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

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.
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.
03/14/2008	2.0.5	Editorial	Revised and edited the technical content.
05/16/2008	2.0.6	Editorial	Revised and edited the technical content.
06/20/2008	2.1	Minor	Updated the technical content.
07/25/2008	2.1.1	Editorial	Revised and edited the technical content.
08/29/2008	3.0	Major	Updated and revised the technical content.
10/24/2008	4.0	Major	Updated and revised the technical content.
12/05/2008	5.0	Major	Updated and revised the technical content.
01/16/2009	5.1	Minor	Updated the technical content.
02/27/2009	6.0	Major	Updated and revised the technical content.
04/10/2009	6.0.1	Editorial	Revised and edited the technical content.
05/22/2009	7.0	Major	Updated and revised the technical content.
07/02/2009	7.1	Minor	Updated the technical content.
08/14/2009	8.0	Major	Updated and revised the technical content.
09/25/2009	9.0	Major	Updated and revised the technical content.
11/06/2009	9.1	Minor	Updated the technical content.
12/18/2009	10.0	Major	Updated and revised the technical content.
01/29/2010	11.0	Major	Updated and revised the technical content.

Date	Revision History	Revision Class	Comments
03/12/2010	11.1	Minor	Updated the technical content.
04/23/2010	11.1.1	Editorial	Revised and edited the technical content.
06/04/2010	11.2	Minor	Updated the technical content.
07/16/2010	11.2	No change	No changes to the meaning, language, or formatting of the technical content.
08/27/2010	12.0	Major	Significantly changed the technical content.
10/08/2010	13.0	Major	Significantly changed the technical content.
11/19/2010	13.0	No change	No changes to the meaning, language, or formatting of the technical content.
01/07/2011	14.0	Major	Significantly changed the technical content.
02/11/2011	15.0	Major	Significantly changed the technical content.
03/25/2011	16.0	Major	Significantly changed the technical content.
05/06/2011	16.0	No change	No changes to the meaning, language, or formatting of the technical content.
06/17/2011	16.1	Minor	Clarified the meaning of the technical content.

Contents

1 Introduction

This document specifies the Message Queuing (MSMQ): Queue Manager Remote Read Protocol, a **remote procedure call (RPC)**-based protocol that is used by **Microsoft Message Queuing (MSMQ)** clients to read or reject a **message** from a **queue**, to move a message between queues, and to 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)
Internet host name
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-MQMQ]:

connector queue
cursor
dead-letter queue
direct format name
distribution list (DL)
message
message body
message packet
message packet header
message packet trailer
message property
message queuing
Microsoft Message Queuing (MSMQ)
MSMQ routing server
private queue
public queue
queue
queue journal
queue manager
remote queue
subqueue
transactional queue

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.

1.2 References

References to Microsoft Open Specification documents do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

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) Specification".

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

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

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

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

[MS-MQDMPR] Microsoft Corporation, "Message Queuing (MSMQ): Common Data Model and Processing Rules".

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

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

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

[MS-MQSO] Microsoft Corporation, "Message Queuing System Overview".

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

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

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

[MS-MQDSSM] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Schema Mapping".

[MSDN-MMSCH] Microsoft Corporation, "Mixed Mode Serialization of Context Handles", http://msdn.microsoft.com/en-us/library/aa367098(VS.85).aspx

1.3 Overview

Microsoft Message Queuing (MSMQ) is a communications service that provides asynchronous and reliable message passing between client applications running on different hosts. In MSMQ, 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 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 **remote queue**.

1.3.1 Messages

Each message exchanged in a MSMQ 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 are 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 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 nondestructive 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 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 cursors 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 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 nontransactional **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 are advised to 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.

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)**

Capability Negotiation: This protocol SHOULD<2> be used for receiving messages from a remote queue manager. The queue manager SHOULD implement a capability negotiation mechanism as described in the processing rules for Opening a Queue (section 3.2.4.1) to determine whether this protocol is supported by the remote queue manager.

Security and authentication methods: This protocol supports the NTLM and Kerberos authentication methods.<3>

1.8 Vendor-Extensible Fields

This protocol uses HRESULT values as defined in [MS-ERREF] section 2.1. Vendors can define their own HRESULT values provided that they set the C bit (0x20000000) for each vendor-defined value, indicating that the value is a customer code.

1.9 Standards Assignments

Parameter	Value	Reference
RPC interface **Universally Unique Identifier (UUID)**	1A9134DD-7B39-45BA-AD88-44D01CA47F28	[C706] A.2.5
Interface version	1.0	[C706] A.2.5

2 Messages

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 uses RPC **dynamic endpoints** as specified in [C706] section 4. This protocol MAY use an RPC static endpoint as specified in [C706] section 4.<4>

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 **Network Data Representation (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-RPCE], additional data types are defined in the following list that summarizes the types defined in this specification:

- HRESULT
- GUID
- **QUEUE_FORMAT**
- Queue Context Handles
- 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 (GUID)**. Unless otherwise qualified, instances of **GUID** in sections 2 and 3 refer to [MS-DTYP] section 2.3.2.

2.2.3 QUEUE_FORMAT

This structure is used to identify a queue. This structure is common to many Microsoft Message Queuing (MSMQ) protocols. For more details, see [MS-MQMQ] section 2.2.7. Only a subset of the **QUEUE_FORMAT_TYPE** enumeration is supported by this protocol. This subset is:

- **QUEUE_FORMAT_TYPE_UNKNOWN**
- **QUEUE_FORMAT_TYPE_PUBLIC**

- **QUEUE_FORMAT_TYPE_PRIVATE**
- **QUEUE_FORMAT_TYPE_DIRECT**
- **QUEUE_FORMAT_TYPE_MACHINE**
- **QUEUE_FORMAT_TYPE_SUBQUEUE**

In addition, this protocol supports only a subset of the Protocol Address Specifications defined for QUEUE_FORMAT in [MS-MQMQ] section 2.1.2 when the m_qft field of this structure is set to **QUEUE_FORMAT_TYPE_DIRECT**. This subset is:

- TCP
- OS

2.2.4 Queue Context Handles

A queue context handle is 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 queue context handle and **R_CloseQueue (section 3.1.4.3)** to delete a queue context handle.

Two **IDL** types are defined to represent these queue context handles, namely **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** and **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**. These two types are identical on the wire, but are defined separately so as to allow the serialization mode to be configured. Refer to [MSDN-MMSCH] for details on modes of the context handles.

2.2.4.1 QUEUE_CONTEXT_HANDLE_NOSERIALIZE

QUEUE_CONTEXT_HANDLE_NOSERIALIZE is an RPC context handle representing an open queue. Refer to [MSDN-MMSCH] for details on modes of the context handles. For the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** context handle, there may be more than one pending RPC call on the server. On the wire it is identical to **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**.

This type is declared as follows:

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

The context handle MUST NOT be type_strict, but it MUST be strict. More details on RPC context handles are specified in [C706] sections 4.2.16.6, 5.1.6, and 6.1 and [MS-RPCE] sections 3.1.1.5.3.2.2.2 and 3.3.1.4.1.

2.2.4.2 QUEUE_CONTEXT_HANDLE_SERIALIZE

QUEUE_CONTEXT_HANDLE_SERIALIZE is an RPC context handle representing an open queue. Refer to [MSDN-MMSCH] for details on modes of the context handles. For this context handle, there can be no more than one pending RPC call on the server. On the wire it is identical to **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)**.

This type is declared as follows:

```
typedef [context_handle] QUEUE_CONTEXT_HANDLE_NOSERIALIZE QUEUE_CONTEXT_HANDLE_SERIALIZE;
```

The context handle MUST NOT be type_strict, but it MUST be strict. More details on RPC context handles are specified in [C706] sections 4.2.16.6, 5.1.6, and 6.1 and [MS-RPCE] sections 3.1.1.5.3.2.2.2 and 3.3.1.4.1.

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

...
ExtendedAddressHeader
...
...
...
...
...
...

UserMessage (variable): A UserMessage (section 2.2.5.1) structure.

ExtensionHeader (12 bytes): An ExtensionHeader (section 2.2.5.2) structure.

SubqueueHeader (148 bytes): A SubqueueHeader (section 2.2.5.3) structure.

DeadLetterHeader (variable): A DeadLetterHeader (section 2.2.5.4) structure.

ExtendedAddressHeader (28 bytes): An ExtendedAddressHeader (section 2.2.5.5) structure.

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) depending on the transport over which the message was originally sent. A Binary Message is sent over the MSMQ: Binary Reliable Messaging Protocol [MS-MQQB], while an SRMP Message is sent over HTTP. The message type is indicated by the UserHeader.Flags.AH bit which is set for SRMP Messages as described in the definition of UserHeader.UserHeaderEnd in this section.

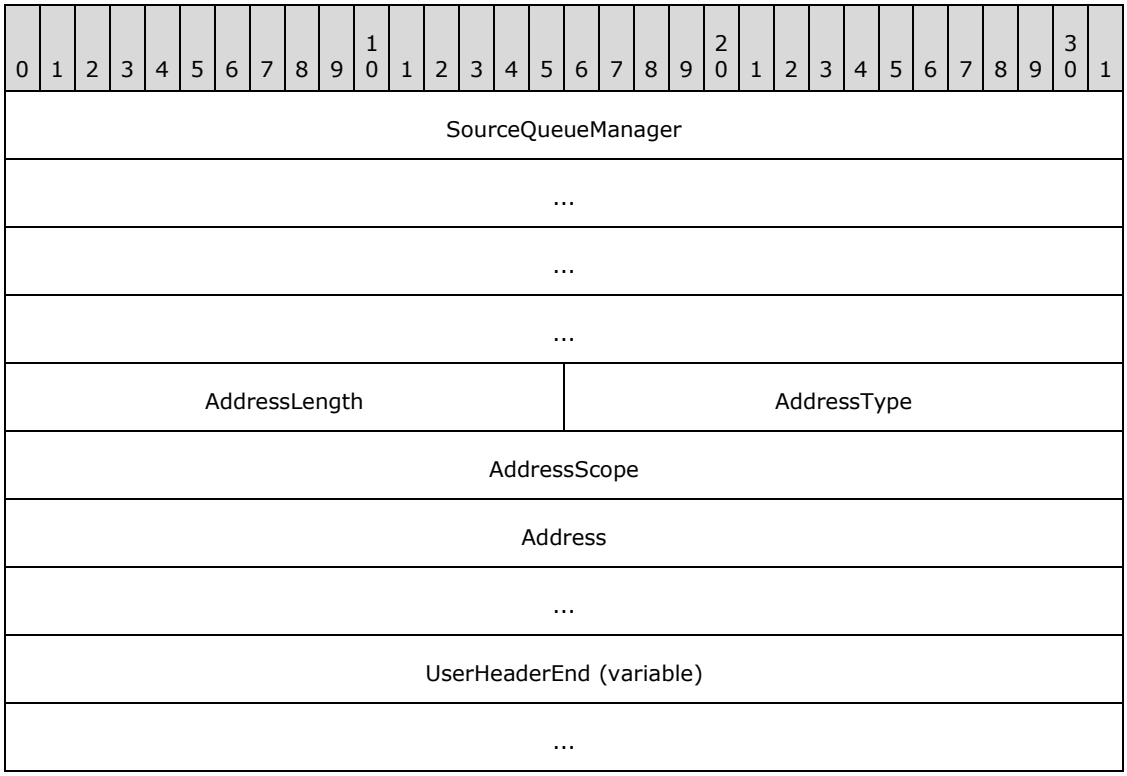
0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
BaseHeader																															
...																															
...																															
...																															
UserHeader (variable)																															
...																															

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

BaseHeader (16 bytes): A BaseHeader, as specified in [MS-MQMQ] section 2.2.19.1. The **TimeToReachQueue** field has the same length and format as that specified in [MS-MQMQ], 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-MQMQ] section 2.2.19.2) 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. If the UserHeader specifies that the destination queue is anything other than a direct format name, the 16 bytes after the **SourceQueueManager** are set to the **GUID** of the host from which the message was received, as specified in [MS-MQMQ] section 2.2.19.2.



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 (8 bytes): An 8-byte array of **UCHAR**, 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-MQMQ] section 2.2.19.2) beginning with the **TimeToBeReceived** field. Within the **Flags** field, **AH** MUST be set only if both **Message.SOAPEnvelope** and **Message.SOAPCompoundMessage** ([MS-MQDMPR] section 3.1.1.12) are populated.

TransactionHeader (variable): A TransactionHeader, as specified in [MS-MQMQ] section 2.2.20.5.

SecurityHeader (variable): A SecurityHeader, as specified in [MS-MQMQ] section 2.2.20.6.

MessagePropertiesHeader (variable): A MessagePropertiesHeader, as specified in [MS-MQMQ] section 2.2.19.3.

DebugHeader (variable): A DebugHeader, as specified in [MS-MQMQ] section 2.2.20.8.

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-MQMQ] section 2.2.20.7.

MultiQueueFormatHeader (variable): A MultiQueueFormatHeader, as specified in [MS-MQMQ] section 2.2.20.1.

SessionHeader (16 bytes): A SessionHeader, as specified in [MS-MQMQ] section 2.2.20.4. The SessionHeader is used to acknowledge express and recoverable UserMessage packets when they are sent on a session. This header MUST be present if and only if the BaseHeader.Flags.SH bit of the UserMessage packet is set. This bit is set when the SessionHeader is piggy-backed onto a UserMessage packet instead of sending it in a stand-alone SessionAck packet.

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 2.2.20.

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 2.2.20 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

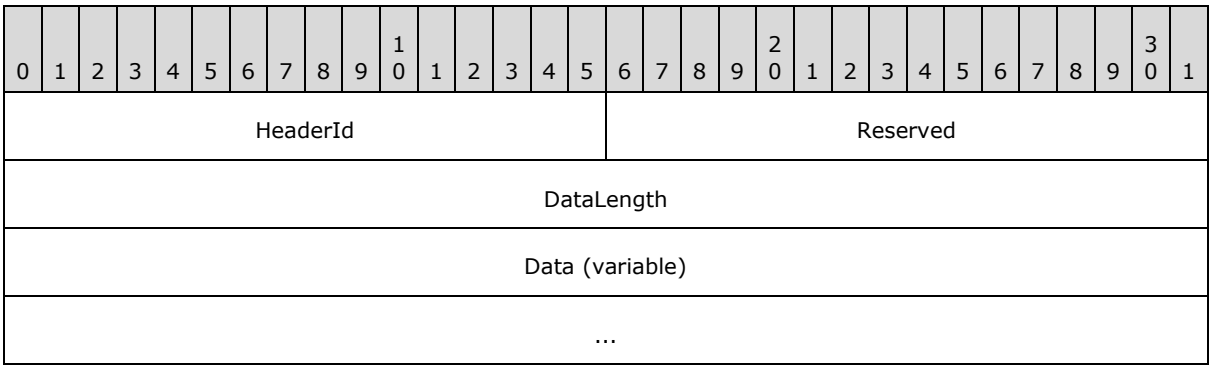
A binary message represents a message being received that was originally sent over the MSMQ: Binary Reliable Messaging Protocol [MS-MQQB]. The **UserHeader.Flags.AH** bit MUST NOT be set, and the SRMPEnvelopeHeader (section 2.2.5.1.2.1) and the CompoundMessageHeader (section 2.2.5.1.2.2) MUST NOT be present in the UserMessage (section 2.2.5.1).

2.2.5.1.2 SRMP Message

An SRMP message represents a message being received that was originally sent over HTTP. The **UserHeader.Flags.AH** bit MUST be set, and the SRMPEnvelopeHeader (section 2.2.5.1.2.1) and the CompoundMessageHeader (section 2.2.5.1.2.2) MUST be present in the UserMessage (section 2.2.5.1).

2.2.5.1.2.1 SRMPEnvelopeHeader

The SRMPEnvelopeHeader contains information about the SOAP envelope used to send the original message over HTTP. This header MUST be present only if UserHeader.Flags.AH is set.



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

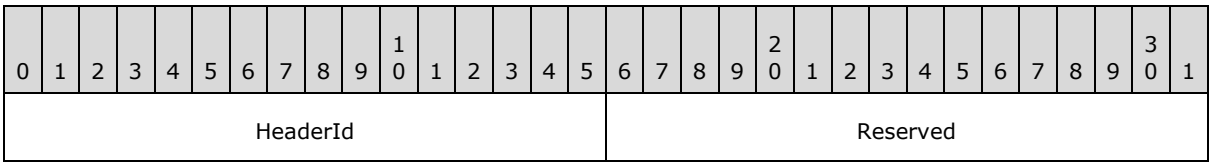
Reserved (2 bytes): A **USHORT**, as specified in [MS-DTYP], that MUST be ignored.

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 only if **UserHeader.Flags.AH** is set.



HTTPBodySize
MsgBodySize
MsgBodyOffset
Data (variable)
...

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.

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), such as DeadLetterHeader (section 2.2.5.4), SubqueueHeader (section 2.2.5.3), and ExtendedAddressHeader (section 2.2.5.5).<5>

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
HeaderSize																															
RemainingHeadersSize																															
Flags																Reserved															

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

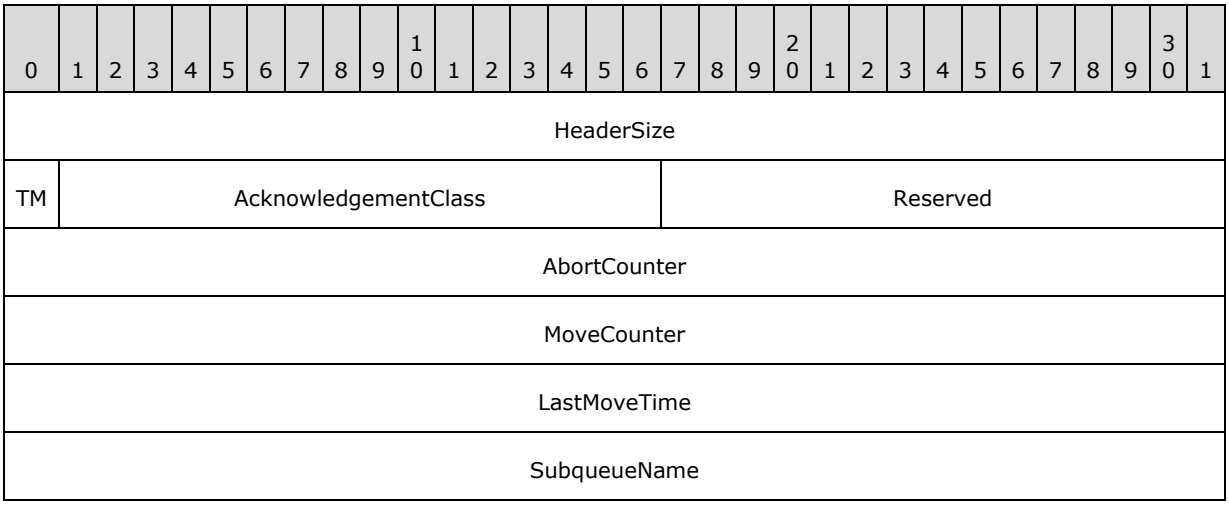
Where the bits are defined as:

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.
SQ	Indicates whether the Message Packet Structure contains a SubqueueHeader. MUST be set to 1.
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	Indicates whether the Message Packet Structure contains an ExtendedAddressHeader. MUST be set to 1.
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 following.<6> This header MUST be ignored if its **SubqueueName** field is an empty string.



...
...
...
...
...
...
...
(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 2.2.18.1.6.

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 3.1.6.1.

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.

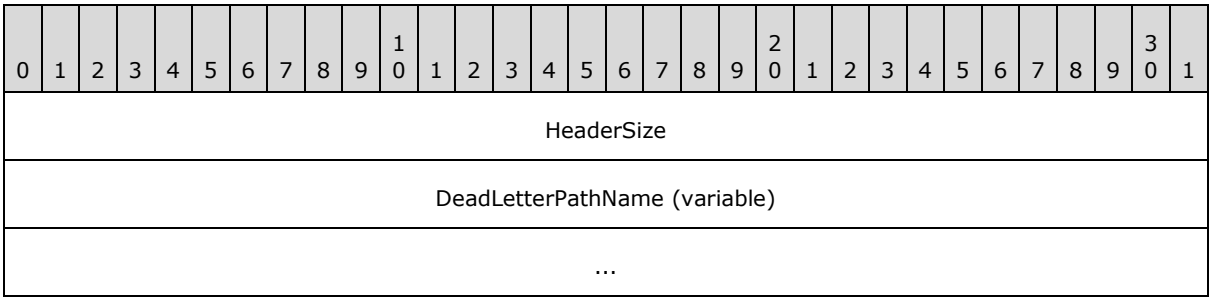
LastMoveTime (4 bytes): A **ULONG**, as specified in [MS-DTYP], that specifies the local time of the most recent move of the message. The time is specified as the number of milliseconds elapsed since midnight of January 1, 1970. If the message has never been moved, this value is 0. See section 3.1.4.10.

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 to 0.

2.2.5.4 DeadLetterHeader

The DeadLetterHeader specifies the path of an application-specified dead-letter queue.<7>

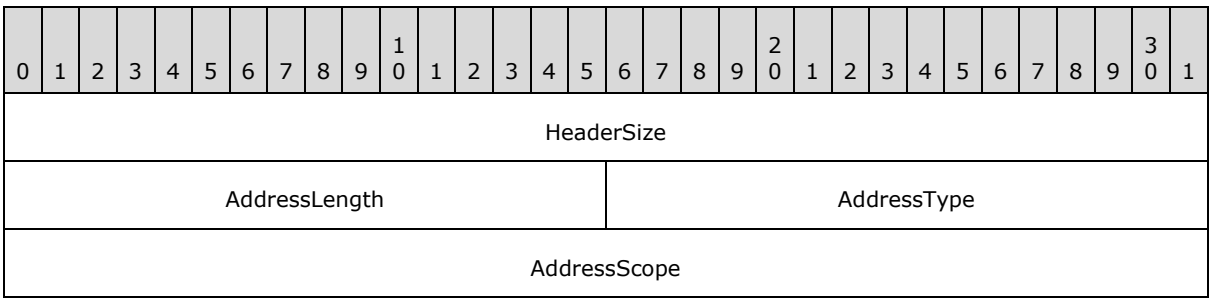


HeaderSize (4 bytes): A **ULONG**, as specified in [MS-DTYP], that MUST be set to the total size in bytes of the DeadLetterHeader.

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.<8> This header MUST be ignored if **AddressType** is 0x0000.



Address
...
...
...

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
0x0000	This header MUST be ignored.
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 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 it indicates 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.

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

stBinarySecondSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the second section of the Binary Message packet from beyond the end of the MessagePropertiesHeader in the UserMessage to the end of the packet.

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

stSrmpSecondSection: The **pSectionBuffer** element of the **SectionBuffer** structure contains the second section of the SRMP Message packet from beyond the end of the CompoundMessageHeader in the UserMessage to the end of the packet.

2.2.8 XACTUOW

The **XACTUOW** structure ([MS-MQMQ] section 2.2.18.1.8) uniquely identifies the unit of work (UOW) for a transactional operation. For an external transaction, this value MUST be acquired from the transaction coordinator. For an internal transaction, a client MUST create a unique random value for each transaction.<9>

2.3 Directory Service Schema Elements

This protocol uses ADM elements specified in section 3.1.1. A subset of these elements can be published in a **directory**. This protocol accesses the directory using the algorithm specified in [MS-MQDSSM] and using LDAP [MS-ADTS]. The Directory Service schema elements for ADM elements published in the directory are defined in [MS-MQDSSM] section 2.4.

3 Protocol Details

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 facilitate the explanation of 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.

The abstract data model for this protocol comprises elements that are private to this protocol and others that are shared between multiple MSMQ protocols that are co-located at a common queue manager. The shared abstract data model is defined in [MS-MQDMPR] section 3.1.1, and the relationship between this protocol, a queue manager, and other protocols that share a common queue manager is described in [MS-MQSO].

Section 3.1.1.1 details the elements from the shared data model that are manipulated by this protocol, and sections 3.1.1.2 through 3.1.1.4 detail the data model elements that are private to this protocol.

3.1.1.1 Shared Data Elements

This protocol manipulates the following abstract data model elements from the shared abstract data model defined in [MS-MQDMPR] section 3.1.1.

QueueManager: as defined in [MS-MQDMPR] section 3.1.1.1.

Queue: as defined in [MS-MQDMPR] section 3.1.1.2.

Message: as defined in [MS-MQDMPR] section 3.1.1.12.

OpenQueueDescriptorCollection: as defined in [MS-MQDMPR] section 3.1.1.2.

OpenQueueDescriptor: as defined in [MS-MQDMPR] section 3.1.1.16.

Cursor: as defined in [MS-MQDMPR] section 3.2.

Transaction: as defined in [MS-MQDMPR] section 3.1.1.14.

3.1.1.2 PendingRequestEntry

The PendingRequestEntry data element encapsulates a pending request to peek or receive a message from an open queue.

3.1.1.2.1 Attributes

RequestId: The request ID, supplied by the client.

LookupIdentifier: The lookup identifier of a Message associated with the request.

QueueContextHandle: An RPC context handle corresponding to an open queue, as defined by **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)**.

TimeStamp: A 32-bit unsigned integer that represents the time, in milliseconds, at which the client request was received.

3.1.1.3 PendingRequestTable

The PendingRequestTable data element represents a hash table that contains references to instances of PendingRequestEntry data elements keyed on {**PendingRequestEntry.RequestId**, **PendingRequestEntry.QueueContextHandle**}.

3.1.1.4 Message

The Message element extends the base Message element as defined in [MS-MQDMPR] section 3.1.1.10.

3.1.1.4.1 Attributes

The server MUST maintain private state for each Message object in addition to the state described for the **Message** element in [MS-MQDMPR] section 3.1.1.10. The following additional attributes are used to reference this private state:

Type: The type of the message packet, either binary or SRMP.

Offset: The offset and byte size of the **message headers**, message body, and **message trailers**.

3.1.2 Timers

The Message Queuing (MSMQ): Queue Manager Remote Read protocol MUST maintain the following timers, described in the following sections:

- RPC Call Timeout Timer (section 3.1.2.1)
- Pending Request Cleanup Timer (section 3.1.2.2)

3.1.2.1 RPC Call Timeout Timer

This protocol uses nondefault 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 300,000 milliseconds,<10> 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)**

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.

3.1.2.2 Pending Request Cleanup Timer

This timer regulates the amount of time that the protocol waits before removing expired entries from the PendingRequestTable. 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). This timer is set when a PendingRequestEntry object is added to the PendingRequestTable. The duration of this timer MUST be set based on the system configuration, which is implementation-dependent.<11>

3.1.3 Initialization

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

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	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	Opens a queue. Opnum: 2
R_CloseQueue	Closes a queue. Opnum: 3
R_CreateCursor	Opens a cursor on a queue. Opnum: 4
R_CloseCursor	Closes a cursor. Opnum: 5
R_PurgeQueue	Deletes all messages in a queue. Opnum: 6
R_StartReceive	Initiates a Receive or Peek request on the queue. Opnum: 7
R_CancelReceive	Cancels a pending Receive request. Opnum: 8
R_EndReceive	Finishes a Receive request. Opnum: 9
R_MoveMessage	Moves a message between two queues.

Method	Description
	Opnum: 10
R_OpenQueueForMove	Opens a queue to be a destination for a move operation. Opnum: 11
R_QMEnlistRemoteTransaction	Enlists in a transaction on a remote machine. Opnum: 12
R_StartTransactionalReceive	Initiates a transactional receive request on the queue. Opnum: 13
R_SetUserAcknowledgementClass	Changes the acknowledgment class for a message in a queue. Opnum: 14
R_EndTransactionalReceive	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 since it does not affect interoperability.<12>

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 default port number used is 2103. If this port is already in use, the server SHOULD increment the port number by 11 until an unused port is found.

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.<13>

```
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].

As specified in section 3.1.3, this protocol configures a fixed listening endpoint at an RPC port number which may vary. This method returns the RPC port number determined at server initialization time.

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)**
- **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.
- **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 **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)** value, which 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] QUEUE_CONTEXT_HANDLE_SERIALIZE* 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 (section 2.2.7)** that identifies the queue to open. NULL is invalid for this parameter. The valid values for the **m_qft** field are QUEUE_FORMAT_TYPE_PUBLIC, QUEUE_FORMAT_TYPE_PRIVATE, QUEUE_FORMAT_TYPE_DIRECT, QUEUE_FORMAT_TYPE_MACHINE, and QUEUE_FORMAT_TYPE_SUBQUEUE.

dwAccess: Specifies the requested type of access to the queue. The required *dwAccess* value for each event is specified in each of the corresponding events. If no requirement is listed, any *dwAccess* value is accepted.

Value	Meaning
RECEIVE_ACCESS	The returned **QUEUE_CONTEXT_HANDLE_SERIALIZE** can be used in the

Value	Meaning
0x00000001	**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 **QUEUE_CONTEXT_HANDLE_SERIALIZE** 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. The following values are valid for this parameter:

Value	Meaning
MQ_DENY_NONE 0x00000000	Permits multiple **QUEUE_CONTEXT_HANDLE_SERIALIZE** handles to the queue to be opened concurrently.
MQ_DENY_SHARE 0x00000001	Permits a single **QUEUE_CONTEXT_HANDLE_SERIALIZE** 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 GUID that uniquely identifies the client. When the queue manager acts as the client, the queue manager sets this value to **QueueManager.Identifier**. The server SHOULD ignore this parameter. The server MAY use this parameter to impose a limit on the number of unique callers.<14> NULL is invalid for this parameter.

fNonRoutingServer: If the client is configured to operate in the role of an **MSMQ routing server**, this parameter MUST be set to FALSE (0x00000000); otherwise, it MUST be set to TRUE (0x00000001).<15> 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.<16>

Minor: MUST be set by the client to an implementation-specific Minor Version number of the client. SHOULD be ignored by the server.<17>

BuildNumber: MUST be set by the client to an implementation-specific Build Number of the client. SHOULD be ignored by the server.<18>

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.<19>

Name	Value
False	0x00000000

Name	Value
True	0x00000001

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

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 output parameter values when any failure **HRESULT** is thrown.

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

- If any of the input parameter values is invalid, throw MQ_ERROR_INVALID_PARAMETER (0xC00E0006).
- Look up the queue name in the **QueueManager.QueueCollection**. If not found, throw MQ_ERROR_QUEUE_NOT_FOUND (0xC00E0003).
- Generate an Open Queue event, as specified in [MS-MQDMPR] section 3.1.7.1.5, with the following inputs:
  - *iFormatName* := *pQueueFormat*
  - *iRequiredAccess* := If *dwAccess* is RECEIVE_ACCESS then **QueueAccessType.ReceiveAccess** else **QueueAccessType.PeekAccess**.
  - *iSharedMode* := If *dwShareMode* is MQ_DENY_NONE then **QueueShareMode.DenyNone** else **QueueShareMode.DenyReceive**.
- If **rStatus** is MQ_OK (0x00000000):
  - Set *pphContext* to **rOpenQueueDescriptor.Handle**

3.1.4.3 R_CloseQueue (Opnum 3)

The **R_CloseQueue** method closes a **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)** 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] QUEUE_CONTEXT_HANDLE_SERIALIZE* pphContext
);
```

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

pphContext: MUST be set by the client to the **QUEUE_CONTEXT_HANDLE_SERIALIZE** 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. If the server returns MQ_OK, it 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:

- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *pphContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**.
- Generate a Close Queue ([MS-MQDMPR] section 3.1.7.1.6) event with the following inputs:
  - *iQueueDesc* := The found **OpenQueueDescriptor** ADM element instance.
- Find all entries in the PendingRequestTable (section 3.1.1.3) that contain the *pphContext* parameter, and remove these entries.
- Set the *pphContext* parameter to NULL.
- Return MQ_OK (0x00000000).

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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContext,
  [out] DWORD* phCursor
);
```

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

phContext: MUST be set by the client to the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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:

- Find the corresponding **OpenQueueDescriptor** instance by comparing *pphContext* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- If not found, return a failure **HRESULT**.
- Generate an Open Cursor event, as specified in [MS-MQDMPR] section 3.1.7.1.1, with the following inputs:
  - iQueueDesc := The found **OpenQueueDescriptor** instance.
- Set *phCursor* to **rCursor.Handle**.
- Return MQ_OK (0x00000000).

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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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 (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:

- Find the corresponding **OpenQueueDescriptor** instance by comparing *phContext* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- If the **OpenQueueDescriptor** instance is found, then find the corresponding **Cursor** instance by comparing *hCursor* with **Cursor.Handle** in **OpenQueueDescriptor.CursorCollection**.
- If not found, return a failure **HRESULT**.
- Generate a Close Cursor event, as specified in [MS-MQDMPR] section 3.1.7.1.2, with the following inputs:
  - iCursor := The found **Cursor** instance.
- Return MQ_OK (0x00000000).

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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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:

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

When processing this call, the server MUST:

- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**, and perform no further actions. Otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.

- If *queueDesc*.**AccessType** is **QueueAccessType.ReceiveAccess**:
  - Generate a Purge Queue ([MS-MQDMPR] section 3.1.7.1.7) event with the following inputs:
    - *iQueue* := *queueDesc*.**QueueReference**
  - Return MQ_OK (0x00000000), and perform no further actions.
- Return STATUS_ACCESS_DENIED (0xC0000022).

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* parameter 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: if *LookupId* is set to zero and *hCursor* is set to zero, the first unlocked message in the queue can be returned. The *ulAction* parameter further specifies whether the first message is to be received or peeked.

The *ppPacketSections* parameter is the address of one or more pointers to one or more **SectionBuffer (section 2.2.6)** structures. The **pSectionBuffer** member of the first **SectionBuffer** structure 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** member of the **SectionBuffer** structure.

```
HRESULT R_StartReceive(
  [in] handle_t hBind,
  [in] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContext,
  [in] ULONGLONG LookupId,
  [in] DWORD hCursor,
```

```
  [in] DWORD ulAction,
  [in] DWORD ulTimeout,
  [in] DWORD dwRequestId,
  [in] DWORD dwMaxBodySize,
  [in] DWORD dwMaxCompoundMessageSize,
  [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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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* parameter to a nonzero value, the valid values for other parameters are as follows:

- *ulTimeout* set to 0x00000000.
- *hCursor* set to 0x00000000.
- *ulAction* 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 client sets the *LookupId* parameter to 0x0000000000000000, all of the preceding values of the *ulAction* parameter are invalid.

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 the *hCursor* parameter to a nonzero value, the valid values for other parameters are as follows:

- *LookupId* set to 0x0000000000000000
- *ulAction* set to one of the following:
  - MQ_ACTION_RECEIVE
  - MQ_ACTION_PEEK_CURRENT
  - MQ_ACTION_PEEK_NEXT

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

Type / Value	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 0x00000000, read and remove the message from the front of the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000.
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 0x00000000, read the message at the front of the queue but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000.
MQ_ACTION_PEEK_NEXT 0x80000001	If *hCursor* is nonzero, advance the cursor to the next position and read the message, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000. ▪ *hCursor* set to a nonzero cursor handle obtained from the **R_CreateCursor** method.
MQ_LOOKUP_PEEK_CURRENT 0x40000010	Read the message specified by *LookupId*, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.
MQ_LOOKUP_PEEK_NEXT 0x40000011	Read the message following the message specified by *LookupId*, but do not remove it. The valid values for other parameters are as follows:

Type / Value	Meaning
	▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.
MQ_LOOKUP_PEEK_PREV 0x40000012	Read the message preceding the message specified by *LookupId*, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Read the message specified by *LookupId*, and remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.
MQ_LOOKUP_RECEIVE_NEXT 0x40000021	Read the message following the message specified by *LookupId*, and remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.
MQ_LOOKUP_RECEIVE_PREV 0x40000022	Read the message preceding the message specified by *LookupId*, and remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.

If *hCursor* is 0x00000000 and *LookupId* is 0x0000000000000000, the valid values for the *ulAction* parameter are as follows:

- MQ_ACTION_RECEIVE
- MQ_ACTION_PEEK_CURRENT

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

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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** until a call to either **R_EndReceive** or **R_CancelReceive** with the same *dwRequestId* parameter value has been completed.

dwMaxBodySize: MUST be set by the client to the maximum size, in bytes, of the message body to be returned. The server SHOULD ignore this parameter when the message is not a Binary Message (section 2.2.5.1.1).

dwMaxCompoundMessageSize: MUST be set by the client to the maximum size, in bytes, of the CompoundMessageHeader. The server SHOULD ignore this parameter when the message is not an SRMP Message (section 2.2.5.1.2).

pdwArriveTime: The server MUST set this value to the time that the message was added to the queue ([MS-MQDMPR] section 3.1.7.3.1), expressed as the number of seconds elapsed since midnight 00:00:00.0, January 1, 1970 Coordinated Universal Time (UTC).

pSequenceId: The server MUST set this parameter to the least significant 7 bytes of the **Message.LookupIdentifier** of the message that is received by this request.

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:

- Create two local variables of type **DWORD** called *maxMessageSize* and *actualMessageSize*. Assign the following values to these variables:

  If the message is a Binary Message:

  - *maxMessageSize* := *dwMaxBodySize*
  - *actualMessageSize* := message packet body size

  If the message is an SRMP Message:

  - *maxMessageSize* := **dwMaxCompoundMessageSize**
  - *actualMessageSize* := size in bytes of **CompoundMessageHeader**

- If the value of *maxMessageSize* is greater than or equal to *actualMessageSize*, 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 *maxMessageSize* is less than *actualMessageSize*, the array MUST contain a first element as follows:
  - **SectionType** MUST be set to one of the following:
    - stBinaryFirstSection if the message packet is a binary packet.
    - stSrmpFirstSection if the message packet is an SRMP packet.
  - **pSectionBuffer** MUST contain the **message packet headers** concatenated with the first *maxMessageSize* bytes of the message body.
  - **SectionSizeAlloc** MUST be set to the message packet header size plus *actualMessageSize*.
  - **SectionSize** MUST be set to the size of the **pSectionBuffer**.
- If the value of *maxMessageSize* is less than *actualMessageSize* 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.
- For the first element in this array, the **pSectionBuffer** member points to a Message Packet Structure (section 2.2.5). Within this structure, set **UserMessage.BaseHeader.TimeToReachQueue** to **UserHeader.SentTime** + **UserMessage.BaseHeader.TimeToReachQueue**.

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 output parameter values when any failure HRESULT is returned.

Return value/code	Description
0x00000000 MQ_OK	
0xC00E0007 MQ_ERROR_INVALID_HANDLE	
0xC00E001B MQ_ERROR_IO_TIMEOUT	
0xC00E0088 MQ_ERROR_MESSAGE_NOT_FOUND	

Return value/code	Description
0xC00E001D MQ_ERROR_MESSAGE_ALREADY_RECEIVED	
0xC00E0008 MQ_ERROR_OPERATION_CANCELLED	
0xC00E0006 MQ_ERROR_INVALID_PARAMETER	

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:

- If any of the input parameter values is invalid, return MQ_ERROR_INVALID_PARAMETER (0xC00E0006).
- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager**.
- If not found, return a failure **HRESULT**, and perform no further actions; otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.
- If *hCursor* is a nonzero value, find the corresponding **Cursor** ADM element instance by comparing *hCursor* with **Cursor.Handle** for all **Cursor** ADM element instances maintained by the local **QueueManager** ADM element. If not found, or the **Cursor** ADM element instance has previously been closed by a call to the **R_CloseCursor** (section 3.1.4.5) method, return STATUS_INVALID_HANDLE (0xC0000008); otherwise, assign the found **Cursor** ADM element instance to the local variable *localCursor*.
- If the *ulAction* parameter is MQ_ACTION_RECEIVE, perform the following steps:
  - Create a new PendingRequestEntry (section 3.1.1.2) ADM element instance with:
    - The **RequestId** ADM attribute set to the *dwRequestId* parameter.
    - The **QueueContextHandle** ADM attribute set to the *phContext* parameter.
    - The **LookupIdentifier** ADM attribute set to zero.
    - The **TimeStamp** ADM attribute set to the current system time, in milliseconds, since the operating system was started.
  - The server MUST create a new instance of the Pending Request Cleanup Timer (section 3.1.2.2) associated with the new **PendingRequestEntry** ADM element instance and MUST start it.
  - Add the new **PendingRequestEntry** ADM element instance to the PendingRequestTable (section 3.1.1.3) ADM element.
  - Generate a Dequeue Message Begin ([MS-MQDMPR] section 3.1.7.1.11) event with the following inputs:

- *iQueueDesc* := *queueDesc*
- *iTimeout* := *ulTimeout*
- *iCursor* := *localCursor* only if *hCursor* is a nonzero value
- *iTag* := *dwRequestId*

- If the *rStatus* value returned from the Dequeue Message Begin event is MQ_OK (0x00000000), the server MUST set the **LookupIdentifier** ADM attribute of the new **PendingRequestEntry** ADM element instance to *rMessage*.**LookupIdentifier**.

- If the *ulAction* parameter is MQ_ACTION_PEEK_CURRENT, generate a Peek Message ([MS-MQDMPR] section 3.1.7.1.15) event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iTimeout* := *ulTimeout*
  - *iCursor* := *localCursor* only if *hCursor* is a nonzero value
- If the *ulAction* parameter is MQ_ACTION_PEEK_NEXT, generate a Peek Next Message ([MS-MQDMPR] section 3.1.7.1.14) event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iTimeout* := *ulTimeout*
  - *iCursor* := *localCursor*
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_CURRENT, generate a Read Message By Lookup Identifier ([MS-MQDMPR] section 3.1.7.1.13) event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*
  - *iPeekOperation* := True
  - *iLookupOperation* := **MessageSeekAction**.**SeekCurrent**
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_NEXT, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*
  - *iPeekOperation* := True
  - *iLookupOperation* := **MessageSeekAction**.**SeekNext**
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_PREV, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*

- *iPeekOperation* := True
- *iLookupOperation* := **MessageSeekAction**.**SeekPrev**

- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_CURRENT, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction**.**SeekCurrent**
  - *iTwoPhaseRead* := True
- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_NEXT, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction**.**SeekNext**
  - *iTwoPhaseRead* := True
- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_PREV, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := *LookupId*
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction**.**SeekPrev**
  - *iTwoPhaseRead* := True

If the *rStatus* value returned from the preceding events is MQ_OK (Ox00000000), the server MUST:

- Use *rMessage* to 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 *pdwArriveTime* to *rMessage*.**ArrivalTime**.

Return *rStatus*.

3.1.4.8 R_CancelReceive (Opnum 8)

The **R_CancelReceive** method cancels a pending call to the **R_StartReceive (section 3.1.4.7)** method or the **R_StartTransactionalReceive (section 3.1.4.13)** method. 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** method provides a way for the client to cancel a blocked request.

```
HRESULT R_CancelReceive(
  [in] handle_t hBind,
  [in] QUEUE_CONTEXT_HANDLE_NOSERIALIZE 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** used in the corresponding call to the **R_StartReceive** method that is to be canceled. The handle MUST have been returned by the server in the *pphQueue* output parameter of a prior call to the **R_OpenQueue (section 3.1.4.2)** method and MUST NOT have been previously closed through a call to the **R_CloseQueue (section 3.1.4.3)** method. 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 the **R_StartReceive** method or the **R_StartTransactionalReceive** method. This parameter acts as an identifier to correlate an **R_CancelReceive** method call to an **R_StartReceive** or an **R_StartTransactionalReceive** method 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:

- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**, and perform no further actions; otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.
- Generate a Cancel Waiting Message Read Request ([MS-MQDMPR] section 3.1.7.1.17) event with the following inputs:
  - *iQueue* := *queueDesc*.**QueueReference**
  - *iTag* := *dwRequestId*
- If the Cancel Waiting Message Read Request event returns an error, return a failure **HRESULT**, and perform no further actions.
- Remove the PendingRequestEntry (section 3.1.1.2) ADM element instance referenced by the {*phContext*, *dwRequestId*} key pair from the PendingRequestTable (section 3.1.1.3) ADM element.

- Respond to the pending **R_StartReceive** or **R_StartTransactionalReceive** method request with 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** 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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** used in the corresponding call to the **R_StartReceive** method. 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** method request. The following table lists possible values.

Value	Meaning
RR_ACK 0x00000002	The client acknowledges that the message packet was received 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 received successfully. The server MUST keep the message in the queue 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 the **R_StartReceive** method. This parameter acts as an identifier to correlate an **R_EndReceive** method call to an **R_StartReceive** method 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:

- If the queue referenced by the *phContext* parameter handle has no PendingRequestEntry (section 3.1.1.2) ADM element instance in its PendingRequestTable (section 3.1.1.3) ADM element, return MQ_ERROR_INVALID_HANDLE (0xC00E0007).
- Look up the **PendingRequestEntry** ADM element instance referenced by the {*phContext*, *dwRequestId*} key pair in the **PendingRequestTable** ADM element. If a match is not found on the {*phContext*, *dwRequestId*} key pair, return MQ_ERROR_INVALID_PARAMETER (0xC00E0006). Otherwise, remove the **PendingRequestEntry** ADM element instance from the **PendingRequestTable** ADM element, and cancel the associated instance of Pending Request Cleanup Timer (section 3.1.2.2).
- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**, and perform no further actions; otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.
- Find the corresponding **Message** ADM element instance by searching **OpenQueueDescriptor.QueueReference.MessagePositionCollection** for a **MessagePosition** ADM element instance where **MessagePosition.MessageReference.LookupIdentifier** equals the **LookupIdentifier** ADM attribute of the **PendingRequestEntry** ADM element instance referenced by {*phContext*, *dwRequestId*}. The corresponding **Message** ADM element instance is referred to by the **MessageReference** ADM attribute of the **MessagePosition** ADM element instance where the match was found.
- If not found, return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
- Set *rStatus* to the result of a Dequeue Message End ([MS-MQDMPR] section 3.1.7.1.12) event with the following inputs:
  - *iQueueDesc* := *queueDesc*.
  - *iMessage* := The found **Message** ADM element instance.
  - *iDeleteMessage* := True if the *dwAck* parameter is equal to RR_ACK and false if the *dwAck* parameter is equal to RR_NACK.
- Return *rStatus*.

3.1.4.10 R_MoveMessage (Opnum 10)

The **R_MoveMessage** method moves a message from one queue to another.<20> 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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContextFrom,
  [in] ULONGLONG ullContextTo,
  [in] ULONGLONG LookupId,
  [in] XACTUOW* 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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.

ullContextTo: MUST be set by the client to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** 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 as a pointer to a transaction identifier or to a zero value **XACTUOW**. If the destination queue is not a transactional queue, this value MUST be a pointer to a zero value **XACTUOW**. If the value of the field is not zero, 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 provides both transactional and non-transactional operations. When using a 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. The non-transactional operation moves a message from the source queue to the destination queue without enlisting in a transaction.

When processing this call, the server MUST:

- Find the corresponding **OpenQueueDescriptor** instance for the source queue by comparing *phContextFrom* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager** and then declare and set **iSourceQueueDescriptor** to the instance.
- If not found, return a failure **HRESULT**.

- Find the corresponding **OpenQueueDescriptor** instance for the destination queue by comparing *ullContextTo* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager** and then declare and set **iDestinationQueueDescriptor** to the instance.
- If not found or if **iDestinationQueueDescriptor.AccessType** is not **QueueAccessType.MoveAccess**, then return MQ_ERROR_INVALID_HANDLE (0xC00E0007).
- If none of the following conditions are met, return STATUS_INVALID_PARAMETER (0xC000000D):
  - **iSourceQueueDescriptor** is part of the collection **iDestinationQueueDescriptor.QueueReference.SubqueueCollection**.
  - **iDestinationQueueDescriptor** is part of collection **iSourceQueueDescriptor.QueueReference.SubqueueCollection**.
  - **iSourceQueueDescriptor.QueueReference.Pathname** and **iDestinationQueueDescriptor.QueueReference.Pathname** have the same parent queue pathname. The parent queue pathname MUST be formed by removing the subqueue portion from the pathname and the preceding ";", as defined in [MS-MQMQ] section 2.1.1.
- If the method is provided with a nonzero *pTransactionId* and if **iDestinationQueueDescriptor.QueueReference.Transactional** is False, return MQ_ERROR_TRANSACTION USAGE (0xC00E0050).
- Find the corresponding **Message** instance by comparing **PendingRequestEntry.LookupIdentifier** with **MessagePosition.MessageReference.Identifier** in the **iSourceQueueDescriptor.QueueReference.MessagePositionCollection** and then declare and set **iFoundMessage** to the instance.
- If not found, then return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
- If the message is already part of another transaction, return MQ_ERROR_MESSAGE_LOCKED_UNDER_TRANSACTION (0xC00E009C).
- If the method is provided with a nonzero *pTransactionId*, find the corresponding **Transaction** instance by comparing *pTransactionId* with **Transaction.Identifier** for all **Transaction** instances in **QueueManager.TransactionCollection** and then declare and set **iFoundTransaction** to the instance.
- If not found, return MQ_ERROR_TRANSACTION_SEQUENCE (0xC00E0051).
- Generate a Move Message event, as specified in [MS-MQDMPR] section 3.1.7.1.16, with the following inputs:
  - *iMessagePos* := **iFoundMessage.MessagePositionReference**.
  - *iTargetQueue* := **iDestinationQueueDescriptor.QueueReference**.
  - If there is a transaction, *iTransaction* := **iFoundTransaction**.
- Return MQ_OK (0x00000000).

3.1.4.11 R_OpenQueueForMove (Opnum 11)

The **R_OpenQueueForMove** method opens the queue and returns a **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)** 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.<21>

```
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] QUEUE_CONTEXT_HANDLE_SERIALIZE* 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 MQ_DENY_NONE (0x00000000), which permits multiple **QUEUE_CONTEXT_HANDLE_SERIALIZE** handles to the queue to be opened concurrently.

pClientId: MUST be set by the client to a pointer to a valid GUID that uniquely identifies the client. When the queue manager acts as the client, the queue manager sets this value to **QueueManager.Identifier**. The server SHOULD ignore this parameter. This value MUST NOT be NULL.

fNonRoutingServer: If the client is configured to operate in the role of an MSMQ routing server, this parameter MUST be set to FALSE (0x00000000); otherwise, it MUST be set to TRUE (0x00000001).<22> 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.<23>

Minor: MUST be set by the client to an implementation-specific Minor Version number of the client. SHOULD be ignored by the server.<24>

BuildNumber: MUST be set by the client to an implementation-specific Build Number of the client. SHOULD be ignored by the server.<25>

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.<26>

Name	Value
FALSE	0x00000000
TRUE	0x00000001

pMoveContext: The server MUST set this parameter to a pointer to a **QUEUE_CONTEXT_HANDLE_SERIALIZE** 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 **QUEUE_CONTEXT_HANDLE_SERIALIZE** returned in *pphContext*.

pphContext: MUST be set by the server to a **QUEUE_CONTEXT_HANDLE_SERIALIZE**. A **QUEUE_CONTEXT_HANDLE_SERIALIZE** 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 **QueueManager.QueueCollection**. If not found, throw MQ_ERROR_QUEUE_NOT_FOUND (0xC00E0003).
- Generate an Open Queue event, as specified in [MS-MQDMPR] section 3.1.7.1.5, with the following inputs:
  - iFormatName := *pQueueFormat*
  - iRequiredAccess := **QueueAccessType.MoveAccess**
  - iSharedMode := If *dwShareMode* is MQ_DENY_NONE then **QueueShareMode.DenyNone** else **QueueShareMode.DenyReceive**.
- If rStatus is MQ_OK (0x00000000) then
  - Set *pphContext* to **rOpenQueueDescriptor.Handle**.

- Set *pMoveContext* to *pphContext*.

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.<27> 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 occur, depending on whether the transaction outcome is Commit or Rollback.

```
HRESULT R_QMEnlistRemoteTransaction(
  [in] handle_t hBind,
  [in] XACTUOW* 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.

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.<28>

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:

- Enlist into the transaction as specified in [MS-DTCO] section 3.5.4.3.
- Generate a Create Transaction event ([MS-MQDMPR] section 3.1.7.1.8) with the following inputs:
  - iTransactionIdentifier := *pTransactionId*
- Return MQ_OK (0x00000000).

3.1.4.13 R_StartTransactionalReceive (Opnum 13)

The **R_StartTransactionalReceive** method peeks or receives a message from the opened queue.<29> If a transaction identifier is provided, it receives a message inside the specified transaction.

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

If the **R_StartTransactionalReceive** method is invoked with a Receive action type, as specified in the *ulAction* parameter, the client MUST pair each call to the **R_StartTransactionalReceive** method with a call to the **R_EndTransactionalReceive (section 3.1.4.15)** method to complete the operation or to the **R_CancelReceive (section 3.1.4.8)** method to cancel the operation. The call to the **R_EndTransactionalReceive** method or the **R_CancelReceive** method is correlated to a call to the **R_StartTransactionalReceive** method 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 the **R_CancelReceive** method with a matching *dwRequestId* parameter to cancel the pending **R_StartTransactionalReceive** method request.

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

- LookupId: A nonzero *LookupId* parameter 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 the *LookupId* parameter 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 message immediately following it. For more details, see the description of the *ulAction* parameter.
- First: If the *LookupId* parameter is set to 0x0000000000000000 and *hCursor* is set to 0x00000000, the first unlocked message in the queue can 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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContext,
  [in] ULONGLONG LookupId,
  [in] DWORD hCursor,
  [in] DWORD ulAction,
  [in] DWORD ulTimeout,
  [in] DWORD dwRequestId,
```

```
  [in] DWORD dwMaxBodySize,
  [in] DWORD dwMaxCompoundMessageSize,
  [in] XACTUOW* 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** 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 the **R_OpenQueue (section 3.1.4.2)** method with the *dwAccess* parameter set to RECEIVE_ACCESS and MUST NOT have been closed through a prior call to the **R_CloseQueue (section 3.1.4.3)** method. NULL is invalid for this parameter.

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

If the client sets the *LookupId* parameter to a nonzero value, the valid values for other parameters are as follows:

- *ulTimeout* set to 0x00000000
- *hCursor* set to 0x00000000
- *ulAction* set to one of the following:
  - MQ_LOOKUP_PEEK_PREV (*pTransactionId* set to NULL)
  - MQ_LOOKUP_PEEK_CURRENT (*pTransactionId* set to NULL)
  - MQ_LOOKUP_PEEK_NEXT (*pTransactionId* set to NULL)
  - MQ_LOOKUP_RECEIVE_PREV
  - MQ_LOOKUP_RECEIVE_CURRENT
  - MQ_LOOKUP_RECEIVE_NEXT

If the client sets the *LookupId* parameter to 0x0000000000000000, all of the preceding values of the *ulAction* parameter are invalid.

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 the **R_CloseCursor (section 3.1.4.5)** method.

If the client sets the *hCursor* parameter to a nonzero value, the valid values for other parameters are as follows:

- *LookupId* set to 0x0000000000000000.
- *ulAction* set to one of the following:
  - MQ_ACTION_RECEIVE

- MQ_ACTION_PEEK_CURRENT (*pTransactionId* set to NULL)
- MQ_ACTION_PEEK_NEXT (*pTransactionId* set to NULL)

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

Type / Value	Meaning
MQ_ACTION_RECEIVE 0x00000000	If the *hCursor* parameter is nonzero, read and remove the message at the current cursor location from the queue, and advance thecursor. If the *hCursor* parameter is 0x00000000, read and remove the message from the front of the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000.
MQ_ACTION_PEEK_CURRENT 0x80000000	If the *hCursor* parameter is nonzero, read the message at the current cursor location, but do not remove it from the queue. If the *hCursor* parameter is 0x00000000, read the message at the front of the queue, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000. ▪ *pTransactionId* set to NULL.
MQ_ACTION_PEEK_NEXT 0x80000001	If the *hCursor* parameter is nonzero, advance the cursor to the next position, and read the message, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to 0x0000000000000000. ▪ *hCursor* set to a nonzero cursor handle obtained from the **R_CreateCursor** method. ▪ *pTransactionId* set to NULL.
MQ_LOOKUP_PEEK_CURRENT 0x40000010	Read the message specified by the *LookupId* parameter, but do not remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000. ▪ *pTransactionId* set to NULL.
MQ_LOOKUP_PEEK_NEXT 0x40000011	Read the message following the message specified by the *LookupId* parameter, but do not remove it. The valid values for other parameters are as follows:

Type / Value	Meaning
	▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000. ▪ *pTransactionId* set to NULL.
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 valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000. ▪ *pTransactionId* set to NULL.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Read the message specified by the *LookupId* parameter, and remove it from the queue. The valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* 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 valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* 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 valid values for other parameters are as follows: ▪ *LookupId* set to a nonzero value. ▪ *hCursor* set to 0x00000000. ▪ *ulTimeout* set to 0x00000000.

If the *hCursor* parameter is 0x00000000 and the *LookupId* parameter is 0x0000000000000000, the valid values for the *ulAction* parameter are as follows:

- MQ_ACTION_RECEIVE

- MQ_ACTION_PEEK_CURRENT (*pTransactionId* set to NULL)

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

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 the **R_EndTransactionalReceive** method or the **R_CancelReceive** method to correlate that call with the call to the **R_StartTransactionalReceive** method. The value MUST NOT be used in another **R_StartTransactionalReceive** method call on the same **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** until a call to either the **R_EndTransactionalReceive** method or the **R_CancelReceive** method with the same *dwRequestId* parameter value has been completed.

dwMaxBodySize: MUST be set by the client to the maximum size, in bytes, of the message body to be returned. The server SHOULD ignore this parameter when the message is not a Binary Message (section 2.2.5.1.1).

dwMaxCompoundMessageSize: MUST be set by the client to the maximum size, in bytes, of the CompoundMessageHeader (section 2.2.5.1.2.2). The server SHOULD ignore this parameter when the message is not an SRMP Message (section 2.2.5.1.2).

pTransactionId: Set to NULL or set by the client to a transaction identifier that was registered with the server through a prior call to the **R_QMEnlistRemoteTransaction (section 3.1.4.12)** method.

pdwArriveTime: The server MUST set this value to the time that the message was added to the queue ([MS-MQDMPR] section 3.1.7.3.1), expressed as the number of seconds elapsed since midnight 00:00:00.0, January 1, 1970 Coordinated Universal Time (UTC).

pSequenceId: The server MUST set this parameter to the lower 7 bytes of the **Message.LookupIdentifier** of the message that is received by this request.

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:

- Create two local variables of type **DWORD** called *maxMessageSize* and *actualMessageSize*. Assign the following values to these variables:

  If the message is a Binary Message:

  - *maxMessageSize* := *dwMaxBodySize*
  - *actualMessageSize* := message packet body size

  If the message is an SRMP Message:

  - *maxMessageSize* := *dwMaxCompoundMessageSize*
  - *actualMessageSize* := size in bytes of CompoundMessageHeader

- If the value of *maxMessageSize* is greater than or equal to *actualMessageSize*, the *ppPacketSections* parameter MUST contain a single element as follows:

- **SectionType** MUST be set to stFullPacket (0x00000000).
- The **SectionSize** and **SectionSizeAlloc** fields MUST be set to the message packet size.
- The **pSectionBuffer** field MUST contain the entire message packet.

- If the value of *maxMessageSize* is less than *actualMessageSize*, the array MUST contain a first element as follows:

  SectionType MUST be set to one of the following:

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

- If the value of *maxMessageSize* is less than *actualMessageSize* 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.
  - The **pSectionBuffer** field MUST contain the message packet trailers.
  - The **SectionSize** and the **SectionSizeAlloc** fields MUST be equal and MUST be set to the message packet trailers size.

- For the first element in this array, the **pSectionBuffer** field points to a Message Packet Structure (section 2.2.5). Within this structure, set **UserMessage.BaseHeader.TimeToReachQueue** to **UserHeader.SentTime** + **UserMessage.BaseHeader.TimeToReachQueue**.

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**s identically. The client MUST disregard all output parameter values when any failure **HRESULT** is returned.

Return value/code	Description
0x00000000 MQ_OK	
0xC00E0007 MQ_ERROR_INVALID_HANDLE	

Return value/code	Description
0xC00E0088 MQ_ERROR_MESSAGE_NOT_FOUND	
0xC00E001B MQ_ERROR_IO_TIMEOUT	
0xC00E0050 MQ_ERROR_TRANSACTION_USAGE	
0xC00E0008 MQ_ERROR_OPERATION_CANCELLED	
0xC00E0006 MQ_ERROR_INVALID_PARAMETER	

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:

- If any of the input parameter values is invalid, return MQ_ERROR_INVALID_PARAMETER (0xC00E0006).
- If the *pTransactionId* parameter is NULL:
  - Call the **R_StartReceive (section 3.1.4.7)** method with the following parameters:
    - *hBind* := *hBind*
    - *phContext* := *phContext*
    - *LookupId* := *LookupId*
    - *hCursor* := *hCursor*
    - *ulAction* := *ulAction*
    - *ulTimeout* := *ulTimeout*
    - *dwRequestId* := *dwRequestId*
    - *dwMaxBodySize* := *dwMaxBodySize*
    - *dwMaxCompoundMessageSize* := *dwMaxCompoundMessageSize*
    - *pdwArriveTime* := *pdwArriveTime*
    - *pSequenceId* := *pSequenceId*
    - *pdwNumberOfSections* := *pdwNumberOfSections*
    - *ppPacketSections* := *ppPacketSections*

- Return the result from the **R_StartReceive** method, and take no further action.

- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**, and perform no further actions; otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.
- If the *hCursor* parameter is a nonzero value, find the corresponding **Cursor** ADM element instance by comparing the *hCursor* parameter with **Cursor.Handle** for all **Cursor** ADM element instances maintained by the local **QueueManager** ADM element. If not found, or the **Cursor** ADM element instance has previously been closed by a call to the **R_CloseCursor** method, return STATUS_INVALID_HANDLE (0xC0000008).
- If *queueDesc*.**QueueReference**.**Transactional** is FALSE, the queue does not support transactional operations. Return MQ_ERROR_TRANSACTION_USAGE (0xC00E0050).
- If the *ulAction* parameter is MQ_ACTION_PEEK_CURRENT, MQ_ACTION_PEEK_NEXT, MQ_LOOKUP_PEEK_CURRENT, MQ_LOOKUP_PEEK_NEXT, or MQ_LOOKUP_PEEK_PREV, return MQ_ERROR_TRANSACTION_USAGE.
- Find the corresponding **Transaction** ADM element instance, referred to as **lpTransaction**, by comparing the *pTransactionId* parameter with **Transaction.Identifier** for all **Transaction** ADM element instances in **QueueManager.TransactionCollection**.
- If a **Transaction** ADM element instance cannot be found:
  - Generate a Create Transaction ([MS-MQDMPR] section 3.1.7.1.8) event with the following input:
    - *iTransactionIdentifier* := NULL
  - On return, set **lpTransaction** to *rTransaction*.
- If the *ulAction* parameter is MQ_ACTION_RECEIVE, perform the following steps:
  - Create a new PendingRequestEntry (section 3.1.1.2) ADM element instance with:
    - The **RequestId** ADM attribute set to the *dwRequestId* parameter.
    - The **QueueContextHandle** ADM attribute set to the *phContext* parameter.
    - The **LookupIdentifier** ADM attribute set to zero.
    - The **TimeStamp** ADM attribute set to the current system time, in milliseconds, since the operating system was started.
  - The server MUST create a new instance of the Pending Request Cleanup Timer (section 3.1.2.2) associated with the new **PendingRequestEntry** ADM element instance and MUST start it.
  - Add the new **PendingRequestEntry** ADM element instance to the PendingRequestTable (section 3.1.1.3) ADM element.
  - Generate a Dequeue Message Begin ([MS-MQDMPR] section 3.1.7.1.11) event with the following inputs:

- *iQueueDesc* := *queueDesc*
- *iTimeout* := *ulTimeout*
- *iCursor* := **Cursor** only if *hCursor* is a nonzero value
- *iTag* := *dwRequestId*
- *iTransaction* := **lpTransaction**

- If the *rStatus* value returned from the Dequeue Message Begin event is MQ_OK (0x00000000), the server MUST set the **LookupIdentifier** ADM attribute of the new **PendingRequestEntry** ADM element instance to *rMessage*.**LookupIdentifier**.

- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_CURRENT, generate a Read Message By Lookup Identifier ([MS-MQDMPR] section 3.1.7.1.13) event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := **LookupId**
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction.SeekCurrent**
  - *iTransaction* := **lpTransaction**
  - *iTwoPhaseRead* := True
- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_NEXT, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := **LookupId**
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction.SeekNext**
  - *iTransaction* := **lpTransaction**
  - *iTwoPhaseRead* := True
- If the *ulAction* parameter is MQ_LOOKUP_RECEIVE_PREV, generate a Read Message By Lookup Identifier event with the following inputs:
  - *iQueueDesc* := *queueDesc*
  - *iLookupId* := **LookupId**
  - *iPeekOperation* := False
  - *iLookupOperation* := **MessageSeekAction.SeekPrev**
  - *iTransaction* := **lpTransaction**
  - *iTwoPhaseRead* := True

- If the *rStatus* value returned from the preceding events is MQ_OK (Ox00000000), the server MUST:
  - Use *rMessage* to fill the *ppPacketSections* parameter array as specified in the *ppPacketSections* parameter description.
  - Set the *pdwArriveTime* parameter to **Message.ArrivalTime**.
- Return *rStatus*.

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.<30> This method MUST be called subsequent to calls to **R_StartTransactionalReceive** and **R_EndTransactionalReceive**, and before the transaction is committed or aborted.

```
HRESULT R_SetUserAcknowledgementClass(
  [in] handle_t hBind,
  [in] QUEUE_CONTEXT_HANDLE_NOSERIALIZE 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** 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.

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_TRANSACTION_USAGE (0xC00E0050)

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:

- Find the corresponding **OpenQueueDescriptor** instance by comparing *phContext* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- If not found, return a failure **HRESULT**.
- Find the corresponding **MessagePosition** instance by comparing *LookupId* with **MessagePosition.MessageReference.Identifier** in the **OpenQueueDescriptor.QueueReference.MessagePositionCollection**.
- If not found, then return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
- Find the corresponding **TransactionOperation** instance by comparing the **MessagePosition** with **Transaction.TransactionalOperationCollection.MessagePositionReference** in **QueueManager.TransactionCollection**.
- If not found, then return MQ_ERROR_TRANSACTION_USAGE (0xC00E0050).
- If *usClass* is not 0x0000, set the **DequeueReason** ADM attribute of the **TransactionOperation** instance to **NackReceiveRejected**, as specified in [MS-MQDMPR] section 3.1.1.12.
- Return MQ_OK (0x00000000).

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.<31>

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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContext,
  [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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** structure used in the corresponding call to the **R_StartTransactionalReceive** method. 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** method request. The following table lists possible values.

Value	Meaning
RR_ACK 0x00000002	The client acknowledges that the message packet was received successfully. The server MUST NOT remove the packet from the queue. The server removes the packet from the queue when the transaction is committed.
RR_NACK 0x00000001	The client acknowledges that the message packet was not received 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 the **R_StartTransactionalReceive** method. This parameter acts as an identifier to correlate an **R_EndTransactionalReceive** method call to an **R_StartTransactionalReceive** method 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 **HRESULT**s 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 do the following:

- If the queue referenced by the *phContext* parameter handle has no PendingRequestEntry (section 3.1.1.2) ADM element instance in its PendingRequestTable (section 3.1.1.3) ADM element, return MQ_ERROR_INVALID_HANDLE (0xC00E0007).
- Look up the **PendingRequestEntry** ADM element instance referenced by the {*phContext*, *dwRequestId*} key pair in the **PendingRequestTable** ADM element. If a match is not found on the {*phContext*, *dwRequestId*} key pair, return MQ_ERROR_INVALID_PARAMETER (0xC00E0006). Otherwise, remove the **PendingRequestEntry** ADM element instance from the **PendingRequestTable** ADM element, and cancel the associated instance of Pending Request Cleanup Timer (section 3.1.2.2).
- Find the corresponding **OpenQueueDescriptor** ADM element instance by comparing the *phContext* parameter with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** ADM element instances maintained by the local **QueueManager** ADM element.
- If not found, return a failure **HRESULT**, and perform no further actions; otherwise, assign the found **OpenQueueDescriptor** ADM element instance to the local variable *queueDesc*.
- Find the corresponding **Message** ADM element instance by searching **OpenQueueDescriptor.QueueReference.MessagePositionCollection** for a **MessagePosition** ADM element instance where **MessagePosition.MessageReference.LookupIdentifier** equals the **LookupIdentifier** ADM

attribute of the **PendingRequestEntry** ADM element instance referenced by {*phContext*, *dwRequestId*}. The corresponding **Message** ADM element instance is referred to by the **MessageReference** ADM attribute of the **MessagePosition** ADM element instance where the match was found.

- If not found, return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0088).
- Set *rStatus* to the result of a Dequeue Message End ([MS-MQDMPR] section 3.1.7.1.12) event with the following inputs:
  - *iQueueDesc* := *queueDesc*.
  - *iMessage* := The found **Message** ADM element instance.
  - *iDeleteMessage* := True if the *dwAck* parameter is equal to RR_ACK and false if the *dwAck* parameter is equal to RR_NACK.
  - *iTransactional* := True.
- Return *rStatus*.

3.1.5 Timer Events

3.1.5.1 Pending Request Cleanup Timer Event

When the Pending Request Cleanup Timer expires, for the PendingRequestEntry associated with this timer, the server MUST:

- Find the **OpenQueueDescriptor** instance by comparing *PendingRequestEntry.QueueContextHandle* with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- Find the corresponding **Message** instance by comparing **PendingRequestEntry.LookupIdentifier** with **MessagePosition.MessageReference.Identifier** in the **OpenQueueDescriptor.QueueReference.MessagePositionCollection**.
- Generate a Dequeue Message End event as specified in [MS-MQDMPR] section 3.1.7.1.12, with the following inputs:
  - iQueueDesc := **OpenQueueDescriptor**.
  - iMessage := **Message**.
  - iDeleteMessage := false.
- Remove the PendingRequestEntry from the PendingRequestTable and cancel the timer.

3.1.6 Other Local Events

3.1.6.1 RPC Failure Event

The event is received when RPC detects a connection failure with a client identified by a specific **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**.

While processing this event, the server MUST:

- Find the corresponding **OpenQueueDescriptor** instance by comparing the **QUEUE_CONTEXT_HANDLE_SERIALIZE** with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- If found then:
  - Find all **Cursor** instances maintained by the local **QueueManager** where **Cursor.OpenQueueDescriptorReference** equals the found **OpenQueueDescriptor**.
  - For each found **Cursor** instance:
    - Generate a Close Cursor event, as specified in [MS-MQDMPR] section 3.1.7.1.2, with the following inputs:
      - iCursor := **Cursor**.
  - Generate a Close Queue event, as specified in [MS-MQDMPR] section 3.1.7.1.6, with the following inputs:
    - iQueueDesc := **OpenQueueDescriptor**.

3.1.6.2 Queue Context Handles Rundown Routine

This event occurs on rundown of queue context handles of type **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** and **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**, as specified in [C706] section 5.1.6. The queue context handle being rundown is referred to as lpQueueContextHandle.

When processing this event, the server MUST:

- Find the corresponding **OpenQueueDescriptor** instance by comparing the lpQueueContextHandle being rundown with **OpenQueueDescriptor.Handle** for all **OpenQueueDescriptor** instances maintained by the local **QueueManager**.
- If not found, then return a failure HRESULT.
- Generate a Close Queue event, as specified in [MS-MQDMPR] section 3.1.7.1.6, with the following inputs:
  - iQueueDesc := the found **OpenQueueDescriptor** instance.
- For each PendingRequestEntry in PendingRequestTable where the **PendingRequestEntry.QueueContextHandle** is equal to the lpQueueContextHandle being rundown:
  - Search the **OpenQueueDescriptor.QueueReference.MessagePositionCollection** of the found **OpenQueueDescriptor** instance for a **MessagePosition** instance where **MessagePosition.MessageReference.Identifier** equals the **PendingRequestEntry.LookupIdentifier** of the current PendingRequestEntry instance.
  - Generate a Dequeue Message End event, as specified in [MS-MQDMPR] section 3.1.7.1.12, with the following inputs:
    - iQueueDesc := the found **OpenQueueDescriptor** instance.
    - iMessage := the Message referred to by the **MessagePosition.MessageReference** of the found **MessagePosition** instance.

- iDeleteMessage := false.
- Remove the PendingRequestEntry from the PendingRequestTable

- Set lpQueueContextHandle to NULL.
- Return MQ_OK (0x00000000).

3.2 RemoteRead Client Details

3.2.1 Abstract Data Model

Clients MUST maintain the following data elements:

- A **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)** associated with a queue.
- A table of cursor handles associated with a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)**.

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 client 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.
- The client MAY<32> call the **R_GetServerPort (section 3.1.4.1)** method by using the RPChandle 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<33> construct a new RPC binding handle to the server by using the RPC endpoint port determined in the previous step and replacing it 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.
- If the previous step returns an error code of EPT_S_NOT_REGISTERED (0x000006D9), the client SHOULD try instead to use the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol.
- The client MUST record the returned **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**.

3.2.4.2 Enlisting in a Transaction

The message queuing application MUST generate an Enlisting in a Transaction event before generating a Receive a Message (section 3.2.4.4) event, Move a Message (section 3.2.4.6) event, or Receive a Message by Using a Cursor (section 3.2.4.10) event, if the message is to be received or moved in a transaction context.

The message queuing application MUST specify the transaction identifier, and subsequent invocations of the Receive a Message event, Move a Message event, or Receive a Message by Using a Cursor event MUST be generated with the same 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:
  - The *pTransactionId* parameter set to the transaction identifier.
  - The *pQueueFormat* parameter set to the queue format name.
  - A transaction propagation token, obtained as specified in [MS-DTCO] section 3.3.4.3.

3.2.4.3 Peek a Message

The message queuing application MUST specify the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** RPC context handle of the queue from which to be read, 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 of the **R_StartReceive (section 3.1.4.7)** method with action type of Peek.

- The client MUST call the **R_StartReceive** method and MUST specify the following parameter values:
  - *phContext* set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** handle that has been returned by the server in the *pphQueue* output parameter of a prior call to the **R_OpenQueue (section 3.1.4.2)** method, which MUST NOT have been previously closed through a call to the **R_CloseQueue (section 3.1.4.3)** method. This value MUST NOT be NULL.
  - *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 specified by the message queuing application.
  - *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.
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** structure received in the *ppPacketSections* parameter, 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** of the queue from which to be read, 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 a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** RPC context handle that has been returned by the server in the *pphQueue* output parameter of a prior call to the **R_OpenQueue (section 3.1.4.2)** method, which MUST NOT have been

previously closed through a call to the **R_CloseQueue (section 3.1.4.3)** method. This value MUST NOT be NULL.

- *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 specified by the message queuing application.
- *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.

- Let *readAck* be a DWORD value initialized to RR_ACK (0x00000002).
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** structure received in the *ppPacketSections* parameter, as specified in section 3.1.4.7. If the message cannot be reconstructed, the client MUST set *readAck* to RR_NACK (0x00000001).
- If **R_StartReceive** was invoked with a Receive action type, as specified in the *ulAction* parameter, then the client MUST advise the server that the message has been received by the client by calling the **R_EndReceive (section 3.1.4.9)** method with the following parameter values.
  - *phContext* as in the call to **R_StartReceive**.
  - *dwAck* := *readAck*
  - *dwRequestId* set to the same value as in the call to **R_StartReceive**.
- If MQ_OK (0x00000000) is returned
  - The client MUST return the reconstructed message to the message queuing application.
- Else if the return value is not MQ_OK
  - The client MAY<34> return MQ_OK to the message queuing application.

3.2.4.4.2 Receive a Message with a Transaction

- The message queuing application MUST specify a transaction identifier for a Receive a Message With a Transaction event. If the message queuing application has not previously done so, it MUST enlist the server in a transaction by generating an Enlisting in a Transaction (section 3.2.4.2) event.
- The client MUST call the **R_StartTransactionalReceive (section 3.1.4.13)** method and MUST specify the following parameter values:
  - *phContext* set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** RPC context handle that has been returned by the server in the *pphQueue* output parameter of a prior call to the **R_OpenQueue (section 3.1.4.2)** method, which MUST NOT have been previously closed through a call to the **R_CloseQueue (section 3.1.4.3)** method. This value MUST NOT be NULL.
  - *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 specified by the message queuing application.
- *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 specified by the message queuing application.

- Let *readAck* be a DWORD value initialized to RR_ACK (0x00000002).
- The client MUST reconstruct the message from the **SectionBuffers (section 2.2.6)** structure received in the *ppPacketSections* parameter, as specified in **R_StartReceive (section 3.1.4.7)**. If the message cannot be reconstructed, the client MUST set *readAck* to RR_NACK (0x00000001).
- If **R_StartTransactionalReceive** was invoked with a Receive action type, as specified in the *ulAction* parameter, then the client MUST advise the server that the message has been received by the client by calling the **R_EndTransactionalReceive (section 3.1.4.15)** method with:
  - The same *phContext* parameter as in the call to **R_StartTransactionalReceive**.
  - *dwAck* := *readAck*.
  - The same *dwRequestId* as in the call to **R_StartTransactionalReceive**.
- If MQ_OK (0x00000000) is returned:
  - The client MUST return the reconstructed message to the message queuing application.
- Else if the return value is not MQ_OK:
  - The client MUST return the value to the message queuing application.

3.2.4.5 Reject a Message

The message queuing application MUST specify the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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 a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** (section 2.2.4.1) handle that has 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.
  - *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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** of the source queue and the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** of

the destination queue. The message queuing application MUST specify the *LookupId* of the message to be moved. The message queuing application MUST specify the transaction identifier if the destination queue is a **transactional queue**.

- If the destination queue is a transactional queue, the message queuing application MUST have enlisted the server in the transaction as specified in section 3.2.4.2, and it MUST specify the same transaction identifier for the Move a Message event.
- The client MUST call the **R_MoveMessage (section 3.1.4.10)** method and MUST specify the following parameter values:
  - *phContextFrom* set to the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** handle of the source queue, which was returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)**, and which MUST NOT have been previously closed through a call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.
  - *ulContextTo* set to the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** handle of the destination queue, which was returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue**, and which MUST NOT have been previously closed through a call to **R_CloseQueue**. This value MUST NOT be NULL.
  - *pTransactionId* set to the transaction identifier specified by the message queuing application if the destination queue is a transactional queue; otherwise, to a zero-value **XACTUOW** ([MS-MQMQ] section 2.2.18.1.8).
  - *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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** of the queue. The client MUST call the **R_PurgeQueue (section 3.1.4.6)** method with *phContext* set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** (section 2.2.4.1) handle that has 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.

3.2.4.8 Creating a Cursor

The message queuing application MUST specify the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** to associate with the created cursor. The client MUST call the **R_CreateCursor (section 3.1.4.4)** method with *phContext* set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** (section 2.2.4.1) handle that has 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. 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** RPC context handle 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 of the **R_StartReceive (section 3.1.4.7)** method 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 specified by the message queuing application.
- *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)** structure received in the *ppPacketSections* parameter, 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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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.

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 a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** handle that has 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.
  - *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).
- If MQ_OK (0x00000000) is returned:
  - The client MUST return the reconstructed message to the message queuing application.
- Else if the return value is not MQ_OK
  - The client MAY<35> return MQ_OK to the message queuing application.

3.2.4.10.2 Receive a Message by Using a Cursor with a Transaction

- The message queuing application 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 a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** handle, which was returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** and which MUST NOT have been previously closed through a call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.
  - *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 specified by the message queuing application.
  - *LookupId* set to 0x0000000000000000.
- 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**.
  - The same *dwRequestId* as in the call to **R_StartTransactionalReceive**.
- If MQ_OK (0x00000000) is returned:
  - The client MUST return the reconstructed message to the message queuing application.
- Else if the return value is not MQ_OK:

- The client MUST return the value to the message queuing application.

3.2.4.11 Cancel a Pending Peek or Receive

The message queuing application MUST specify the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** and the *dwRequestId* of the operation to be canceled.

- The client MUST call the **R_CancelReceive (section 3.1.4.8)** method and MUST specify the following parameter values:
  - *phContext* set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** (section 2.2.4.1) handle that has 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* set to the *dwRequestId* passed by the message queuing application.

3.2.4.12 Closing a Cursor

The message queuing application MUST specify the **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** 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.<36>
- The client MUST call the **R_CloseCursor (section 3.1.4.5)** method with the following:
  - The *phContext* parameter set to a **QUEUE_CONTEXT_HANDLE_NOSERIALIZE** (section 2.2.4.1) handle that has been returned by the server in the *pphQueue* output parameter of a prior call to **R_OpenQueue (section 3.1.4.2)** and that MUST NOT have been previously closed through a call to **R_CloseQueue (section 3.1.4.3)**. This value MUST NOT be NULL.
  - *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 **QUEUE_CONTEXT_HANDLE_NOSERIALIZE (section 2.2.4.1)** handle, that is to be closed, that has 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. If there are any pending requests associated with the **QUEUE_CONTEXT_HANDLE_SERIALIZE**, the client SHOULD cancel them as specified in section 3.2.4.11. If any open cursor handles are associated with the **QUEUE_CONTEXT_HANDLE_SERIALIZE**, 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 **QUEUE_CONTEXT_HANDLE_SERIALIZE**. The client MUST remove the **QUEUE_CONTEXT_HANDLE_SERIALIZE** from its state.<37>

3.2.5 Timer Events

None.

3.2.6 Other Local Events

None.

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 **QUEUE_CONTEXT_HANDLE_SERIALIZE (section 2.2.4.2)**.

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 **QUEUE_CONTEXT_HANDLE_SERIALIZE** with a call to **R_CloseQueue (section 3.1.4.3)**.

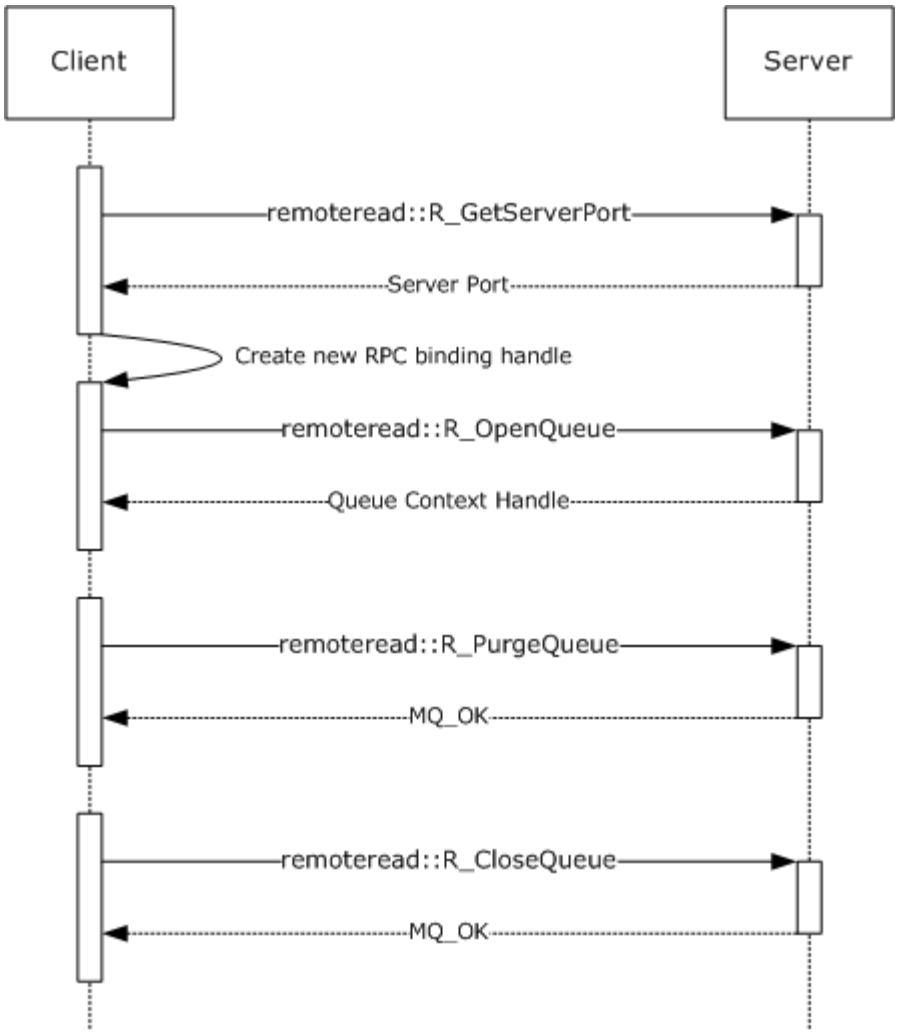


Figure 1: The client binds to a server and purges a queue

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.

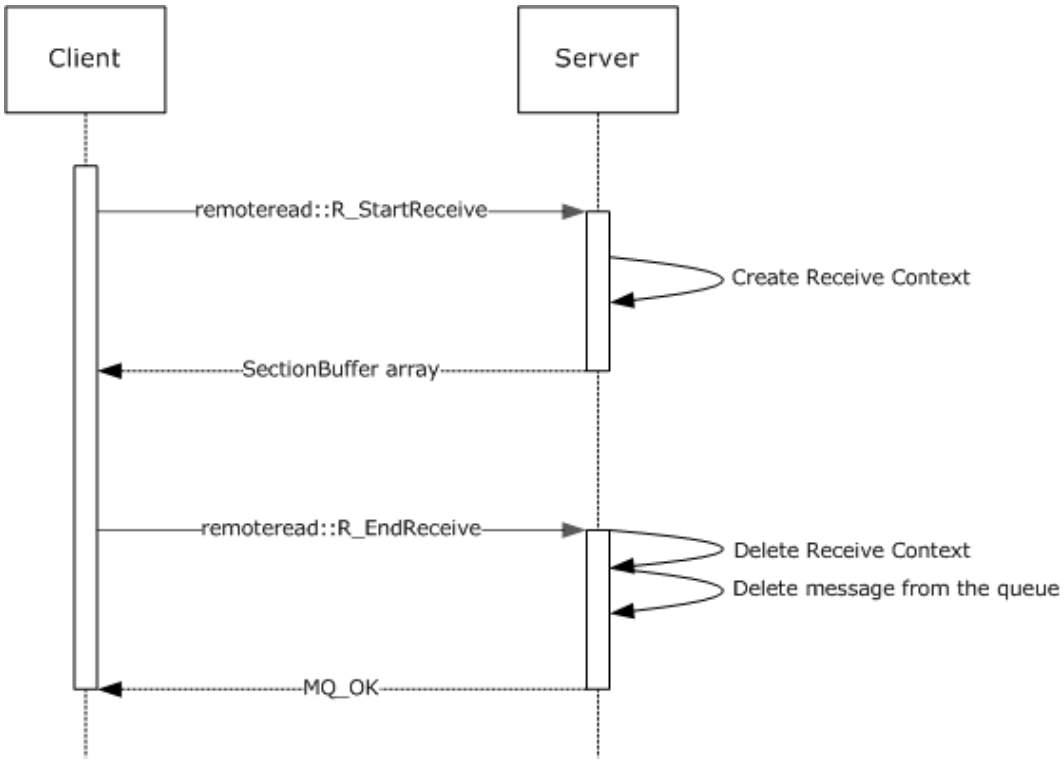


Figure 2: The client receives a message

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.

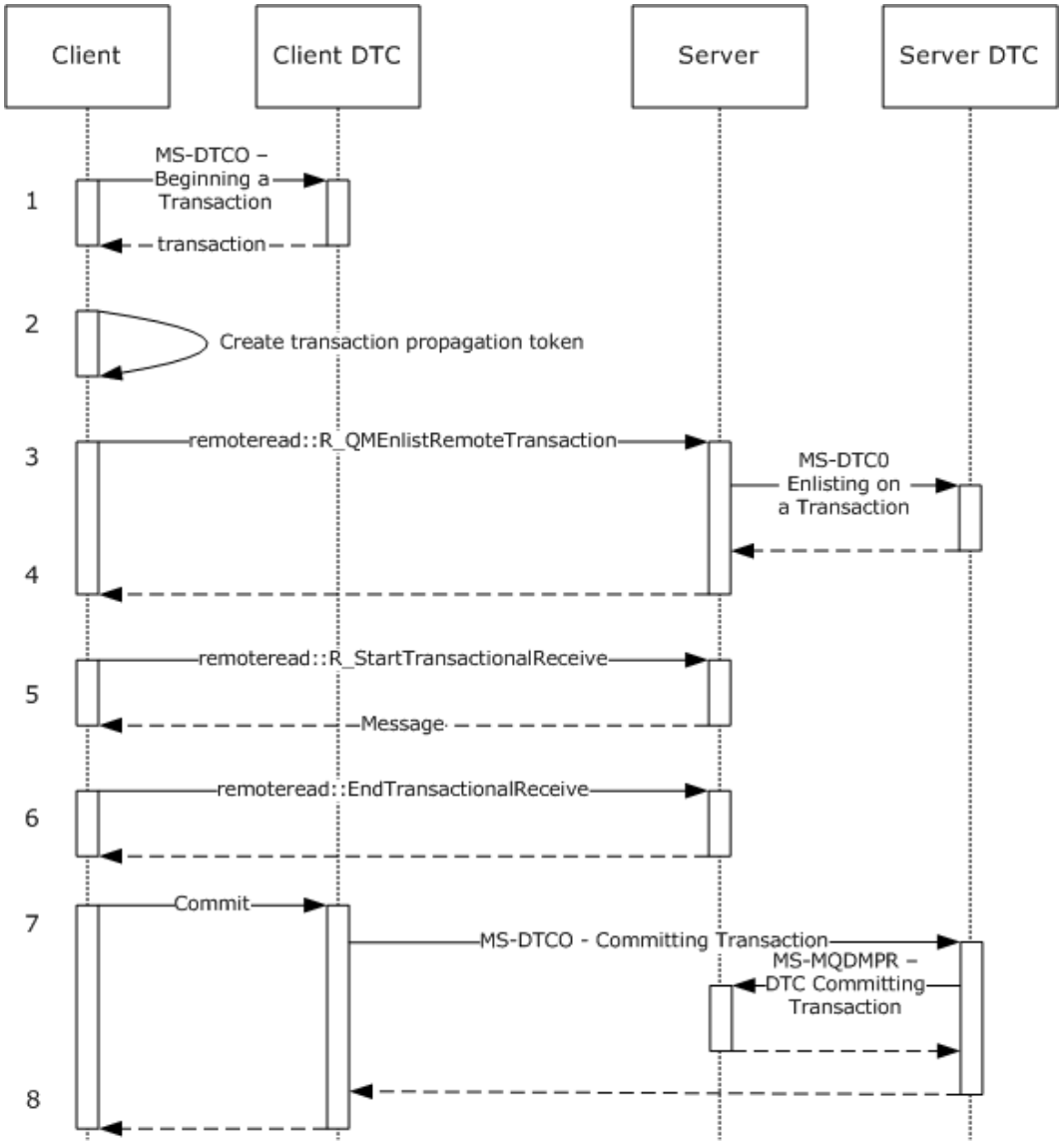


Figure 3: The client receives a message within a transaction

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.

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 3.3.4.12 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. The client commits the transaction, as specified in [MS-DTCO] section 3.3.4.8.2. The client DTC transaction manager notifies the server DTC transaction manager of the commit request.

8. The server deletes the message from the queue after it receives notification of the commit, by the DTC Transaction Commit event ([MS-MQDMPR] section 3.1.4.7), from the server DTC.

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<38> require a different minimum RPC authentication level from the client, depending on whether the client is a member of a Microsoft 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)**

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, and "ms-mqmq.idl" is the IDL as specified in [MS-MQMQ] Appendix A.

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


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

    typedef [context_handle]
    QUEUE_CONTEXT_HANDLE_NOSERIALIZE QUEUE_CONTEXT_HANDLE_SERIALIZE;

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

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

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

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

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

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

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

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

```
    [in] DWORD dwRequestId
    );

HRESULT R_MoveMessage(
    [in]  handle_t hBind,
    [in]  QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContextFrom,
    [in]  ULONGLONG ullContextTo,
    [in]  ULONGLONG LookupId,
    [in]  XACTUOW *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] QUEUE_CONTEXT_HANDLE_SERIALIZE* pphContext
    );

HRESULT R_QMEnlistRemoteTransaction(
    [in] handle_t hBind,
    [in] XACTUOW* 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] QUEUE_CONTEXT_HANDLE_NOSERIALIZE phContext,
    [in] ULONGLONG LookupId,
    [in] DWORD hCursor,
    [in] DWORD ulAction,
    [in] DWORD ulTimeout,
    [in] DWORD dwRequestId,
    [in] DWORD dwMaxBodySize,
    [in] DWORD dwMaxCompoundMessageSize,
    [in] XACTUOW* pTransactionId,
    [out] DWORD* pdwArriveTime,
    [out] ULONGLONG* pSequenceId,
    [out] DWORD* pdwNumberOfSections,
    [out, size_is(, *pdwNumberOfSections)]
           SectionBuffer** ppPacketSections
    );

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

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

7 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft Windows NT® operating system
- Microsoft Windows® 2000 operating system
- Windows Server® 2003 operating system
- Windows Vista® operating system
- Windows Server® 2008 operating system
- Windows® 7 operating system
- Windows Server® 2008 R2 operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 1.4: If the originating MSMQ application receives messages from a remote queue through a supporting server, the Queue Manager on the supporting server uses MSMQ: Queue Manager to Queue Manager Protocol (as specified in [MS-MQQP]) but does not support the MSMQ: Queue Manager Remote Read Protocol.

<2> Section 1.7: Windows NT, Windows 2000, and Windows XP do not support this protocol.

<3> Section 1.7: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 use Kerberos when the computer is a member of a Windows domain; otherwise, they use NTLM.

<4> 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 clients.

<5> Section 2.2.5.2: The ExtensionHeader is not supported on Windows Server 2003.

<6> Section 2.2.5.3: The SubqueueHeader is not supported on Windows Server 2003.

<7> Section 2.2.5.4: The DeadLetterHeader is not supported on Windows Server 2003.

<8> Section 2.2.5.5: The ExtendedAddressHeader is not supported on Windows Server 2003.

<9> Section 2.2.8: All Windows clients produce new **XACTUOW** values by calling the Windows RPC function UuidCreate.

<10> Section 3.1.2.1: If the registry key HKLM\SOFTWARE\Microsoft\MSMQ\Parameters\RpcCancelTimeout is defined and is set to a nonzero **DWORD** value, the protocol servers on Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 interpret this value as the RPC Call Timeout value in minutes.

<11> Section 3.1.2.2: The Windows default timeout is 5 * 60 * 1000 milliseconds (5 minutes). This default value can be overridden by setting the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ\Parameters\RpcCancelTimeout to the desired value, in minutes. This value MUST not be set to 0.

<12> Section 3.1.4: Opnums reserved for local use apply to Windows as follows.

Opnum	Description
1	Not used by Windows

<13> 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 clients.

<14> 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 (0xc00e0067).

Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers do not enforce limits on the number of unique concurrent clients. The *pClientId* parameter is ignored.

<15> Section 3.1.4.2: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 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).

<16> Section 3.1.4.2: The Windows Server 2003 protocol client sets the message queuing Major Version to 5. The Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol clients set the message queuing Major Version to 6.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing Major Version parameter.

<17> Section 3.1.4.2: The 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. The Windows 7 and Windows Server 2008 R2 protocol clients set the message queuing Minor Version to 1.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing Minor Version parameter.

<18> Section 3.1.4.2: The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol clients set the message queuing *BuildNumber* to a build-specific number.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing *BuildNumber* parameter.

<19> 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.
  - 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 protocol servers on Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 have their minimum RPC authentication level 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.

- 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.

<20> Section 3.1.4.10: **R_MoveMessage** is not implemented on Windows Server 2003.

<21> Section 3.1.4.11: **R_OpenQueueForMove (section 3.1.4.11)** is not implemented on Windows Server 2003.

<22> Section 3.1.4.11: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol clients set the *fNonRoutingServer* parameter 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).

<23> Section 3.1.4.11: The Windows Server 2003 protocol client sets the message queuing Major Version to 5. The Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol clients set the message queuing Major version to 6.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing Major Version parameter.

<24> Section 3.1.4.11: The 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.

The Windows 7 and Windows Server 2008 R2 protocol clients set the message queuing Minor Version to 1.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing Minor Version parameter.

<25> Section 3.1.4.11: The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol clients set the message queuing *BuildNumber* to a build-specific number.

The Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 protocol servers ignore the message queuing *BuildNumber* parameter.

<26> 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.

    - 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.
- 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.

<27> Section 3.1.4.12: **R_QMEnlistRemoteTransaction (section 3.1.4.12)** is not implemented on Windows Server 2003.

<28> Section 3.1.4.12: A server running Windows Vista, Windows Server 2008, Windows 7, or Windows Server 2008 R2 ignores the *pQueueFormat* parameter.

<29> Section 3.1.4.13: The **R_StartTransactionalReceive** method is not implemented on Windows Server 2003.

<30> Section 3.1.4.14: **R_SetUserAcknowledgementClass (section 3.1.4.14)** is not implemented on Windows Server 2003.

<31> Section 3.1.4.15: The **R_EndTransactionalReceive (section 3.1.4.15)** method is not implemented on Windows Server 2003.

<32> 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.

<33> 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 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.

<34> Section 3.2.4.4.1: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 clients return the message to the message queuing application with an MQ_OK (0x00000000) status even if the call to **R_EndReceive (section 3.1.4.9)** fails.

<35> Section 3.2.4.10.1: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 clients return the message to the message queuing application with an MQ_OK (0x00000000) status even if the call to **R_EndReceive (section 3.1.4.9)** fails.

<36> Section 3.2.4.12: Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 clients do not cancel pending requests associated with open cursor handles.

<37> Section 3.2.4.13: Windows Server 2003, Windows Vista, Windows Server 2008, and Windows 7 clients do not cancel pending requests or close associated cursor handles.

<38> 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.
  - 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, Windows Server 2008, Windows 7, and Windows Server 2008 R2 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, and the Anonymous Logon account is granted Peek or Receive permissions on the queue being accessed.
- 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.

8 Change Tracking

This section identifies changes that were made to the [MS-MQRR] protocol document between the May 2011 and June 2011 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

Major and minor changes can be described further using the following change types:

- New content added.
- Content updated.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.

- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.
- Content removed for template compliance.
- Obsolete document removed.

Editorial changes are always classified with the change type **Editorially updated.**

Some important terms used in the change type descriptions are defined as follows:

- **Protocol syntax** refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.
- **Protocol revision** refers to changes made to a protocol that affect the bits that are sent over the wire.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
1.2 References	Added explanatory statement regarding the removal of the publishing year from Microsoft Open Specification document references.	N	Content updated.
3.1.4.6 R_PurgeQueue (Opnum 6)	Substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard suffixes.	N	Content updated.
3.1.4.7 R_StartReceive (Opnum 7)	Substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard suffixes.	N	Content updated.
3.1.4.8 R_CancelReceive (Opnum 8)	Revised the processing logic to handle the absence of a PendingRequestEntry ADM element instance, substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard suffixes.	N	Content updated.
3.1.4.9 R_EndReceive (Opnum 9)	Substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard suffixes.	N	Content updated.
3.1.4.13 R_StartTransactionalReceive (Opnum 13)	Substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard	N	Content updated.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
	suffixes.		
3.1.4.15 R_EndTransactionalReceive (Opnum 15)	Substituted temporary variables for ADM element type names, and qualified all ADM element instances and temporary variables with standard suffixes.	N	Content updated.
3.2.4.2 Enlisting in a Transaction	64768 Added the pQueueFormat parameter to the required input parameter list for the R_QMEnlistRemoteTransaction method invocation.	N	Content updated.

9 Index

S

T

U

V