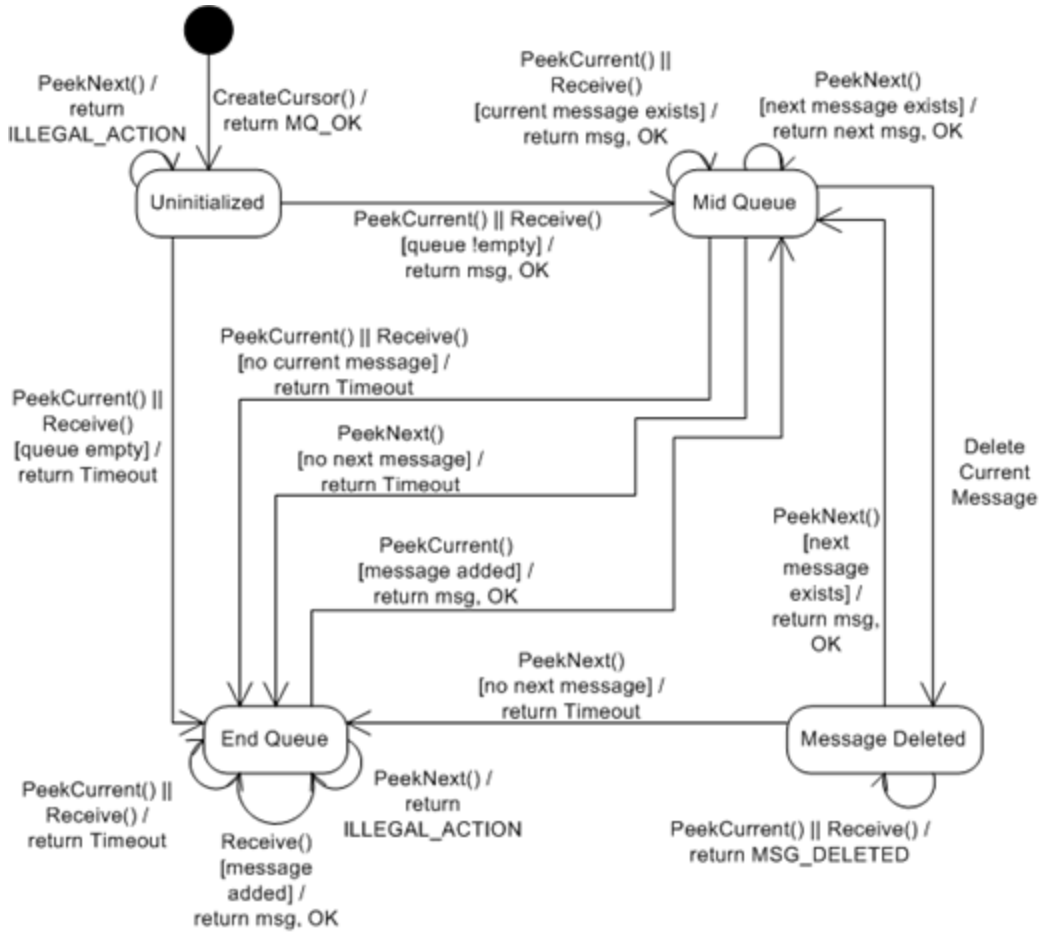


Figure 2: Cursor state

3.1.2 Timers

This protocol uses non-default behavior for the RPC Call Timeout timer, as specified in [MS-RPCE] section 3.3.2.2.2. This protocol uses a timer value of 300000 milliseconds,<13> which applies to the following method calls.

- **R_OpenQueue (Opnum 2) (section 3.1.4.2)**
- **R_OpenQueueForMove (Opnum 11) (section 3.1.4.11)**
- **R_QMEnlistRemoteTransaction (Opnum 12) (section 3.1.4.12)**

Call Timer: The server MUST maintain a per-call timer for each call to **R_StartReceive (Opnum 7) (section 3.1.4.7)** or **R_StartTransactionalReceive (Opnum 13) (section 3.1.4.13)** in which the *dwTimeout* parameter is nonzero. The timer MUST be set to the *dwTimeout* parameter that is specified on the call.

3.1.3 Initialization

The server MUST listen on the RPC protocols, as specified in section 2.1.

Copyright © 2008 Microsoft Corporation.

3.1.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 6.0, as specified in [MS-RPCE] section 3.

This protocol MUST indicate to the RPC runtime that it is to reject a NULL unique or full pointer with nonzero conformant value, as specified in [MS-RPCE] section 3.

The RemoteRead interface includes the following methods.

Methods in RPC Opnum Order

Method	Description
R_GetServerPort (section 3.1.4.1)	Returns an RPC endpoint port number to use in subsequent calls on the interface. Opnum: 0
Opnum1NotUsedOnWire	Reserved for local use. Opnum: 1
R_OpenQueue (section 3.1.4.2)	Opens a queue. Opnum: 2
R_CloseQueue (section 3.1.4.3)	Closes a queue. Opnum: 3
R_CreateCursor (section 3.1.4.4)	Opens a cursor on a queue. Opnum: 4
R_CloseCursor (section 3.1.4.5)	Closes a cursor. Opnum: 5
R_PurgeQueue (section 3.1.4.6)	Deletes all messages in a queue. Opnum: 6
R_StartReceive (section 3.1.4.7)	Initiates a Receive or Peek request on the queue. Opnum: 7
R_CancelReceive (section 3.1.4.8)	Cancels a pending Receive request. Opnum: 8
R_EndReceive (section 3.1.4.9)	Finishes a Receive request. Opnum: 9
R_MoveMessage (section 3.1.4.10)	Moves a message between two queues. Opnum: 10
R_OpenQueueForMove (section 3.1.4.11)	Opens a queue to be a destination for a move operation. Opnum: 11
R_QMEnlistRemoteTransaction (section 3.1.4.12)	Enlists in a transaction on a remote machine. Opnum: 12

Copyright © 2008 Microsoft Corporation.

Method	Description
R_StartTransactionalReceive (section 3.1.4.13)	Initiates a transactional receive request on the queue. Opnum: 13
R_SetUserAcknowledgementClass (section 3.1.4.14)	Changes the acknowledgment class for a message in a queue. Opnum: 14
R_EndTransactionalReceive (section 3.1.4.15)	Finishes a transactional receive request. Opnum: 15

Note In the preceding table, the term "Reserved for local use" means that the client MUST NOT send the opnum, and the server behavior is undefined.<14>

3.1.4.1 R_GetServerPort (Opnum 0)

The **R_GetServerPort** method returns the RPC endpoint port for the client to use in subsequent method calls on the RemoteRead interface.

The server MUST return the TCP port number for the RemoteRead RPC interface.

The client MAY call this method prior to calling any other method on the protocol. The client MAY use the returned value to obtain another RPC binding handle to use with the remaining methods on the protocol.<15>

```
DWORD R_GetServerPort(
  [in] handle_t hBind
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

Return Values: On success, this method MUST return a nonzero TCP port value for the RPC interface. If an error occurs, the server MUST return 0x00000000.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

If the server uses a static RPC endpoint for the RemoteRead interface, it MUST return that endpoint port number. If the server does not use a static RPC endpoint for the RemoteRead interface, it MUST return the dynamic endpoint that was assigned.

3.1.4.2 R_OpenQueue (Opnum 2)

The **R_OpenQueue** method opens a queue in preparation for subsequent operations against it. This method MUST be called prior to calling any of the following operations.

- R_CreateCursor (section 3.1.4.4)
- R_CloseCursor (section 3.1.4.5)
- R_PurgeQueue (section 3.1.4.6)

Copyright © 2008 Microsoft Corporation.

- **R_StartReceive (section 3.1.4.7)**
- **R_CancelReceive (section 3.1.4.8)**
- **R_EndReceive (section 3.1.4.9)**
- **R_MoveMessage (section 3.1.4.10)**, for the source queue only, as specified in section 3.1.4.10.
- **R_StartTransactionalReceive (section 3.1.4.13)**
- **R_SetUserAcknowledgementClass (section 3.1.4.14)**
- **R_EndTransactionalReceive (section 3.1.4.15)**

This method returns a **QueueContextHandle (section 2.2.4)** that is required as input in the operations listed above.

```
void R_OpenQueue(
  [in] handle_t hBind,
  [in] struct QUEUE_FORMAT* pQueueFormat,
  [in] DWORD dwAccess,
  [in] DWORD dwShareMode,
  [in] GUID* pClientId,
  [in] LONG fNonRoutingServer,
  [in] unsigned char Major,
  [in] unsigned char Minor,
  [in] USHORT BuildNumber,
  [in] LONG fWorkgroup,
  [out] QueueContextHandle* pphContext
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

pQueueFormat: MUST be a pointer to a **QUEUE_FORMAT structure** that identifies the queue to open. This value MUST NOT be NULL. The value of the **m_qft** field MUST be one of QUEUE_FORMAT_TYPE_PUBLIC, QUEUE_FORMAT_TYPE_PRIVATE, QUEUE_FORMAT_TYPE_DIRECT, QUEUE_FORMAT_TYPE_MACHINE, or QUEUE_FORMAT_TYPE_SUBQUEUE.

dwAccess: Specifies the requested type of access to the queue. MUST be set by the client to one of the following values.

Value	Meaning
RECEIVE_ACCESS 0x00000001	The returned **QueueContextHandle** can be used in the **R_StartReceive** or **R_StartTransactionalReceive** methods with *ulAction* set to either a Peek or Receive action type as defined in the table under the *ulAction* parameter in **R_StartReceive**.
PEEK_ACCESS 0x00000020	The returned **QueueContextHandle** can be used in the **R_StartReceive** method with *ulAction* set only to a Peek action type as defined in the table under the *ulAction* parameter in **R_StartReceive**.

dwShareMode: Specifies whether the client needs exclusive access to the queue. MUST be set by the client to one of the following values.

Copyright © 2008 Microsoft Corporation.

Value	Meaning
DENY_NONE 0x00000000	Permits multiple **QueueContextHandles** to the queue to be opened concurrently.
DENY_SHARE 0x00000001	Permits a single **QueueContextHandle** to the queue at a time, providing exclusive access to the queue.

pClientId: MUST be set by the client to a pointer to a valid globally unique identifier (GUID) that uniquely identifies the client.<16> The server SHOULD ignore this parameter. The server MAY use this parameter to impose a limit on the number of unique callers.<17> This value MUST NOT be NULL.

fNonRoutingServer: If the client is configured to operate in the role of a routing server (as specified in [MS-MQMA]), this parameter MUST be set to FALSE (0x00000000); otherwise, it MUST be set to TRUE (0x00000001).<18> If the value of the *fNonRoutingServer* parameter is FALSE (0x00000000), the server MUST ignore *pClientId*.

Name	Value
False	0x00000000
True	0x00000001

Major: MUST be set by the client to an implementation-specific Major Version number of the client. SHOULD be ignored by the server.<19>

Minor: MUST be set by the client to an implementation-specific Minor Version number of the client. SHOULD be ignored by the server.<20>

BuildNumber: MUST be set by the client to an implementation-specific Build Number of the client. SHOULD be ignored by the server.<21>

fWorkgroup: MUST be set to TRUE (0x00000001) by the client if the client machine is not a member of a Windows domain; otherwise, it MUST be set to FALSE (0x00000000). The RPC **authentication level** required by the server MAY be based on this value in subsequent calls on the interface.<22>

Name	Value
False	0x00000000
True	0x00000001

pphContext: MUST be set by the server to a **QueueContextHandle**.

Return Values: The method has no return values. If the method fails, an RPC exception is thrown.

Exceptions Thrown:

In addition to the exceptions thrown by the underlying RPC protocol as specified in [MS-RPCE], the method throws **HRESULT** failure codes as RPC exceptions. The client MUST treat all thrown **HRESULT** codes identically. The client MUST disregard all out-parameter values when any failure **HRESULT** is thrown.

Copyright © 2008 Microsoft Corporation.

When processing this call, the server MUST do the following:

- Look up the queue name in the queue table. If not found, throw MQ_ERROR_QUEUE_NOT_FOUND (0xC00E0003).
- Using the current state of the queue, process the request according to the following queue state table.

Current queue state	Requested share mode	Result	Effect on queue state
Closed	DENY _SHARE	Success	Transition to Exclusive Open state
Closed	DENY_NONE	Success	Transition to Share Open state
Exclusive Open	DENY_NONE or DENY _SHARE	Fail	No change
Share Open	DENY _SHARE	Fail	No change
Share Open	DENY_NONE	Success	Remain in Share Open state and Increment Shared Open Count

- If the Result from the previous table is Fail, throw MQ_ERROR_SHARING_VIOLATION (0xC00E0009).
- If the Result from the previous table is Success, do the following:
  - Adjust the queue entry state and share count as specified in the previous table.
  - Create a **QueueContextHandle**.
  - Add a new **QueueContextHandle** table entry with the following:
    - **QueueContextHandle** set to the created **QueueContextHandle**.
    - Access Mode set to *dwAccess*.
    - A reference to the queue entry.
  - Set *pphContext* to the **QueueContextHandle** value.

3.1.4.3 R_CloseQueue (Opnum 3)

The **R_CloseQueue** method closes a **QueueContextHandle (section 2.2.4)** that was previously opened by using a call to the **R_OpenQueue (section 3.1.4.2)** method or the **R_OpenQueueForMove (section 3.1.4.11)** method.

```
HRESULT R_CloseQueue(
  [in] handle_t hBind,
  [in, out] QueueContextHandle* pphContext
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

Copyright © 2008 Microsoft Corporation.

pphContext: MUST be set by the client to the **QueueContextHandle** to be closed. The handle MUST have been returned by the server in the *pphContext* of a prior call to **R_OpenQueue** or **R_OpenQueueForMove**, and MUST NOT have been closed through a prior call to **R_CloseQueue**. This value MUST NOT be NULL. The server MUST set this value to NULL.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in[MS-RPCE].

When processing this call, the server MUST:

- Look up *phContext* in the **QueueContextHandle** table.
- If found:
  - Find all entries in the pending request table that contain the **QueueContextHandle** and cancel the operations, as specified in section 3.1.4.8.
  - Find all entries in the cursor table that are associated with the **QueueContextHandle** and close them, as specified in section 3.1.4.5.
  - If the queue state in the associated queue entry is Exclusive Open:
    - Transition the queue state to Closed.
  - Else:
    - Decrement the shared open count.
    - If the shared open count is zero, transition the queue state to Closed.
  - Remove the entry from the **QueueContextHandle** table.
  - Set *pphContext* to NULL.
- Else return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.4 R_CreateCursor (Opnum 4)

The **R_CreateCursor** method creates a cursor and returns a handle to it. The handle can be used in subsequent calls to **R_StartReceive (section 3.1.4.7)** or **R_StartTransactionalReceive (section 3.1.4.13)** to specify a relative location in the queue from which to receive a message.

```
HRESULT R_CreateCursor(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [out] DWORD* phCursor
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

Copyright © 2008 Microsoft Corporation.

phContext: MUST be set by the client to the **QueueContextHandle (section 2.2.4)** with which to associate the cursor. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

phCursor: MUST be set by the server to a handle for the created cursor.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure, and the client MUST treat all failure **HRESULTs** identically.

The client MUST disregard all out-parameter values when any failure **HRESULT** is returned.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up *phContext* in the **QueueContextHandle** table.
- If found:
  - Create a new cursor handle. The handle MUST be unique across all cursor handles associated with the **QueueContextHandle**.
  - Set *phCursor* to the value of the created cursor handle.
  - Create a new entry in the cursor table keyed by the {*phCursor*, *phContext*} key pair.
  - Set the cursor position to the head of the queue.
  - Set the cursor state to Uninitialized, as specified in section 3.1.1.2.
- Else return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.5 R_CloseCursor (Opnum 5)

The **R_CloseCursor** method closes the handle for a previously created cursor. The client MUST call this method to reclaim resources on the server allocated by the **R_CreateCursor (section 3.1.4.4)** method.

```
HRESULT R_CloseCursor(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in] DWORD hCursor
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

phContext: MUST be set by the client to the **QueueContextHandle (section 2.2.4)** with which the cursor was associated in a call to **R_CreateCursor**. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue**

Copyright © 2008 Microsoft Corporation.

(section 3.1.4.2), and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

hCursor: MUST be set by the client to the handle of the cursor to be closed. The handle MUST have been obtained by a prior call to the **R_CreateCursor** method and MUST NOT have been closed through a prior call to **R_CloseCursor**.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown:

No exceptions are thrown except those that are thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up the {*phCursor*, *phContext*} key pair in the cursor table.
- If found:
  - For each entry in the pending request table that contains the cursor handle value matching *phCursor* and **QueueContextHandle** matching *phContext*, cancel the operation as specified in section 3.1.4.8.
  - Remove the entry from the cursor table.
- Else return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.6 R_PurgeQueue (Opnum 6)

The **R_PurgeQueue** method removes all messages from the queue.

```
HRESULT R_PurgeQueue(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

phContext: MUST be set by the client to a **QueueContextHandle (section 2.2.4)** of the queue to be purged. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** with the *dwAccess* parameter set to RECEIVE_ACCESS, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULT** identically.

Exceptions Thrown:

Copyright © 2008 Microsoft Corporation.

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up *phContext* in the **QueueContextHandle** table.
- If found and the access mode in the entry is RECEIVE_ACCESS:
  - Remove all messages from the message list associated with the queue. If the message has a non-NULL transaction identifier associated with it, or if its locked flag is set to TRUE, it MUST not be removed.
- Else return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.7 R_StartReceive (Opnum 7)

The **R_StartReceive** method peeks or receives a message from an open queue.

If **R_StartReceive** is invoked with a Peek action type, as specified in the *ulAction* parameter, the operation completes when **R_StartReceive** returns.

If **R_StartReceive** is invoked with a Receive action type, as specified in the *ulAction* parameter, the client MUST pair each call to **R_StartReceive** with a call to **R_EndReceive (section 3.1.4.9)** to complete the operation or to **R_CancelReceive (section 3.1.4.8)** to cancel the operation. The call to **R_EndReceive** or **R_CancelReceive** is correlated to a call to **R_StartReceive** through matching *dwRequestId* parameters.

If the client specifies a nonzero *ulTimeout* parameter, and a message is not available in the queue at the time of the call, the server waits up to the specified time-out for a message to become available in the queue before responding to the call. The client can call **R_CancelReceive** with a matching *dwRequestId* to cancel the pending **R_StartReceive** request.

The message to be returned can be specified in one of three ways:

- *LookupId*: A nonzero *LookupId* value specifies the unique identifier for the message to be returned. The *ulAction* parameter further specifies whether the message to be returned is the one identified by *LookupId*, or the first unlocked message immediately preceding or following it. For more details, see the description of the *ulAction* parameter.
- Cursor: A nonzero cursor handle specifies the cursor to be used to identify the message to be returned. The cursor specifies a location in the queue. The *ulAction* parameter further specifies whether the message to be returned is the one identified by the cursor, or the first unlocked message immediately following it. For more details, see the description of the *ulAction* parameter.
- First or Last: if *LookupId* is set to zero and *hCursor* is set to zero, the first or last unlocked message in the queue can be returned. The *ulAction* parameter further specifies whether the first message or last message is to be returned.

The *ppPacketSections* field is the address of one or more pointers to one or more **SectionBuffer (section 2.2.6)** structures. The **pSectionBuffer** field of the first **SectionBuffer** points to the beginning of the message packet. If more than one **SectionBuffer** structure is present, then the packet sections should be concatenated, in the order in which they appear in the array, to form the entire packet. The size of each section is stored in the **SectionSizeAlloc** field of the **SectionBuffer** structure.

Copyright © 2008 Microsoft Corporation.

```
HRESULT R_StartReceive(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in] ULONGLONG LookupId,
  [in] DWORD hCursor,
  [in] DWORD ulAction,
  [in] DWORD ulTimeout,
  [in] DWORD dwRequestId,
  [in] DWORD dwMaxBodySize,
  [in] DWORD dwReserved,
  [out] DWORD* pdwArriveTime,
  [out] ULONGLONG* pSequenceId,
  [out] DWORD* pdwNumberOfSections,
  [out, size_is(, *pdwNumberOfSections)]
    SectionBuffer** ppPacketSections
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

phContext: MUST be set by the client to a **QueueContextHandle (section 2.2.4)** of the queue from which to read a message. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and MUST NOT have been closed through a call prior to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

The handle MUST have been opened with a *dwAccess* value that permits the operation specified by the *ulAction* parameter. For more details, see the *dwAccess* parameter in **R_OpenQueue**.

LookupId: If nonzero, specifies the lookup identifier of the message to be acted on.

If the client sets the *LookupId* to nonzero, the parameter *ulTimeout* MUST be set to 0x00000000, parameter *hCursor* MUST be set to zero and parameter *ulAction* MUST be set to one of the following:

- MQ_LOOKUP_PEEK_PREV
- MQ_LOOKUP_PEEK_CURRENT
- MQ_LOOKUP_PEEK_NEXT
- MQ_LOOKUP_RECEIVE_PREV
- MQ_LOOKUP_RECEIVE_CURRENT
- MQ_LOOKUP_RECEIVE_NEXT

If the *LookupId* is 0x00000000, the *ulAction* parameter MUST NOT be set to any of the above values.

hCursor: If nonzero, specifies a handle to a cursor that MUST have been obtained from a prior call to the **R_CreateCursor (section 3.1.4.4)** method. The handle MUST NOT have been closed through a prior call to **R_CloseCursor (section 3.1.4.5)**.

If the client sets this parameter to nonzero, the parameter *LookupId* MUST be 0, and the *ulAction* parameter MUST be set to one of the following:

Copyright © 2008 Microsoft Corporation.

- MQ_ACTION_RECEIVE
- MQ_ACTION_PEEK_CURRENT
- MQ_ACTION_PEEK_NEXT

ulAction: Specifies the action to perform. The following table lists possible actions.

Value / Type	Meaning
MQ_ACTION_RECEIVE 0x00000000	If *hCursor* is nonzero, read and remove the message for the current cursor location and advance the cursor to the next position. If *hCursor* is 0, read and remove the message from the front of the queue. The *LookupId* parameter MUST be set to 0.
MQ_ACTION_PEEK_CURRENT 0x80000000	If *hCursor* is nonzero, read the message at the current cursor location, but do not remove it from the queue. If *hCursor* is 0, read the message at the front of the queue but do not remove it from the queue. The *LookupId* parameter MUST be set to 0.
MQ_ACTION_PEEK_NEXT 0x80000001	Read the message following the message at the current cursor location but do not remove it. The *LookupId* parameter MUST be set to 0. The *hCursor* parameter MUST be set to a nonzero cursor handle obtained from the **R_CreateCursor** method.
MQ_LOOKUP_PEEK_CURRENT 0x40000010	Read the message specified by the *LookupId* parameter but do not remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_NEXT 0x40000011	Read the message following the message specified by *LookupId* but do not remove it. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_PREV 0x40000012	Read the message preceding the message specified by the *LookupId* parameter but do not remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_FIRST 0x40000014	Read the first message in the queue but do not remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_LAST	Read the last message in the queue but do not remove it from

Copyright © 2008 Microsoft Corporation.

Value / Type	Meaning
0x40000018	the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Read the message specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_NEXT 0x40000021	Read the message following the message specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_PREV 0x40000022	Read the message preceding the message specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_FIRST 0x40000024	Read the first message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_LAST 0x40000028	Read the last message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.

If *hCursor* is 0 and *LookupId* is 0, *ulAction* MUST be set to one of the following:

- MQ_ACTION_RECEIVE
- MQ_ACTION_PEEK_CURRENT
- MQ_LOOKUP_PEEK_FIRST
- MQ_LOOKUP_PEEK_LAST
- MQ_LOOKUP_RECEIVE_FIRST
- MQ_LOOKUP_RECEIVE_LAST

ulTimeout: Specifies the timeout, in milliseconds, to wait for a message to become available in the queue. The client MUST set this parameter to 0x00000000 if the *LookupId* parameter

Copyright © 2008 Microsoft Corporation.

value is not 0. The client MUST set this to 0x00000000 if the action is not one of MQ_ACTION_RECEIVE, MQ_ACTION_PEEK_CURRENT, or MQ_ACTION_PEEK_NEXT.

If a message is not available when the server processes the request, the server MUST NOT return until any one of the following occurs.

- A message becomes available, as specified in section 3.1.6.3.
- The specified time-out elapses, as specified in section 3.1.5.
- The *hCursor* handle is closed, as specified in section 3.1.4.5.
- The phContext is closed, as specified in section 3.1.4.3.
- A call to **R_CancelReceive** that corresponds to this request is received.

dwRequestId: MUST be set by the client to a unique correlation identifier for the receive request. This value MUST be used in a subsequent call to **R_EndReceive** or **R_CancelReceive** to correlate that call with the call to **R_StartReceive**. The value MUST NOT be used in another **R_StartReceive** call on the same **QueueContextHandle** until a call to either **R_EndReceive** or **R_CancelReceive** with the same *dwRequestId* value has been completed.

dwMaxBodySize: MUST be set by the client to the maximum size, in bytes, of the message body to be returned.

dwReserved: The client MUST set this parameter to 0x00000000. The server SHOULD ignore it.

pdwArriveTime: The server MUST set this value to the time the message was added to the queue, expressed as the number of seconds elapsed since midnight 00:00:00.0, January 1, 1970 Coordinated Universal Time (UTC), as specified in section 3.1.6.3.

pSequenceId: If the message packet contains a TransactionHeader, as specified in [MS-MQQB] section 2, the server MUST set this value to the value of the **SequenceID** field of the TransactionHeader. If the message packet does not contain a TransactionHeader, the server MUST set this value to 0x00000000.

pdwNumberOfSections: The server MUST set this parameter to the number of elements in the array pointed to by the *ppPacketSections* parameter.

ppPacketSections: The server MUST set this parameter to an array of pointers to **SectionBuffer** structures. The server MUST fill this array in the following manner.

- If the value of the *dwMaxBodySize* parameter is greater than or equal to the message packet body size, the *ppPacketSections* MUST contain a single element as follows:
  - **SectionType (section 2.2.7)** MUST be set to stFullPacket (0x00000000).
  - The **SectionSize** and **SectionSizeAlloc** elements MUST be set to the message packet size.
  - **pSectionBuffer** MUST contain the entire message packet.
- If the value of the *dwMaxBodySize* parameter is less than the message packet body size, the array MUST contain a first element as follows:

Copyright © 2008 Microsoft Corporation.

- **SectionType** MUST be set to stBinaryFirstSection if the message packet is a binary packet or stSrmpFirstSection if the message packet is an SRMP packet.
- **pSectionBuffer** MUST contain the message packet concatenated with the first *dwMaxBodySize* bytes of the message body.
- **SectionSizeAlloc** MUST be set to the **message packet headers** size plus the message packet body size.
- **SectionSize** MUST be set to the size of the **pSectionBuffer**.

- If the value of the *dwMaxBodySize* parameter is less than the message packet body size and the message packet are not empty, the array MUST contain a second element as follows:
  - **SectionType** MUST be set to stBinarySecondSection if the message packet is a binary packet and stSrmpSecondSection if the message packet is an SRMP packet.
  - **pSectionBuffer** MUST contain the **message packet trailers**.
  - **SectionSize** and **SectionSizeAlloc** MUST be equal and set to the message packet trailers size.

Return Values: If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULT identically.

The client MUST disregard all out-parameter values when any failure HRESULT is returned.

MQ_OK (0x00000000)
MQ_ERROR_INVALID_HANDLE (0xC00E0007)
MQ_ERROR_IO_TIMEOUT (0xC00E001B)
MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088)

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

While processing this method, the server MUST:

- Look up *phContext* in the QueueContextHandle table.
- If found:
  - Find the message in the queue message list as specified in the *ulAction* parameter description.
    - If the *ulAction* parameter is MQ_LOOKUP_PEEK_CURRENT or MQ_LOOKUP_RECEIVE_CURRENT, the message is specified by the *LookupId* parameter.
    - If the *ulAction* parameter is MQ_LOOKUP_PEEK_PREV or MQ_LOOKUP_RECEIVE_PREV, the message is the first available message found starting immediately preceding the message specified by the *LookupId* parameter and walking toward the front of the queue.
    - If the *ulAction* parameter is MQ_LOOKUP_PEEK_NEXT or MQ_LOOKUP_RECEIVE_NEXT, the message is first available message found starting immediately following the message specified by the *LookupId* parameter and walking toward the end of the queue.

Copyright © 2008 Microsoft Corporation.

- If the *hCursor* parameter is nonzero and the *ulAction* parameter is MQ_ACTION_RECEIVE or MQ_ACTION_PEEK_CURRENT, the message is at the current cursor location.
- If the *ulAction* parameter is MQ_PEEK_NEXT, the message is the first available message found starting immediately following the message at the current cursor location and walking toward the end of the queue.
- If the *hCursor* parameter is 0 and the *ulAction* parameter is MQ_ACTION_RECEIVE or MQ_ACTION_PEEK_CURRENT, the message is the first available message found starting at the front of the queue and walking towards the end of the queue.
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_FIRST or MQ_LOOKUP_RECEIVE_FIRST, the message is the first available message found starting at the front of the queue and walking towards the end of the queue.
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_LAST or MQ_LOOKUP_RECEIVE_LAST, the message is the first available message found starting at the end of the queue and working towards the front of the queue.

- If the message is not found:
  - If the *ulTimeout* parameter is 0x00000000:
    - Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0008).
  - If the *ulTimeout* parameter is not 0x00000000:
    - Add an entry to the pending request table with:
      - If the *ulAction* type, as defined in the table under the *ulAction* parameter, is Receive, the entry type set to ReceiveWaitForMessage; otherwise the entry type set to PeekWaitForMessage.
      - The request ID set to *dwRequestId*.
      - The QueueContextHandle set to *phContext*.
      - Request time-out set to *ulTimeout*.
      - *LookupId* set to 0.
      - The cursor handle set to *hCursor*.
    - Create a call timer that waits for the message for the *ulTimeout* milliseconds.
    - If the message is not found before the time-out expires:
      - Return MQ_ERROR_IO_TIMEOUT (0xC00E001B).
- Fill the *ppPacketSections* array as specified in the *ppPacketSections* parameter description.
- If the *ulAction* type, as defined in the table under the *ulAction* parameter, is Receive, the server MUST do the following:
  - Set the message locked flag to TRUE, indicating that the message is locked by the pending request.
  - Add an entry to the pending request table with:

Copyright © 2008 Microsoft Corporation.

- The type set to ReceiveWaitForEndOrCancel.
- The request ID set to *dwRequestId*.
- The QueueContextHandle set to *phContext*.
- The request time-out set to 0x00000000.
- The *LookupId* set to the *LookupId* of the message that was found.
- The cursor handle set to *hCursor*.

- Else (*phContext* is not found in the QueueContextHandle table) return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.8 R_CancelReceive (Opnum 8)

The **R_CancelReceive** method cancels a pending call to **R_StartReceive (section 3.1.4.7)** or **R_StartTransactionalReceive (section 3.1.4.13)**. Each of those methods takes a time-out parameter that can cause the server to not return a response until a message becomes available or the time-out expires. The **R_CancelReceive** (section 3.1.4.8) method provides a way for the client to cancel a blocked request.

```
HRESULT R_CancelReceive(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in] DWORD dwRequestId
);
```

hBind: MUST be an RPC binding handle parameter as described in [MS-RPCE] section 2.

phContext: MUST be set by the client to the **QueueContextHandle (section 2.2.4)** used in the corresponding call to **R_StartReceive** that is to be canceled. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and MUST NOT have been previously closed through a call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

dwRequestId: MUST be set by the client to the same value as the *dwRequestId* parameter in the corresponding call to **R_StartReceive** or **R_StartTransactionalReceive**. This parameter acts as an identifier to correlate an **R_CancelReceive** call to an **R_StartReceive** or an **R_StartTransactionalReceive** call.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up the {*phContext*, *dwRequestId*} key pair in the pending requests table.

Copyright © 2008 Microsoft Corporation.

- If the request type is ReceiveWaitForMessage or PeekWaitForMessage:
  - If a call timer is associated with the call, cancel the timer.
  - Respond to the pending **R_StartReceive** or **R_StartTransactionalReceive** request with MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).
  - Remove the entry from the pending requests table.
- Else return MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

3.1.4.9 R_EndReceive (Opnum 9)

The client MUST invoke the **R_EndReceive** method to advise the server that the message packet returned by the **R_StartReceive (section 3.1.4.7)** method has been received.

The combination of the **R_StartReceive** method and the positive acknowledgment of the **R_EndReceive** (section 3.1.4.9) method ensures that a message packet is not lost in transit from the server to the client due to a network outage during the call sequence.

Note that a call to **R_StartTransactionalReceive (section 3.1.4.13)** is ended through a corresponding call to **R_EndTransactionalReceive (section 3.1.4.15)**, not through a call to this method.

```
HRESULT R_EndReceive(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in, range(1,2)] DWORD dwAck,
  [in] DWORD dwRequestId
);
```

hBind: MUST be an RPC binding handle parameter for use by the server, as described in [MS-RPCE] section 2.

phContext: MUST be set by the client to the QueueContextHandle used in the corresponding call to **R_StartReceive**. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

dwAck: MUST be set to an Acknowledgment (ACK) or a Negative Acknowledgment (NACK) for the message packet received from the server in an **R_StartReceive** request. The following table lists possible values.

Value	Meaning
RR_ACK 0x00000002	The client acknowledges that the message packet was delivered successfully. The server MUST remove the message from the queue and make it unavailable for subsequent consumption.
RR_NACK 0x00000001	The client acknowledges that the message packet was not delivered successfully. The server MUST keep the message in the queue and make it available for subsequent consumption.

Copyright © 2008 Microsoft Corporation.

dwRequestId: MUST be set by the client to the same value as the *dwRequestId* parameter in the corresponding call to **R_StartReceive**. This parameter acts as an identifier to correlate an **R_EndReceive** call to an **R_StartReceive** call.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol [MS-RPCE].

When processing this call, the server MUST:

- Look up the {*phContext*, *dwRequestId*} key pair in the pending requests table.
- If found and the type is ReceiveWaitForEndOrCancel:
  - Find the message identified by *LookupId* from the pending request entry.
  - If the supplied *dwAck* parameter value is RR_ACK (0x00000002), remove the message from the message list.
  - If the supplied *dwAck* parameter value is RR_NACK (0x00000001), set the message locked flag to FALSE, indicating that the message is no longer locked by the pending request.
  - Remove the entry from the pending request table.
- Else return MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

3.1.4.10 R_MoveMessage (Opnum 10)

The **R_MoveMessage** method moves a message from one queue to another.<23> The source and destination queues MUST be related as follows:

- The source is a queue and the destination is a subqueue of the source queue, or
- The destination is a queue and the source is a subqueue of the destination queue, or
- The source and destination are two subqueues of the same queue.

```
HRESULT R_MoveMessage(
  [in] handle_t hBind,
  [in] QueueContextHandle phContextFrom,
  [in] ULONGLONG ullContextTo,
  [in] ULONGLONG LookupId,
  [in] GUID* pTransactionId
);
```

hBind: MUST be an RPC binding handle parameter, as described in [MS-RPCE] section 2.

phContextFrom: MUST be set by the client to a **QueueContextHandle (section 2.2.4)** representing the source queue. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** with the *dwAccess* parameter set to RECEIVE_ACCESS, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

Copyright © 2008 Microsoft Corporation.

ullContextTo: MUST be set by the client to a **QueueContextHandle** representing the destination queue. The handle MUST have been returned by the server in the *pMoveContext* output parameter of a prior call to **R_OpenQueueForMove (section 3.1.4.11)**, and MUST NOT have been closed through a prior call to **R_CloseQueue**. This value MUST NOT be NULL.

LookupId: MUST be set by the client to the lookup identifier of the message to be moved.

pTransactionId: MUST be set by the client to a transaction identifier. The transaction identifier MUST have been registered with the server through a prior call to the **R_QMEnlistRemoteTransaction (section 3.1.4.12)** method. It MUST NOT be NULL.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure, and the client **HRESULT** treat all failure **HRESULTs** identically.

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

The **R_MoveMessage** method is a transactional operation and requires a valid transaction identifier. This method provisionally moves a message from the source queue to the destination queue, pending notification of the transaction outcome. See section 3.1.6.

When processing this call, the server MUST:

- Look up *phContextFrom* in the **QueueContextHandle** table.
- If found:
  - Look up *ullContextTo* in the **QueueContextHandle** table.
  - If found:
    - If the queue entry identified by the *phContextFrom* parameter or the queue entry identified by the *ullContextTo* parameter has the transactional flag set to FALSE, return MQ_ERROR_TRANSACTION_USAGE (0xC00E0050).
    - Look up *LookupId* in the message list of the queue identified by the *phContextFrom* parameter.
    - If found:
      - Set the message list entry transaction identifier to *pTransactionId*.
      - Add an entry to the request list associated with the transaction identifier, containing the **R_MoveMessage** type, the *LookupId* of the message, *phContextFrom*, and *ullContextTo*.
    - Else (*LookupId* is not found in the message list):
      - Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
  - Else (*ullContextTo* is not found in the **QueueContextHandle** table):
    - Return MQ_ERROR_INVALID_HANDLE (0xC00E0007).
- Else (*phContextFrom* in the **QueueContextHandle** table is not found):

Copyright © 2008 Microsoft Corporation.

- Return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.11 R_OpenQueueForMove (Opnum 11)

The **R_OpenQueueForMove** method opens the queue and returns a **QueueContextHandle (section 2.2.4)** that can subsequently be used as the *ullContextTo* (destination queue) parameter of a call to **R_MoveMessage (section 3.1.4.10)**. This method MUST be called before the **R_MoveMessage** operation.<24>

```
void R_OpenQueueForMove(
  [in] handle_t hBind,
  [in] struct QUEUE_FORMAT* pQueueFormat,
  [in] DWORD dwAccess,
  [in] DWORD dwShareMode,
  [in] GUID* pClientId,
  [in] LONG fNonRoutingServer,
  [in] unsigned char Major,
  [in] unsigned char Minor,
  [in] USHORT BuildNumber,
  [in] LONG fWorkgroup,
  [out] ULONGLONG* pMoveContext,
  [out] QueueContextHandle* pphContext
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

pQueueFormat: MUST be a pointer to a **QUEUE_FORMAT** structure that identifies the queue to open. This value MUST NOT be NULL. The value of the **m_qft** field MUST be one of QUEUE_FORMAT_TYPE_PUBLIC, QUEUE_FORMAT_TYPE_PRIVATE, QUEUE_FORMAT_TYPE_DIRECT, QUEUE_FORMAT_TYPE_MACHINE, or QUEUE_FORMAT_TYPE_SUBQUEUE.

dwAccess: Specifies the required type of access to the queue. MUST be set by the client to MQ_MOVE_ACCESS (0x00000004).

dwShareMode: Specifies whether the client needs exclusive access to the queue. MUST be set by the client to one of the following values.

Value	Meaning
DENY_NONE 0x00000000	Permits multiple **QueueContextHandles** to the queue to be opened concurrently.
DENY_SHARE 0x00000001	Permits a single open **QueueContextHandle** to the queue at a time, providing exclusive access.

pClientId: MUST be set by the client to a pointer to a valid globally unique identifier (GUID) that uniquely identifies the client.<25> The server SHOULD ignore this parameter. The server MAY use this parameter to impose a limit on the number of unique callers.<26> This value MUST NOT be NULL.

fNonRoutingServer: If the client is configured to operate in the role of a routing server, as specified in [MS-MQMA], this parameter MUST be set to FALSE (0x00000000); otherwise, it MUST be set to TRUE (0x00000001).<27> If the value of the *fNonRoutingServer* parameter is FALSE (0x00000000), the server MUST ignore *pClientId*.

Copyright © 2008 Microsoft Corporation.

Name	Value
FALSE	0x00000000
TRUE	0x00000001

Major: MUST be set by the client to an implementation-specific Major Version number of the client. SHOULD be ignored by the server.<28>

Minor: MUST be set by the client to an implementation-specific Minor Version number of the client. SHOULD be ignored by the server.<29>

BuildNumber: MUST be set by the client to an implementation-specific Build Number of the client. SHOULD be ignored by the server.<30>

fWorkgroup: MUST be set to TRUE (0x00000001) by the client if the client machine is not a member of a Windows domain; otherwise, it MUST be set to FALSE (0x00000000). The RPC authentication level required by the server MAY be based on this value in subsequent calls on the interface.<31>

Name	Value
FALSE	0x00000000
TRUE	0x00000001

pMoveContext: The server MUST set this parameter to a pointer to a **QueueContextHandle** and MUST set the value of this parameter to the same value as the contents of *pphContext*. The server MUST set this value to a context that can be used as the *dwContextTo* in a subsequent call to the **R_MoveMessage** method. Logically, it represents a reference to the **QueueContextHandle** returned in *pphContext*.

pphContext: MUST be set by the server to a **QueueContextHandle**. A **QueueContextHandle** opened through a call to this method can be closed through a subsequent call to **R_CloseQueue (section 3.1.4.3)**.

Return Values: The method has no return values. If the method fails, an RPC exception is thrown.

Exceptions Thrown:

In addition to the exceptions thrown by the underlying RPC protocol [MS-RPCE], the method throws **HRESULT** failure codes as RPC exceptions. The client MUST treat all thrown **HRESULT** codes identically.

The client MUST disregard all out-parameter values when any failure **HRESULT** is thrown.

When processing this call, the server MUST do the following:

- Look up the queue name in the queue table. If not found, throw MQ_ERROR_QUEUE_NOT_FOUND (0xC00E0003).
- Using the current state of the queue, process the request according to the following queue state table.

Copyright © 2008 Microsoft Corporation.

Current queue state	Requested share mode	Result	Effect on queue state
Closed	DENY_SHARE	Success	Transition to Exclusive Open state.
Closed	DENY_NONE	Success	Transition to Share Open state.
Exclusive Open	DENY_NONE or DENY_SHARE	Fail	No change.
Share Open	DENY_SHARE	Fail	No change.
Share Open	DENY_NONE	Success	Remain in Share Open state and Increment Shared Open Count.

- If the Result from the table above is Fail, throw MQ_ERROR_SHARING_VIOLATION (0xC00E0009).
- If the Result from the table above is Success:
  - Adjust the queue entry state and share count as defined in the table above.
  - Create a **QueueContextHandle**.
  - Add a new **QueueContextHandle** table entry with:
    - **QueueContextHandle** set to the created **QueueContextHandle**.
    - Access mode set to *dwAccess*.
    - A reference to the queue entry.
- Set *pphContext* to the **QueueContextHandle** value.

3.1.4.12 R_QMEnlistRemoteTransaction (Opnum 12)

The **R_QMEnlistRemoteTransaction** method propagates a distributed atomic transaction context to the server. The server MUST enlist in the transaction context. The client MUST call this method prior to the **R_StartTransactionalReceive (section 3.1.4.13)** or the **R_MoveMessage (section 3.1.4.10)** calls.<32> Subsequent calls to **R_StartTransactionalReceive** and **R_MoveMessage** that use the same transaction identifier are coordinated such that they either all occur or none of them occurs, depending on whether the transaction outcome is Commit or Rollback.

```
HRESULT R_QMEnlistRemoteTransaction(
  [in] handle_t hBind,
  [in] GUID* pTransactionId,
  [in, range(0, 131072)] DWORD cbPropagationToken,
  [in, size_is (cbPropagationToken)]
    unsigned char* pbPropagationToken,
  [in] struct QUEUE_FORMAT* pQueueFormat
);
```

hBind: MUST be an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

pTransactionId: MUST be a pointer to a transaction identifier obtained, as specified in [MS-DTCO] section 3.3.4.1. This value MUST NOT be NULL.

Copyright © 2008 Microsoft Corporation.

cbPropagationToken: MUST be the size, in bytes, of *pbPropagationToken*.

pbPropagationToken: MUST be a transaction propagation token, as specified in [MS-DTCO] section 2.2.5.4, that represents the transaction identified by the *pTransactionId* parameter. This parameter MUST NOT be NULL.

pQueueFormat: MUST be a pointer to a **QUEUE_FORMAT structure** that identifies the queue which will be passed to the **R_StartTransactionalReceive** method. SHOULD be ignored by the server.<33>

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

MQ_OK (0x00000000)

Exceptions Thrown:

No exceptions are thrown except those that are thrown by the underlying RPC protocol, as specified in [MS-RPCE].

While processing this operation, the server MUST:

- Look up the *pTransactionId* in the transactional request table. If not found, add an entry to the table with an empty request list.
- Enlist into the transaction as specified in [MS-DTCO] section 3.5.4.3.

3.1.4.13 R_StartTransactionalReceive (Opnum 13)

The **R_StartTransactionalReceive** method transactionally receives a message from the opened queue.<34>

Unlike **R_StartReceive (section 3.1.4.7)**, **R_StartTransactionalReceive** MUST be specified with an action type of Receive. A call to **R_StartTransactionalReceive** MUST be paired with a subsequent call to **R_EndTransactionalReceive (section 3.1.4.15)** or **R_CancelReceive (section 3.1.4.8)** to complete the operation. The call to **R_EndTransactionalReceive** or **R_CancelReceive** is correlated to a call to **R_StartTransactionalReceive** through matching *dwRequestId* parameters.

If the client specifies a nonzero *ulTimeout* parameter and a message is not available in the queue at the time of the call, the server waits up to the specified time-out for a message to become available in the queue before responding to the call. The client can call **R_CancelReceive** with matching *dwRequestId* to cancel the pending **R_StartTransactionalReceive** request.

The message to be returned can be specified in one of three ways.

- *LookupId*: A nonzero *LookupId* value that specifies the unique identifier for the message to be returned. The *ulAction* parameter further specifies whether the message to be returned is the one identified by *LookupId*, or the first unlocked message immediately preceding or following it. For more details, see the description of the *ulAction* parameter.
- Cursor: A nonzero cursor handle that specifies the cursor to be used to identify the message to be returned. The cursor specifies a location in the queue. The *ulAction* parameter further specifies whether the message to be returned is the one identified by the cursor, or the first unlocked

Copyright © 2008 Microsoft Corporation.

message immediately following it. For more details, see the description of the *ulAction* parameter.

- First or Last: If *LookupId* is set to 0 and *hCursor* is set to 0, the first or last unlocked message in the queue can be returned. The *ulAction* parameter further specifies whether the first message or last message is to be returned. For more details, see the description of the *ulAction* parameter.

The *ppPacketSections* parameter is the address of one or more pointers to one or more **SectionBuffer (section 2.2.6)** structures. The **pSectionBuffer** field of the first **SectionBuffer** points to the beginning of the message packet. If more than one **SectionBuffer** structure is present, the packet sections should be concatenated, in the order in which they appear in the array, to form the entire packet. The size of each section is stored in the **SectionSizeAlloc** field of the **SectionBuffer** structure.

```
HRESULT R_StartTransactionalReceive(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in] ULONGLONG LookupId,
  [in] DWORD hCursor,
  [in] DWORD ulAction,
  [in] DWORD ulTimeout,
  [in] DWORD dwRequestId,
  [in] DWORD dwMaxBodySize,
  [in] DWORD dwReserved,
  [in] GUID* pTransactionId,
  [out] DWORD* pdwArriveTime,
  [out] ULONGLONG* pSequenceId,
  [out] DWORD* pdwNumberOfSections,
  [out, size_is(, *pdwNumberOfSections)]
    SectionBuffer** ppPacketSections
);
```

hBind: MUST be an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

phContext: MUST be set by the client to a **QueueContextHandle (section 2.2.4)** of the queue from which to read a message. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** with the *dwAccess* parameter set to RECEIVE_ACCESS, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

LookupId: If nonzero, specifies the lookup identifier of the message to be acted on.

If the client sets the *LookupId* to nonzero, the parameter *ulTimeout* MUST be set to 0x00000000, the *hCursor* parameter MUST be set to 0, and the *ulAction* parameter MUST be set to one of the following:

- MQ_LOOKUP_RECEIVE_PREV
- MQ_LOOKUP_RECEIVE_CURRENT
- MQ_LOOKUP_RECEIVE_NEXT

If the *LookupId* is 0x00000000, the *ulAction* parameter MUST NOT be set to any of the above values.

Copyright © 2008 Microsoft Corporation.

hCursor: If nonzero, specifies a handle to a cursor that MUST have been obtained from a prior call to the R_CreateCursor (section 3.1.4.4) method. The handle MUST NOT have been closed through a prior call to R_CloseCursor (section 3.1.4.5).

If the client sets this parameter to nonzero, the parameter *LookupId* MUST be 0, and the *ulAction* parameter MUST be set to MQ_ACTION_RECEIVE.

ulAction: Specifies a receive action to perform on the message. The following table lists possible actions.

Value	Meaning
MQ_ACTION_RECEIVE 0x00000000	If *hCursor* is nonzero, read and remove the message at the current cursor location from the queue and advance the cursor. If *hCursor* is 0, read and remove the message from the front of the queue. The *LookupId* parameter MUST be set to 0.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Read the message specified by *LookupId* and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_NEXT 0x40000021	Read the message following the message specified by *LookupId* and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_PREV 0x40000022	Read the message preceding the message specified by *LookupId* and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_FIRST 0x40000024	Read the first message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_LAST 0x40000028	Read the last message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to 0. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.

If *hCursor* is 0 and *LookupId* is 0, *ulAction* MUST be set to one of the following:

- MQ_ACTION_RECEIVE
- MQ_LOOKUP_RECEIVE_FIRST

Copyright © 2008 Microsoft Corporation.

- MQ_LOOKUP_RECEIVE_LAST

ulTimeout: Specifies the time-out, in milliseconds, to wait for a message to become available in the queue. The client MUST set this parameter to 0x00000000 if the *LookupId* parameter value is not 0. The client MUST set this parameter to 0x00000000 if the action is not MQ_ACTION_RECEIVE.

If a message is not available when the server processes the request, the server MUST NOT return until any one of the following occurs:

- A message becomes available, as specified in section 3.1.6.3.
- The specified time-out elapses, as specified in section 3.1.5.
- The *hCursor* handle is closed, as specified in section 3.1.4.5.
- The *phContext* is closed, as specified in section 3.1.4.3.
- A call to **R_CancelReceive** corresponding to this request is received.

dwRequestId: MUST be set by the client to a unique correlation identifier for the receive request. This value MUST be used in a subsequent call to **R_EndTransactionalReceive** or **R_CancelReceive** to correlate that call with the call to **R_StartTransactionalReceive**. The value MUST NOT be used in another **R_StartTransactionalReceive** call on the same **QueueContextHandle** until a call to either **R_EndTransactionalReceive** or **R_CancelReceive** with the same *dwRequestId* value has been completed.

dwMaxBodySize: MUST be set by the client to the maximum size, in bytes, of the message body to be returned.

dwReserved: MUST be set by the client to 0x00000000. The server SHOULD ignore it.

pTransactionId: MUST be set by the client to a transaction identifier. The transaction identifier MUST have been registered with the server through a prior call to the **R_QMEnlistRemoteTransaction (section 3.1.4.12)** method. It MUST NOT be NULL.

pdwArriveTime: MUST be set by the server to the time the message was added to the queue, expressed as the number of seconds elapsed since midnight 00:00:00.0, January 1, 1970 Coordinated Universal Time (UTC), as specified in section 3.1.6.3.

pSequenceId: If the message packet contains a TransactionHeader, as specified in [MS-MQQB] section 2, the server MUST set this parameter to the value of the **SequenceID** field of the TransactionHeader. If the message packet does not contain a TransactionHeader, the server MUST set this value to 0x00000000.

pdwNumberOfSections: MUST be set by the server to the number of elements in the array that are pointed to by the *ppPacketSections* parameter.

ppPacketSections: MUST be set by the server to an array of pointers to **SectionBuffer** (section 2.2.6) structures. The server MUST fill this array in the following manner:

- If the value of the *dwMaxBodySize* parameter is greater than or equal to the message packet body size, the *ppPacketSections* MUST contain a single element as follows:
  - **SectionType (section 2.2.7)** MUST be set to stFullPacket (0x00000000).
  - The **SectionSize** and **SectionSizeAlloc** MUST be set to the message packet size.

Copyright © 2008 Microsoft Corporation.

- **pSectionBuffer** MUST contain the entire message packet.

- If the value of the *dwMaxBodySize* parameter is less than the message packet body size, the array MUST contain a first element as follows:

  - **SectionType** MUST be set to stBinaryFirstSection if the message packet is a binary packet or stSrmpFirstSection if the message packet is an SRMP packet.

  - **pSectionBuffer** MUST contain the message packet headers concatenated with the first *dwMaxBodySize* bytes of the message body.

  - **SectionSizeAlloc** MUST be set to the message packet headers size plus the message packet body size.

  - **SectionSize** MUST be set to the size of the **pSectionBuffer**.

- If the value of the *dwMaxBodySize* parameter is less than the message packet body size and the message packet trailers are not empty, the array MUST contain a second element as follows:

  - **SectionType** MUST be set to one of the following:

    - stBinarySecondSection if the message packet is a binary packet.

    - stSrmpSecondSection if the message packet is an SRMP packet.

  - **pSectionBuffer** MUST contain the message packet trailers.

  - **SectionSize** and **SectionSizeAlloc** MUST be equal and set to the message packet trailers size.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULTs identically.

The client MUST disregard all out-parameter values when any failure HRESULT is returned.

MQ_OK (0x00000000)
MQ_ERROR_INVALID_HANDLE (0xC00E0007)
MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0008)
MQ_ERROR_IO_TIMEOUT (0xC00E001B)
MQ_ERROR_TRANSACTION_USAGE (0xC00E0050)

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

While processing this method, the server MUST:

- Look up the *phContext* in the **QueueContextHandle** table.

- If found:

  - If the queue entry transactional flag is FALSE, the queue does not support transactional operations. Return MQ_ERROR_TRANSACTION_USAGE (0xC00E0050).

Copyright © 2008 Microsoft Corporation.

- Find the message in the queue message list as specified in the *ulAction* parameter description:
  - If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_CURRENT, the message is specified by the *LookupId* parameter.
  - If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_PREV, the first unlocked message found starting immediately preceding the message specified by the *LookupId* parameter and walking toward the front of the queue.
  - If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_NEXT, the message is the message found succeeding the message specified by the *LookupId* parameter and walking toward the end of the queue.
  - If the *hCursor* parameter is nonzero and the *ulAction* parameter is MQ_ACTION_RECEIVE, the message is at the current cursor location.
  - If the *hCursor* parameter is 0 and the *ulAction* parameter is MQ_ACTION_RECEIVE, the message is the first unlocked message found starting at the front of the queue and walking toward the end of the queue.
  - If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_FIRST, the message is the first unlocked message found starting at the front of the queue and walking toward the end of the queue.
  - If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_LAST, the message is the first message found starting at the end of the queue and walking toward the front of the queue.
- If the message is not found:
  - If the *ulTimeout* parameter is 0x00000000:
    - Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0008).
  - If the *ulTimeout* parameter is not 0x00000000:
    - Create a timer that waits for the message for up to *ulTimeout* milliseconds.
    - Add an entry to the pending request table with:
      - The type set to ReceiveWaitForMessage.
      - The request ID set to *dwRequestId*.
      - The **QueueContextHandle** set to *phContext*.
      - The request time-out set to *ulTimeout*.
      - The *LookupId* set to 0.
      - The cursor handle set to *hCursor*.
    - If the message is not found before the time-out expires:
      - Return MQ_ERROR_IO_TIMEOUT (0xC00E001B).
- Fill the *ppPacketSections* array as specified in the ppPacket description.
- Set the message locked flag to TRUE, indicating that the message is locked by the pending request.

Copyright © 2008 Microsoft Corporation.

- Add an entry to the pending request table with:
  - The type set to ReceiveWaitForEndOrCancel.
  - The request ID set to *dwRequestId*.
  - The **QueueContextHandle** set to *phContext*.
  - The request time-out set to 0x00000000.
  - The *LookupId* set to the *LookupId* of the message that was found.
  - The cursor handle set to *hCursor*.
- Add an entry to the request list in the transactional request table keyed by *pTransactionId* with:
  - The operation type set to **R_StartTransactionalReceive**.
  - The *LookupId* set to the *LookupId* of the message that was found.
  - The **QueueContextHandle** set to *phContext*.

- Else (the *phContext* is not found in the QueueContextHandle table):
  - Return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.14 R_SetUserAcknowledgementClass (Opnum 14)

The **R_SetUserAcknowledgementClass** method sets the acknowledgment class property of a message in the queue. This allows marking the message as rejected.<35>

```
HRESULT R_SetUserAcknowledgementClass(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
  [in] ULONGLONG LookupId,
  [in] USHORT usClass
);
```

hBind: MUST be an RPC binding handle parameter, as described in [MS-RPCE] section 2.

phContext: MUST be set by the client to a **QueueContextHandle (section 2.2.4)** representing the queue containing the message on which to set the acknowledgment class. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** with *dwAccess* set to MQ_RECEIVE_ACCESS, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

LookupId: MUST be set by the client to the lookup identifier of the message on which to set the acknowledgment class.

usClass: The acknowledgment class to set. It MUST be set by the client to one of the following values:

Copyright © 2008 Microsoft Corporation.

Value	Meaning
0x0000	No-op. No change is made to the acknowledgment class.
MQMSG_CLASS_NACK_RECEIVE_REJECTED 0xC004	Marks the message as rejected.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULTs identically.

MQ_OK (0x00000000)
MQ_ERROR_INVALID_HANDLE (0xC00E0007)
MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088)

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST do the following:

- Look up *phContext* in the **QueueContextHandle** table.
- If found:
  - Look up *LookupId* in the message list of the queue table entry.
    - If found:
      - If *usClass* is not 0x0000, set the acknowledgment class of the message to the value of *usClass*.
    - Else:
      - Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
- Else (*phContext* not found in the **QueueContextHandle** table):
  - Return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.15 R_EndTransactionalReceive (Opnum 15)

The client MUST invoke the **R_EndTransactionalReceive** method to advise the server that the message packet returned by the **R_StartTransactionalReceive (section 3.1.4.13)** method has been received by the client.<36>

The combination of the **R_StartTransactionalReceive** method and the positive acknowledgment of the **R_EndTransactionalReceive** method ensures that a message packet is not lost in transit from the server to the client due to a network outage during the call sequence.

```
HRESULT R_EndTransactionalReceive(
  [in] handle_t hBind,
  [in] QueueContextHandle phContext,
```


Copyright © 2008 Microsoft Corporation.

```
  [in, range(1,2)] DWORD dwAck,
  [in] DWORD dwRequestId
);
```

hBind: MUST be an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

phContext: MUST be set by the client to the **QueueContextHandle (section 2.2.4)** used in the corresponding call to **R_StartTransactionalReceive**. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and MUST NOT have been closed through a prior call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.

dwAck: MUST be set to an Acknowledgment (ACK) or a Negative Acknowledgment (NACK) for the message packet received from the server in an **R_StartTransactionalReceive** request. The following table lists possible values.

Value	Meaning
RR_ACK 0x00000002	The client acknowledges that the message packet was delivered successfully. The server MUST remove the packet from the queue and make it unavailable for subsequent consumption.
RR_NACK 0x00000001	The client acknowledges that the message packet was not delivered successfully. The server MUST keep the message packet and make it available for subsequent consumption.

dwRequestId: MUST be set by the client to the same value as the *dwRequestId* parameter in the corresponding call to **R_StartTransactionalReceive**. This parameter acts as an identifier to correlate an **R_EndTransactionalReceive** call to an **R_StartTransactionalReceive** call.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULTs identically.

MQ_OK (0x00000000)
MQ_ERROR_OPERATION_CANCELLED (0xC00E0008)

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST do the following:

- Look up the {*phContext*, *dwRequestId*} key pair in the pending requests table.
- If found and the type is ReceiveWaitForEndOrCancel:
  - Find the message identified by *LookupId* from the pending request entry.
  - If the supplied *dwAck* parameter value is RR_NACK (0x00000001)
    - Remove the associated entry from the Transactional Request table.

Copyright © 2008 Microsoft Corporation.

- Set the message locked flag to FALSE, indicating that the message is not locked by a pending request.

- Else (the {*phContext*, *dwRequestId*} key pair is not found in the pending requests table):
  - Return MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

3.1.5 Timer Events

The server MUST maintain a timer associated with each **R_StartReceive (section 3.1.4.7)** or **R_StartTransactionalReceive (section 3.1.4.13)** request in which the *ulTimeout* parameter is nonzero. The length of time to wait is specified by the client in the *ulTimeout* parameter of the request. If the server is unable to satisfy the request within the specified time-out, it MUST return MQ_ERROR_IO_TIMEOUT (0xC00E001B).

3.1.5.1 Call Timer Expired

The server sets a per-call timer for each call to **R_StartReceive (section 3.1.4.7)** or **R_StartTransactionalReceive (section 3.1.4.13)** in which the *ulTimeout* parameter is nonzero. If the server is unable to satisfy the pending request and the timer expires, the server receives this event.

While processing this event, the server MUST:

- Look up the entry in the pending request table that is associated with the {*phContext*, *dwRequestId*} key pair that is passed as parameters to the **R_StartReceive** or **R_StartTransactionalReceive** method.
- Cancel the **R_StartReceive** or **R_StartTransactionalReceive** operation, as specified in section 3.1.4.8; return MQ_ERROR_IO_TIMEOUT (0xC00E001B) to the caller.
- Remove the entry from the pending request table.

3.1.6 Other Local Events

The following local events trigger operations on the server.

- Transaction Commit notification
- Transaction Rollback notification
- Message added to the queue
- Message deleted from the queue

3.1.6.1 Transaction Commit Notification

This event is received when the transaction coordinator notifies the server of the outcome of a **transaction** that the server has previously enlisted in. The event notification MUST specify the transaction identifier for the transaction being committed, as specified in [MS-DTCO] section 3.5.4.5.

While processing this event, the server MUST:

- Look up the transaction identifier in the transactional request table.
- If found:

Copyright © 2008 Microsoft Corporation.

- For each entry in the request list, the server MUST:
  - If the entry type is **R_MoveMessage**:
    - Remove the message identified by the *LookupId* from the message list associated with the source queue.
    - Add the message identified by the *LookupId* to the message list associated with the destination queue.
    - Unlock the message by setting the transaction identifier on the message to NULL.
  - If the entry type is **R_StartTransactionalReceive (section 3.1.4.13)**:
    - Remove the message identified by the *LookupId* from the message list associated with the queue.
  - Remove the entry from the transactional request table.
- Else:
  - Do nothing.

3.1.6.2 Transaction Rollback Notification

This event is received when the transaction coordinator notifies the server of the outcome of a transaction the server has previously enlisted in. The event notification MUST specify the transaction identifier for the transaction being rolled back, as specified in [MS-DTCO] section 3.5.4.4.

While processing this event, the server MUST:

- Look up the transaction identifier in the transactional request table.
- If found:
  - For each entry in the pending request list, the server MUST:
    - Unlock the message identified by the *LookupId* by setting the transaction identifier on the message to NULL.
  - Remove the entry from the transactional request table.
- Else:
  - Do nothing

3.1.6.3 Message Added to the Queue

This event is received when a new message is added to the end of the message list of a queue.

While processing this event, the server MUST:

Set the arrived time of the new message to the current time.

Find the first entry in the pending request list that has the type ReceiveWaitForMessage and that has a **QueueContextHandle (section 2.2.4)** associated with the queue to which the message was added.

Copyright © 2008 Microsoft Corporation.

- If found:
  - Complete the **R_StartReceive (section 3.1.4.7)** or **R_StartTransactionalReceive (section 3.1.4.13)** request associated with the *dwRequestId* by returning the message to the caller, as specified in section 3.1.4.7.
  - Change the entry type to ReceiveWaitForEndOrCancel.
- Else:
  - For each entry in the pending request list with the type PeekWaitForMessage:
    - Complete the **R_StartReceive** request with the *dwRequestId* by returning the message to the caller, as specified in section 3.1.4.7.
    - Remove the entry from the pending request list.
  - For each entry in the cursor table that has the cursor state End Queue:
    - Transition the state to Mid Queue.

3.1.6.4 Message Deleted from the Queue

The event is received when a message is deleted from the message list of a queue.

While processing this event, the server MUST:

- For each entry in the cursor table that has a cursor location that points to the deleted message:
  - Change the state to Message Deleted.

3.1.6.5 RPC Failure Event

The event is received when RPC detects a connection failure with a client identified by a specific QueueContextHandle.

While processing this event, the server MUST:

- For each entry in the PendingRequest table associated with the QueueContextHandle:
  - If a timer is associated with the request, cancel the timer.
  - If the type of the pending message is ReceiveWaitForEndOrCancel:
    - Find the message identified by LookupId from the pending request table.
    - Set the message locked flag to FALSE indicating that the message is no longer locked by the pending request.
  - Remove the entry from the pending request table.
- For each entry in the Cursor table associated with the QueueContextHandle:
  - Remove the entry from the cursor table.
- Remove the QueueContextHandle from the QueueContextHandle table.

Copyright © 2008 Microsoft Corporation.

3.2 RemoteRead Client Details

3.2.1 Abstract Data Model

Clients MUST maintain the following data elements:

- A **QueueContextHandle (section 2.2.4)** associated with a queue.
- A table of cursor handles associated with a **QueueContextHandle**.

3.2.2 Timers

No protocol timers are required except those that are used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE].

3.2.3 Initialization

The client MUST create an RPC connection to the remote computer by using the details specified in section 2.1.

3.2.4 Message Processing Events and Sequencing Rules

The operation of the protocol is initiated and subsequently driven by the following higher-layer triggered events.

- The message queuing application opens a queue.
- The message queuing application enlists in a transaction.
- The message queuing application Peeks or Receives a message.
- The message queuing application rejects a received message.
- The message queuing application cancels a pending Peek or Receive.
- The message queuing application moves a message between the queue and its subqueue or between two subqueues of the same queue.
- The message queuing application purges a queue.
- The message queuing application creates a cursor.
- The message queuing application uses the cursor to Peek or Receive messages.
- The message queuing application closes the cursor.
- The message queuing application closes the queue.

3.2.4.1 Opening a Queue

The message queuing application MUST supply a queue name, an access mode, and a share mode. Opening a queue consists of the following sequence of operations.

- The client MUST construct an RPC binding handle to the server, as specified in [C706-Ch2Intro] section 2.3.

Copyright © 2008 Microsoft Corporation.

- The client MAY<37> call the **R_GetServerPort (section 3.1.4.1)** method by using the RPC handle from the previous step. This method returns the RPC endpoint port on which subsequent method calls to this interface are to be invoked.
- The client MAY<38> construct a new RPC binding handle to the server by using the RPC endpoint port determined in the previous step and replace with it the initial RPC binding handle to the server.
- The client MUST call the **R_OpenQueue (section 3.1.4.2)** method and MUST specify the following parameter values:
  - The RPC binding handle constructed in previous steps.
  - *pQueueFormat* set to the queue format name.
  - *dwAccess* mode set to the access mode.
  - *dwShareMode* set to the share mode.
  - Other parameters are as specified in section 3.1.4.2.
- The client MUST record the returned **QueueContextHandle (section 2.2.4)**.

3.2.4.2 Enlisting in a Transaction

The message queuing application MUST specify the transaction identifier.

- The client MUST enlist the server in the transaction through a call to the **R_QMEnlistRemoteTransaction (section 3.1.4.12)** method with:
  - *pTransactionId* set to the transaction identifier.
  - A transaction propagation token, as specified in [MS-DTCO] section 2.2.5.4.

3.2.4.3 Peek a Message

The message queuing application MUST specify the QueueContextHandle (section 2.2.4) of the queue to be read from, the time-out parameter for the operation, a *LookupId*, a maximum message body size, and an action from the table in the description of the *ulAction* parameter in section **R_StartReceive (section 3.1.4.7)** with action type of Peek.

- The client MUST call the **R_StartReceive** (section 3.1.4.7) method and MUST specify the following parameter values:
  - *phContext* set to the **QueueContextHandle** (section 2.2.4).
  - *hCursor* set to NULL.
  - *LookupId* set to the value specified by the message queuing application.
  - *ulAction* set to the action specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - A *dwRequestId* value that uniquely identifies this call from all other pending calls to this protocol.

Copyright © 2008 Microsoft Corporation.

- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** received in ppPacketSections, as specified in section 3.1.4.7.
- The client MUST return the message to the message queuing application.

3.2.4.4 Receive a Message

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** of the queue to be read from, a transaction identifier, the time-out parameter for the operation, a LookupId, a maximum message body size, and an action from the table in the description of the *ulAction* parameter in **R_StartReceive (section 3.1.4.7)** with action type of Receive.

- If the transaction identifier specified by the message queuing application is NULL, follow the sequencing rules as specified in section 3.2.4.4.1.
- If the transaction identifier specified by the message queuing application is non-NULL, follow the sequencing rules as specified in section 3.2.4.4.2.

3.2.4.4.1 Receive a Message Without a Transaction

- The client MUST call the **R_StartReceive (section 3.1.4.7)** method and MUST specify the following parameter values.
  - *phContext* set to the QueueContextHandle.
  - *hCursor* set to NULL.
  - *ulAction* set to the value specified by the message queuing application.
  - *LookupId* set to the value specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - *dwRequestId* set to a value that uniquely identifies this call from all other pending calls to this protocol.
- The client MUST reconstruct the message from the **SectionBuffers** received in ppPacketSections, as specified in section 3.1.4.7.
- The client MUST advise the server that the message was received by the message queuing application by calling the **R_EndReceive (section 3.1.4.9)** method with the following parameter values.
  - *phContext* as in the call to **R_StartReceive**.
  - *dwRequestId* set to the same value as in the call to **R_StartReceive**.
- The client MUST return the reconstructed message to the message queuing application.

3.2.4.4.2 Receive a Message with a Transaction

- If the client has not previously done so, it MUST enlist the server in a transaction as specified in section 3.2.4.2.

Copyright © 2008 Microsoft Corporation.

- The client MUST call **R_StartTransactionalReceive (section 3.1.4.13)** and MUST specify the following parameter values.
  - *phContext* set to the **QueueContextHandle (section 2.2.4)**.
  - *hCursor* set to NULL.
  - *ulAction* set to the value specified by the message queuing application.
  - *LookupId* set to the value specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - *dwRequestId* set to a value that uniquely identifies this call from all other pending calls to this protocol.
  - *pTransactionId* set to the transaction identifier.
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** received in ppSectionBuffers, as specified in section 3.1.4.7.
- The client MUST advise the server that the message was received by the message queuing application by calling the **R_EndTransactionalReceive (section 3.1.4.15)** method with:
  - The same *phContext* parameter as in the call to **R_StartTransactionalReceive**.
  - The same *dwRequestId* as in the call to **R_StartTransactionalReceive**.
- The client MUST return the reconstructed message to the message queuing application.

3.2.4.5 Reject a Message

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** and the *LookupId* of the message to be rejected.

- The client MUST call the **R_SetUserAcknowledgementClass (section 3.1.4.14)** method and MUST specify the following parameter values:
  - *phContext* set to the **QueueContextHandle**.
  - *LookupId* set to the value passed by the client.
  - *ulClass* set to MQMSG_CLASS_NACK_RECEIVE_REJECTED.

3.2.4.6 Move a Message

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** of the source queue and the **QueueContextHandle** of the destination queue. The message queuing application MUST specify the transaction identifier and the *LookupId* of the message to be moved.

- The client MUST have enlisted the server in the transaction as specified in section 3.2.4.2.
- The client MUST call the **R_MoveMessage** method and MUST specify the following parameter values.
  - *phContextFrom* set to the **QueueContextHandle** of the source queue.

Copyright © 2008 Microsoft Corporation.

- *phContextFrom* set to the **QueueContextHandle** of the destination queue.
- *pTransactionId* set to the transaction identifier.
- *LookupId* set to the value specified by the message queuing application.

3.2.4.7 Purging a Queue

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** of the queue. The client MUST call the **R_PurgeQueue (section 3.1.4.6)** method with *phContext* set to the **QueueContextHandle**.

3.2.4.8 Creating a Cursor

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** to associate with the created cursor. The client MUST call the **R_CreateCursor (section 3.1.4.4)** method with *phContext* set to the **QueueContextHandle**. The client MUST record the returned cursor handle and return it to the message queuing application.

3.2.4.9 Peek a Message by Using a Cursor

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** of the queue to be read from, the cursor handle, the time-out parameter for the operation, a maximum message body size, and an action from the table in the description of the *ulAction* parameter in **R_StartReceive (section 3.1.4.7)** with an action type of Peek.

- The client MUST call the **R_StartReceive** method and MUST specify the following parameter values.
  - *hCursor* set to the value specified by the message queuing application.
  - *LookupId* set to NULL.
  - *ulAction* set to the action specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - *dwRequestId* set to a value that uniquely identifies this call from all other pending calls to this protocol.
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** received in ppPacketSections, as specified in section 3.1.4.7.
- The client MUST return the message to the message queuing application.

3.2.4.10 Receive a Message by Using a Cursor

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** of the queue to be read from, the cursor handle, a transaction identifier, the time-out parameter for the operation, a maximum message body size, and an action from the table in the description of the *ulAction* parameter (as specified in section 3.1.4.7) with action type of Receive.

If the transaction identifier specified by the message queuing application is NULL, follow the sequencing rules as specified in section 3.2.4.10.1.

Copyright © 2008 Microsoft Corporation.

If the transaction identifier specified by the message queuing application is non-NULL, follow the sequencing rules as specified in section 3.2.4.10.2.

3.2.4.10.1 Receive a Message by Using a Cursor Without a Transaction

- The client MUST call the **R_StartReceive (section 3.1.4.7)** method and MUST specify the following parameter values.
  - *phContext* set to the **QueueContextHandle (section 2.2.4)**.
  - *hCursor* set to the value specified by the message queuing application.
  - *ulAction* set to the value specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - *dwRequestId* set to a value that uniquely identifies this call from all other pending calls to this protocol.
  - *LookupId* set to 0.
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** received in ppPacketSections, as specified in section **R_StartReceive** (section 3.1.4.7).
- The client MUST advise the server that the message was received by the message queuing application by calling the **R_EndReceive (section 3.1.4.9)** method with:
  - The same *phContext* parameter as in the call to **R_StartReceive**.
  - The same *dwRequestId* as in the call to **R_StartReceive** (section 3.1.4.7).
- The client MUST return the reconstructed message to the message queuing application.

3.2.4.10.2 Receive a Message by Using a Cursor with a Transaction

- The client MUST have previously enlisted the server in the transaction as specified in section 3.2.4.2.
- The client MUST call the **R_StartTransactionalReceive (section 3.1.4.13)** method and MUST specify the following parameter values.
  - *phContext* set to the **QueueContextHandle (section 2.2.4)**.
  - *hCursor* set to the cursor handle specified by the message queuing application.
  - *ulAction* parameter set to the value specified by the message queuing application.
  - *ulTimeout* set to the time-out value.
  - *dwMaxBodySize* set to the value specified by the message queuing application.
  - A *dwRequestId* parameter value that uniquely identifies this call from all other pending calls to this protocol.
  - *pTransactionId* set to the transaction identifier.

Copyright © 2008 Microsoft Corporation.

- *LookupId* set to 0.

- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** received in ppPacketSections, as specified in section 3.1.4.7.
- The client MUST advise the server that the message was received by the message queuing application by calling the **R_EndTransactionalReceive (section 3.1.4.15)** method with:
  - The same *phContext* parameter as in the call to **R_StartTransactionalReceive** (section 3.1.4.13).
  - The same *dwRequestId* as in the call to **R_StartTransactionalReceive**.
- The client MUST return the reconstructed message to the message queuing application.

3.2.4.11 Cancel a Pending Peek or Receive

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** and the *dwRequestId* of the operation to be canceled.

- The client MUST call the **R_CancelReceive (section 3.1.4.8)** method with *phContext* set to the **QueueContextHandle** and *dwRequestId* set to the *dwRequestId* passed by the message queuing application.

3.2.4.12 Closing a Cursor

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** and the cursor handle to be closed.

- If there are any pending requests associated with the cursor handle, the client SHOULD cancel them as specified in section 3.2.4.11.<39>
- The client MUST call the **R_CloseCursor (section 3.1.4.5)** method with the following:
  - *phContext* set to the **QueueContextHandle**.
  - *phCursor* set to the cursor handle.
- The client MUST remove the cursor handle from its state.

3.2.4.13 Closing a Queue

The message queuing application MUST specify the **QueueContextHandle (section 2.2.4)** to be closed. If there are any pending requests associated with the **QueueContextHandle**, the client SHOULD cancel them as specified in section 3.2.4.11. If any open cursor handles are associated with the **QueueContextHandle**, the client SHOULD close them as specified in section 3.2.4.12. The client MUST call the **R_CloseQueue (section 3.1.4.3)** method with *pphContext* set to the **QueueContextHandle**. The client MUST remove the **QueueContextHandle** from its state.<40>

3.2.5 Timer Events

None.

3.2.6 Other Local Events

None.

Copyright © 2008 Microsoft Corporation.

4 Protocol Examples

The following sections describe several operations that are used in common scenarios in order to illustrate the function of the Message Queuing (MSMQ): Queue Manager Remote Read Protocol.

4.1 Binding to a Server and Purging a Queue

The sequence diagram that follows illustrates a scenario when the client purges a queue. In addition, it shows how the static RPC endpoint port is acquired by the client to create an RPC binding handle.

1. The client begins the sequence by creating an RPC binding for the server. Next, the client calls the **R_GetServerPort (section 3.1.4.1)** method, which returns an RPC endpoint port number with which the client creates a new binding. The client uses the new binding for all subsequent calls to the server.

2. Using the binding from the previous step, the client calls the **R_OpenQueue (section 3.1.4.2)** method, requesting the MQ_RECEIVE_ACCESS (0x00000001) access mode and a share mode, in addition to client-specific values for the following parameters. *pClientId*, *fNonRoutingServer*, *Major*, *Minor*, *BuildNumber*, and *fWorkgroup*. On success, the server returns a new **QueueContextHandle (section 2.2.4)**.

3. The client calls the **R_PurgeQueue (section 3.1.4.6)** method. The server confirms that the queue was opened with the MQ_RECEIVE_ACCESS (0x00000001) access mode, and then removes all messages from the queue.

4. Finally, the client closes the **QueueContextHandle** with a call to **R_CloseQueue (section 3.1.4.3)**.

Copyright © 2008 Microsoft Corporation.

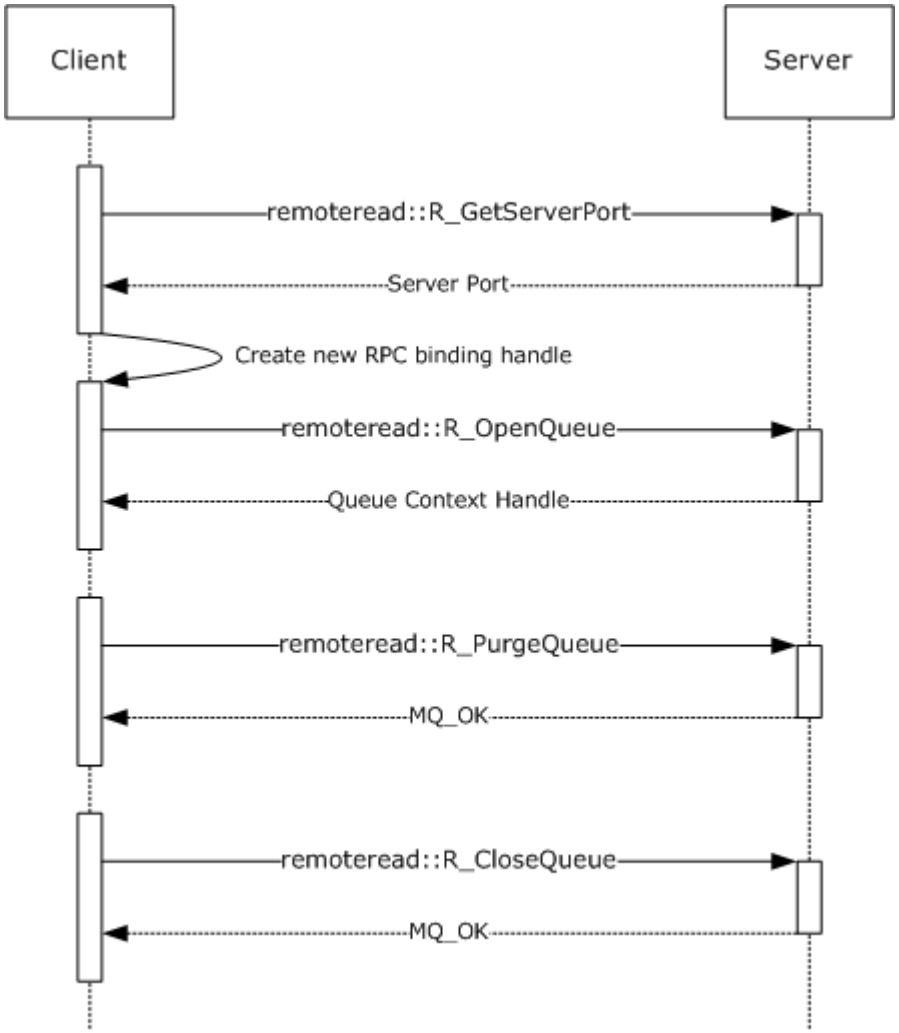


Figure 3: The client binds to a server and purges a queue

4.2 Receiving a Message

This sequence diagram illustrates a client receiving a message from a queue at the server. The call to **R_StartReceive (section 3.1.4.7)** includes a *ulAction* value of MQ_ACTION_RECEIVE (0x00000000), and a unique *dwRequestId* value chosen by the client. In response, the server associates a pending request with the passed *dwRequestId*, which is used to correlate a subsequent call to **R_EndReceive (section 3.1.4.9)** or **R_CancelReceive (section 3.1.4.8)** with the same value for *dwRequestId*. Additionally, the server returns a **SectionBuffer (section 2.2.6)** array that contains the message.

Next, the client indicates that the message was successfully received by calling **R_EndReceive**, specifying RR_ACK (0x00000002) for *dwAck*. The server completes the corresponding pending request created by the call to **R_StartReceive**, and because RR_ACK is specified, removes the message from the queue.

Copyright © 2008 Microsoft Corporation.

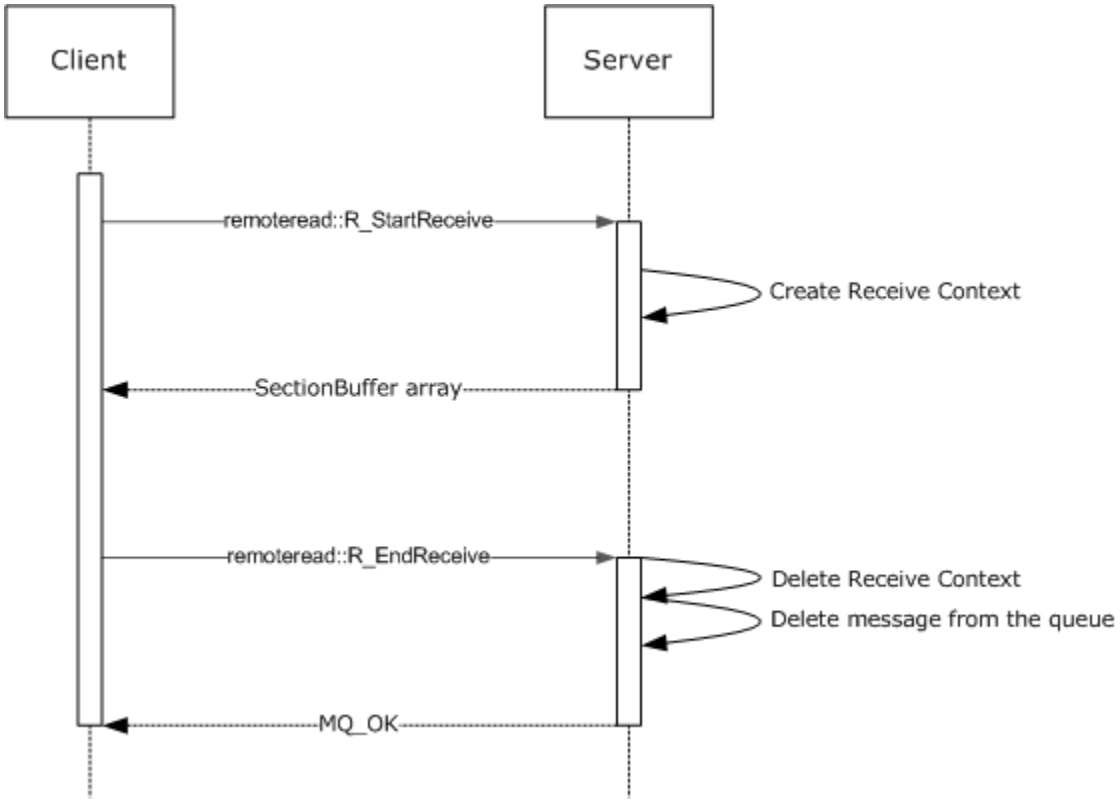


Figure 4: The client receives a message

4.3 Receiving a Message in a Transaction

This sequence diagram demonstrates a scenario in which a client receives a message from a queue within the context of a transaction. Although four roles are used to illustrate the participants in this scenario, the protocol that is described by this specification is used only between the client and server roles. The "Client Distributed Transaction Coordinator (DTC)" role (as specified in [MS-DTCO]) and the "Server DTC" role are included to illustrate a typical end-to-end sequence of a transactional receive request.

The diagram includes reference numbers on the left side that identify operations of interest, which are explained in detail below.

Copyright © 2008 Microsoft Corporation.

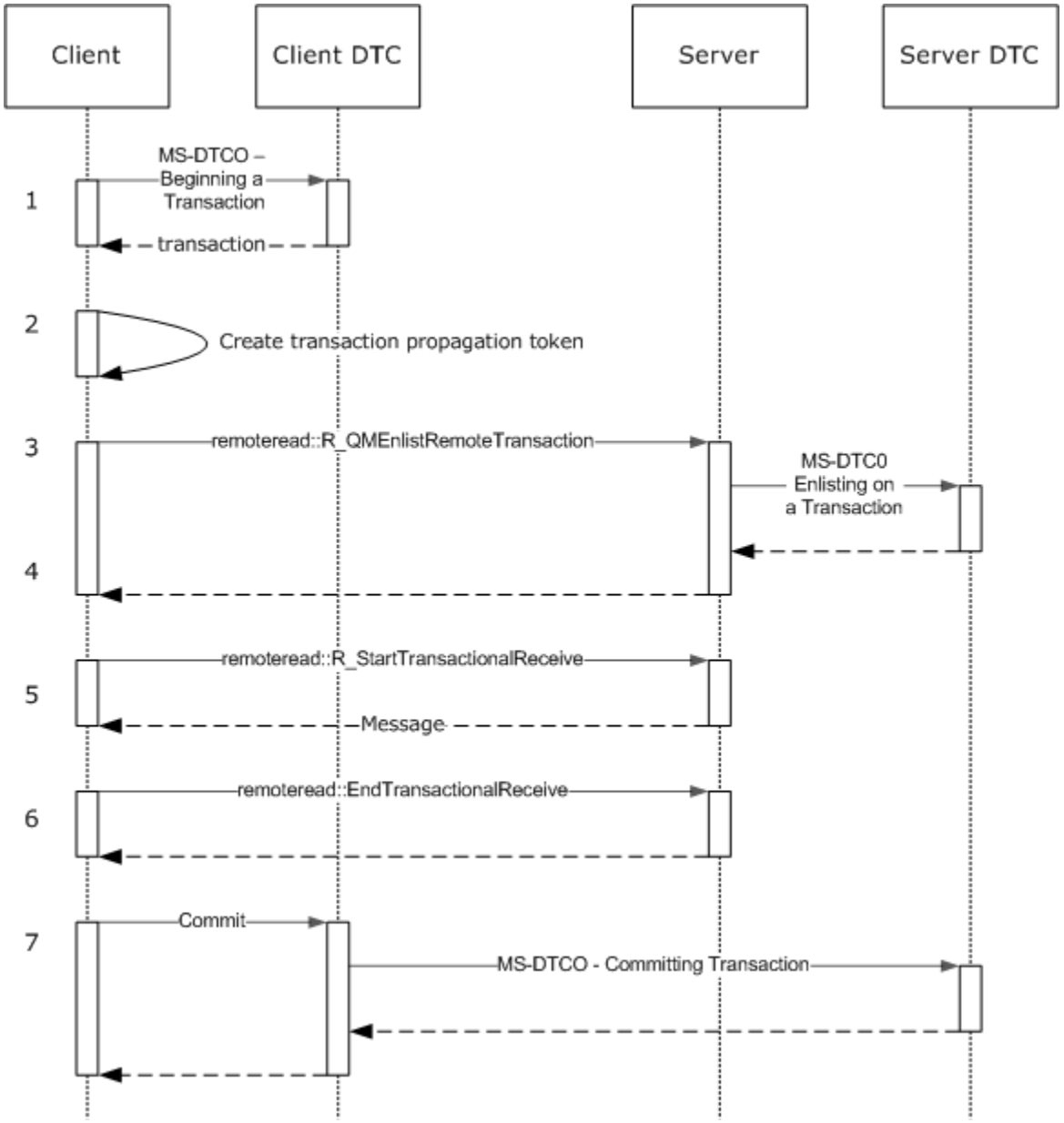


Figure 5: The client receives a message within a transaction

1. The client communicates with the local DTC to create a new transaction, as specified in [MS-DTCO] section 3.3.4.1.

2. The client constructs a propagation token to be marshaled to the server's transaction manager, as specified in [MS-DTCO] section 2.2.5.4

3. The client calls the **R_QMEnlistRemoteTransaction (section 3.1.4.12)** method, specifying the transaction identifier and the transaction propagation token.

Copyright © 2008 Microsoft Corporation.

4. The server marshals the transaction propagation token to its local transaction manager and enlists its local resource manager in the transaction, as specified in [MS-DTCO] sections 3.3.4.11 and 3.5.4.3.

5. The client calls the **R_StartTransactionalReceive (section 3.1.4.13)** method to receive a message in the context of the transaction. The client specifies the transaction identifier to associate the receive operation with the transaction enlisted in the prior steps. The server returns a message in the **SectionBuffer (section 2.2.6)** array.

6. The client advises the server that the message was received correctly by calling the **R_EndTransactionalReceive (section 3.1.4.15)** method, specifying RR_ACK (0x00000002) for *dwAck*.

7. Finally, the client commits the transaction. The server is notified of the commit, and deletes the received message from the queue, as specified in [MS-DTCO] section 3.4.1.1.6.

Copyright © 2008 Microsoft Corporation.

5 Security

The following sections specify security considerations for implementers of the Message Queuing (MSMQ): Queue Manager Remote Read Protocol.

5.1 Security Considerations for Implementers

The server SHOULD impose the minimum RPC authentication level on the RPC handle for incoming calls. The server MAY<41> require different minimum RPC authentication level from the client, depending on whether the client is a member of a Windows domain, as specified by the *fWorkgroup* parameter in the **R_OpenQueue (section 3.1.4.2)** and the **R_OpenQueueForMove (section 3.1.4.11)** methods.

5.2 Index of Security Parameters

Security parameter	Section
fWorkgroup	**R_OpenQueue (Opnum 2) (section 3.1.4.2)** **R_OpenQueueForMove (Opnum 11) (section 3.1.4.11)**

Copyright © 2008 Microsoft Corporation.

6 Appendix A: Full IDL

For ease of implementation, the full **IDL** is provided in this section, where "ms-dtyp.idl" is the IDL as specified in [MS-DTYP] Appendix A.

```
import "ms-dtyp.idl";

[
    uuid(1a9134dd-7b39-45ba-ad88-44d01ca47f28),
    version(1.0),
    pointer_default(unique)
]
interface RemoteRead
{
    typedef [context_handle] void* QueueContextHandle;

    typedef enum
    {
      QUEUE_FORMAT_TYPE_UNKNOWN = 0,
      QUEUE_FORMAT_TYPE_PUBLIC = 1,
      QUEUE_FORMAT_TYPE_PRIVATE = 2,
      QUEUE_FORMAT_TYPE_DIRECT = 3,
      QUEUE_FORMAT_TYPE_MACHINE = 4,
      QUEUE_FORMAT_TYPE_CONNECTOR = 5,
      QUEUE_FORMAT_TYPE_DL = 6,
      QUEUE_FORMAT_TYPE_MULTICAST = 7,
      QUEUE_FORMAT_TYPE_SUBQUEUE = 8
    } QUEUE_FORMAT_TYPE;

    typedef enum
    {
      QUEUE_SUFFIX_TYPE_NONE = 0,
      QUEUE_SUFFIX_TYPE_JOURNAL=1,
      QUEUE_SUFFIX_TYPE_DEADLETTER=2,
      QUEUE_SUFFIX_TYPE_DEADXACT=3,
      QUEUE_SUFFIX_TYPE_XACTONLY=4,
      QUEUE_SUFFIX_TYPE_SUBQUEUE=5
    } QUEUE_SUFFIX_TYPE;

    typedef struct _OBJECTID
    {
        GUID Lineage;
        unsigned long Uniquifier;
    } OBJECTID;

    typedef struct _DL_ID
    {
        GUID   m_DlGuid;
        wchar_t* m_pwzDomain;
    } DL_ID;

    typedef struct _MULTICAST_ID
    {
        unsigned long m_address;
        unsigned long m_port;
```


Copyright © 2008 Microsoft Corporation.

```
} MULTICAST_ID;

typedef struct QUEUE_FORMAT
{
  unsigned char m_qft;
  unsigned char m_SuffixAndFlags;
  unsigned short m_reserved;
  [switch_is(m_qft)] union {

    [case(QUEUE_FORMAT_TYPE_PUBLIC)]
      GUID m_gPublicID;
    [case(QUEUE_FORMAT_TYPE_PRIVATE)]
      OBJECTID  m_oPrivateID;
    [case(QUEUE_FORMAT_TYPE_DIRECT)]
      wchar_t * m_pDirectID;
    [case(QUEUE_FORMAT_TYPE_MACHINE)]
      GUID m_gMachineID;
    [case(QUEUE_FORMAT_TYPE_CONNECTOR)]
      GUID m_GConnectorID;
    [case(QUEUE_FORMAT_TYPE_DL)]
      DL_ID m_DlID;
    [case(QUEUE_FORMAT_TYPE_MULTICAST)]
      MULTICAST_ID m_MulticastID;
    [case(QUEUE_FORMAT_TYPE_SUBQUEUE)]
      wchar_t * m_pDirectSubqueueID;
  };
} __QUEUE_FORMAT;

typedef enum
{
  stFullPacket = 0,
  stBinaryFirstSection = 1,
  stBinarySecondSection = 2,
  stSrmpFirstSection = 3,
  stSrmpSecondSection = 4
} SectionType;

typedef struct _SectionBuffer {
    SectionType  SectionBufferType;
    DWORD SectionSizeAlloc;
    DWORD SectionSize;
    [unique, size_is(SectionSize)] byte* pSectionBuffer;
} SectionBuffer;

DWORD R_GetServerPort(
    [in] handle_t hBind
    );

void Opnum1NotUsedOnWire(void);

void R_OpenQueue(
    [in] handle_t hBind,
    [in] struct QUEUE_FORMAT* pQueueFormat,
    [in] DWORD dwAccess,
    [in] DWORD dwShareMode,
    [in] GUID* pClientId,
    [in] LONG fNonRoutingServer,
    [in] unsigned char Major,
```


Copyright © 2008 Microsoft Corporation.

```
    [in] unsigned char Minor,
    [in] USHORT BuildNumber,
    [in] LONG fWorkgroup,
    [out] QueueContextHandle* pphContext
    );

HRESULT R_CloseQueue(
    [in] handle_t  hBind,
    [in, out] QueueContextHandle* pphContext
    );

HRESULT R_CreateCursor(
    [in]  handle_t hBind,
    [in]  QueueContextHandle phContext,
    [out] DWORD* phCursor
    );

HRESULT R_CloseCursor(
    [in] handle_t hBind,
    [in] QueueContextHandle phContext,
    [in] DWORD hCursor
    );

HRESULT R_PurgeQueue(
    [in] handle_t hBind,
    [in] QueueContextHandle phContext
    );

HRESULT R_StartReceive(
    [in] handle_t  hBind,
    [in] QueueContextHandle phContext,
    [in] ULONGLONG LookupId,
    [in] DWORD hCursor,
    [in] DWORD ulAction,
    [in] DWORD ulTimeout,
    [in] DWORD dwRequestId,
    [in] DWORD dwMaxBodySize,
    [in] DWORD dwReserved,
    [out] DWORD* pdwArriveTime,
    [out] ULONGLONG* pSequenceId,
    [out] DWORD* pdwNumberOfSections,
    [out, size_is(, *pdwNumberOfSections)]
         SectionBuffer** ppPacketSections
    );

HRESULT R_CancelReceive(
    [in] handle_t hBind,
    [in] QueueContextHandle phContext,
    [in] DWORD dwRequestId
    );

HRESULT R_EndReceive(
    [in] handle_t  hBind,
    [in] QueueContextHandle phContext,
    [in, range(1,2)] DWORD dwAck,
    [in] DWORD dwRequestId
    );
```


Copyright © 2008 Microsoft Corporation.

```
HRESULT R_MoveMessage(
    [in]  handle_t hBind,
    [in]  QueueContextHandle phContextFrom,
    [in]  ULONGLONG ullContextTo,
    [in]  ULONGLONG LookupId,
    [in]  GUID *pTransactionId
    );

void R_OpenQueueForMove(
    [in] handle_t hBind,
    [in] struct QUEUE_FORMAT* pQueueFormat,
    [in] DWORD dwAccess,
    [in] DWORD dwShareMode,
    [in] GUID* pClientId,
    [in] LONG fNonRoutingServer,
    [in] unsigned char Major,
    [in] unsigned char Minor,
    [in] USHORT BuildNumber,
    [in] LONG fWorkgroup,
    [out] ULONGLONG  *pMoveContext,
    [out] QueueContextHandle* pphContext
    );

HRESULT R_QMEnlistRemoteTransaction(
    [in] handle_t hBind,
    [in] GUID* pTransactionId,
    [in, range(0, 131072)] DWORD cbPropagationToken,
    [in, size_is (cbPropagationToken)]
          unsigned char* pbPropagationToken,
    [in] struct QUEUE_FORMAT* pQueueFormat
    );

HRESULT R_StartTransactionalReceive(
    [in] handle_t  hBind,
    [in] QueueContextHandle phContext,
    [in] ULONGLONG LookupId,
    [in] DWORD hCursor,
    [in] DWORD ulAction,
    [in] DWORD ulTimeout,
    [in] DWORD dwRequestId,
    [in] DWORD dwMaxBodySize,
    [in] DWORD dwReserved,
    [in] GUID* pTransactionId,
    [out] DWORD* pdwArriveTime,
    [out] ULONGLONG* pSequenceId,
    [out] DWORD* pdwNumberOfSections,
    [out, size_is(, *pdwNumberOfSections)]
           SectionBuffer** ppPacketSections
    );

HRESULT R_SetUserAcknowledgementClass(
    [in]  handle_t  hBind,
    [in]  QueueContextHandle phContext,
    [in]  ULONGLONG LookupId,
    [in]  USHORT usClass
    );

HRESULT R_EndTransactionalReceive(
```


Copyright © 2008 Microsoft Corporation.

```
            [in] handle_t  hBind,
            [in] QueueContextHandle phContext,
            [in, range(1,2)] DWORD dwAck,
            [in] DWORD dwRequestId
            );
    }
```


Copyright © 2008 Microsoft Corporation.

7 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows NT
- Windows 2000
- Windows Server 2003
- Windows Vista
- Windows Server 2008

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 1.4: Windows NT and Windows 2000 support MSMQ: Queue Manager to Queue Manager Protocol (as specified in [MS-MQQP]) but do not support the MSMQ: Queue Manager Remote Read Protocol.

<2> Section 1.7: Windows Server 2003, Windows Vista, and Windows Server 2008 use Kerberos when the computer is a member of a Windows domain; otherwise, they use NTLM.

<3> Section 2.1: The Windows Server 2003 protocol client uses RPC dynamic endpoints to obtain the initial RPC binding handle. The client calls the **R_GetServerPort** method with the initial RPC binding handle and uses the returned value to obtain a new RPC binding handle to be used for all subsequent RPC method calls on the protocol. The Windows Vista and Windows Server 2008 protocol clients use RPC dynamic endpoints to obtain the RPC binding handle. This handle is used for all RPC method calls on the protocol. The **R_GetServerPort** method is not called by the Windows Vista or Windows Server 2008 client.

<4> Section 2.2.3.1: QUEUE_FORMAT_TYPE_SUBQUEUE is not supported in Windows Server 2003.

<5> Section 2.2.3.2: QUEUE_SUFFIX_TYPE_SUBQUEUE is not supported on Windows Server 2003.

<6> Section 2.2.3.6: QUEUE_SUFFIX_TYPE_SUBQUEUE is not supported in Windows Server 2003.

<7> Section 2.2.3.6: If the **m_qft** element of the QUEUE_FORMAT structure is set to QUEUE_FORMAT_TYPE_DIRECT, Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers require the address specification to be TCP or OS.

<8> Section 2.2.3.6: Subqueues are not supported in Windows Server 2003.

<9> Section 2.2.5.2: The ExtensionHeader is not supported on Windows Server 2003.

<10> Section 2.2.5.3: The SubqueueHeader is not supported on Windows Server 2003.

<11> Section 2.2.5.4: The DeadLetterHeader is not supported on Windows Server 2003.

<12> Section 2.2.5.5: The ExtendedAddressHeader is not supported on Windows Server 2003.

<13> Section 3.1.2: If the registry key HKLM\SOFTWARE\Microsoft\MSMQ\Parameters\RpcCancelTimeout is defined and is set to a nonzero DWORD value, the Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers interpret this value as the RPC Call Timeout value in minutes.

Copyright © 2008 Microsoft Corporation.

<14> Section 3.1.4: Opnums reserved for local use apply to Windows as follows.

Opnum	Description
1	Not used by Windows

<15> Section 3.1.4.1: The Windows Server 2003 protocol client uses RPC dynamic endpoints to obtain the initial RPC binding handle. The client calls the **R_GetServerPort (section 3.1.4.1)** method with the initial RPC binding handle and uses the returned value to obtain a new RPC binding handle to be used for all subsequent RPC method calls on the protocol.

The Windows Vista and Windows Server 2008 protocol clients use RPC dynamic endpoints to obtain the RPC binding handle. This handle is used for all RPC method calls on the protocol. The **R_GetServerPort** method is not called by the Windows Vista or Windows Server 2008 client.

<16> Section 3.1.4.2: Windows Server 2003, Windows Vista, and Windows Server 2008 generate a unique GUID when installed, and store as a binary value under the HKLM\SOFTWARE\Microsoft\MSMQ\Parameters\MachineCache\QMId registry key.

<17> Section 3.1.4.2: Windows Server 2003 protocol servers limit the number of unique concurrent clients if the following **DWORD** registry key exists and its value is 0x00000001: HKLM\SYSTEM\CurrentControlSet\Services\LicenseInfo\FilePrint. The maximum number of unique concurrent clients permitted is taken from the **DWORD** registry key HKLM\System\CurrentControlSet\Services\LicenseInfo\FilePrint\ConcurrentLimit. If the number of existing unique callers is equal to this value, R_OpenQueue throws an RPC exception MQ_ERROR_DEPEND_WKS_LICENSE_OVERFLOW (0xc00e0067L).

Windows Vista and Windows Server 2008 protocol servers do not enforce limits on the number of unique concurrent clients. The *pClientId* parameter is ignored.

<18> Section 3.1.4.2: Windows Server 2003, Windows Vista, and Windows Server 2008 protocol clients set the *fNonRoutingServer* value based on the registry key HKLM\Software\Windows\MSMQ\Parameters\MachineCache\MQS_Routing.

If this key exists and is set to the **DWORD** value 0x00000001, the parameter is set to FALSE (0x00000000); otherwise it is set to TRUE (0x00000001).

<19> Section 3.1.4.2: Windows Server 2003 protocol client sets the message queuing Major Version to 5. The Windows Vista and Windows Server 2008 protocol clients set the message queuing Major version to 6.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing Major Version parameter.

<20> Section 3.1.4.2: Windows Server 2003 protocol client sets the message queuing Minor Version to 2.

The Windows Vista and Windows Server 2008 protocol clients set the message queuing Minor Version to 0.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing Minor Version parameter.

<21> Section 3.1.4.2: Windows Server 2003, Windows Vista, and Windows Server 2008 protocol clients set the message queuing *BuildNumber* to a build-specific number.

Copyright © 2008 Microsoft Corporation.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing *BuildNumber* parameter.

<22> Section 3.1.4.2: The Windows Server 2003 protocol server minimum RPC authentication level is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if any of the following conditions is true
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any DWORD value other than 0x00000000.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroupClient is set to the DWORD value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

The Windows Vista and Windows Server 2008 protocol servers minimum RPC authentication level is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\AllowNonauthenticatedRpc exists and is set to any DWORD value other than 0x00000000 and any of the following conditions is true:
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroupClient is set to the DWORD value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

<23> Section 3.1.4.10: **R_MoveMessage** is not implemented on Windows Server 2003.

<24> Section 3.1.4.11: **R_OpenQueueForMove (section 3.1.4.11)** is not implemented on Windows Server 2003.

<25> Section 3.1.4.11: Windows Server 2003, Windows Vista, and Windows Server 2008 generate a unique GUID when installed and store as a binary value under the HKLM\SOFTWARE\Microsoft\MSMQ\Parameters\MachineCache\QMId registry key.

<26> Section 3.1.4.11: Windows Server 2003 protocol servers limit the number of unique concurrent clients if the following **DWORD** registry key exists and its value is 0x00000001:

Copyright © 2008 Microsoft Corporation.

HKLM\SYSTEM\CurrentControlSet\Services\LicenseInfo\FilePrint. The maximum number of unique concurrent clients permitted is taken from the **DWORD** registry key HKLM\System\CurrentControlSet\Services\LicenseInfo\FilePrint\ConcurrentLimit. If the number of existing unique callers is equal to this value, R_OpenQueue throws an RPC exception MQ_ERROR_DEPEND_WKS_LICENSE_OVERFLOW (0xc00e0067L).

Windows Vista and Windows Server 2008 protocol servers do not enforce limits on the number of unique concurrent clients. The *pClientId* parameter is ignored.

<27> Section 3.1.4.11: Windows Server 2003, Windows Vista, and Windows Server 2008 protocol clients set the fNonRoutingServer value based on the registry key HKLM\Software\Windows\MSMQ\Parameters\MachineCache\MQS_Routing.

If this key exists and is set to the **DWORD** value 0x00000001, the parameter is set to FALSE (0x00000000); otherwise it is set to TRUE (0x00000001).

<28> Section 3.1.4.11: Windows Server 2003 protocol client sets the message queuing Major Version to 5. The Windows Vista and Windows Server 2008 protocol clients set the message queuing Major version to 6.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing Major Version parameter.

<29> Section 3.1.4.11: Windows Server 2003 protocol client sets the message queuing Minor Version to 2.

The Windows Vista and Windows Server 2008 protocol clients set the message queuing Minor Version to 0.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing Minor Version parameter.

<30> Section 3.1.4.11: Windows Server 2003, Windows Vista, and Windows Server 2008 protocol clients set the message queuing *BuildNumber* to a build-specific number.

Windows Server 2003, Windows Vista, and Windows Server 2008 protocol servers ignore the message queuing *BuildNumber* parameter.

<31> Section 3.1.4.11: The Windows Server 2003 protocol server minimum RPC authentication level is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if any of the following conditions is true.
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any **DWORD** value other than 0x00000000.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroupClient is set to the **DWORD** value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any **DWORD** value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

Copyright © 2008 Microsoft Corporation.

The Windows Vista and Windows Server 2008 protocol servers minimum RPC authentication level is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\AllowNonauthenticatedRpc exists and is set to any **DWORD** value other than 0x00000000 and any of the following conditions is true:
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroup Client is set to the **DWORD** value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any **DWORD** value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any **DWORD** value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

<32> Section 3.1.4.12: **R_QMEnlistRemoteTransaction (section 3.1.4.12)** is not implemented on Windows Server 2003.

<33> Section 3.1.4.12: A server running Windows Vista or Windows Server 2008 ignores the *pQueueFormat* parameter.

<34> Section 3.1.4.13: **R_StartTransactionalReceive** is not implemented on Windows Server 2003.

<35> Section 3.1.4.14: **R_SetUserAcknowledgementClass (section 3.1.4.14)** is not implemented on Windows Server 2003.

<36> Section 3.1.4.15: **R_EndTransactionalReceive (section 3.1.4.15)** is not implemented on Windows Server 2003.

<37> Section 3.2.4.1: The Windows Server 2003 protocol client uses RPC dynamic endpoints to obtain the initial RPC binding handle. The client calls the **R_GetServerPort (section 3.1.4.1)** method with the initial RPC binding handle and uses the returned value to obtain a new RPC binding handle to be used for all subsequent RPC method calls on the protocol. The Windows Vista and Windows Server 2008 protocol clients use RPC dynamic endpoints to obtain the RPC binding handle. This handle is used for all RPC method calls on the protocol. The **R_GetServerPort** method is not called by the Windows Vista or Windows Server 2008 client.

<38> Section 3.2.4.1: The Windows Server 2003 protocol client uses RPC dynamic endpoints to obtain the initial RPC binding handle. The client calls the **R_GetServerPort (section 3.1.4.1)** method with the initial RPC binding handle and uses the returned value to obtain a new RPC binding handle to be used for all subsequent RPC method calls on the protocol. The Windows Vista and Windows Server 2008 protocol clients use RPC dynamic endpoints to obtain the RPC binding handle. This handle is used for all RPC method calls on the protocol. The **R_GetServerPort** method is not called by the Windows Vista or Windows Server 2008 client.

<39> Section 3.2.4.12: Windows Server 2003, Windows Vista, and Windows Server 2008 clients do not cancel pending requests associated with open cursor handles.

Copyright © 2008 Microsoft Corporation.

<40> Section 3.2.4.13: Windows Server 2003, Windows Vista, and Windows Server 2008 clients do not cancel pending requests or close associated cursor handles.

<41> Section 5.1: The minimum RPC authentication level for Windows Server 2003 protocol server is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if any of the following conditions is true.
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any DWORD value other than 0x00000000.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroupClient is set to the DWORD value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

The Windows Vista and Windows Server 2008 protocol servers minimum RPC authentication level is determined as follows:

- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\AllowNonauthenticatedRpc exists and is set to any DWORD value other than 0x00000000 and any of the following conditions is true.
  - The *fWorkgroup* parameter is TRUE.
  - The registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerDenyWorkgroupClient is set to the DWORD value 0x00000000 or does not exist.
- RPC_C_AUTHN_LEVEL_NONE, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\NewRemoteReadServerAllowNoneSecurityClient exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, if the registry key HKLM\Software\Microsoft\MSMQ\Parameters\security\DebugRpc exists and is set to any DWORD value other than 0x00000000.
- RPC_C_AUTHN_LEVEL_PKT_PRIVACY otherwise.

Copyright © 2008 Microsoft Corporation.

8 Index

Copyright © 2008 Microsoft Corporation.

S

T

U

V

W

Copyright © 2008 Microsoft Corporation.