[MS-MQQB]:
Message Queuing (MSMQ):
Message Queuing Binary Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
05/11/2007	0.1		MCPP Milestone 4 Initial Availability
08/10/2007	1.0	Major	Updated and revised the technical content.
09/28/2007	2.0	Major	Updated and revised the technical content.
10/23/2007	2.0.1	Editorial	Revised and edited the technical content.

Date	Revision History	Revision Class	Comments
11/30/2007	2.0.2	Editorial	Revised and edited the technical content.
01/25/2008	2.0.3	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies the Message Queuing (MSMQ): Message Queuing Binary Protocol, which defines a mechanism for reliably transferring messages between two message **queues** located on two different hosts. The protocol uses TCP/SPX to transport the data, but augments it with additional levels of acknowledgment that ensure that the messages are reliably transferred regardless of TCP/SPX connection failures, application failures, or node failures.

Familiarity with public key infrastructure (PKI) concepts such as asymmetric and symmetric cryptography, asymmetric and symmetric encryption techniques, digital certificate concepts, and cryptographic key establishment is required for a complete understanding of this specification. In addition, a comprehensive understanding of the [X509] standard is required for a complete understanding of the protocol and its usage.

1.1 Glossary

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

Active Directory (AD)
Coordinated Universal Time (UTC)
Globally Unique Identifier (GUID)
Lightweight Directory Access Protocol (LDAP)
Little-Endian
NetBIOS
Network Byte Order
UCHAR
ULONG
Unicode
USHORT
X.509

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

Dead-Letter Queue
Independent Client
Message
Message Body
Message Queuing
MSMQ
MSMQ Routing Server
Private Queue
Public Queue
Queue
Queue Manager

The following terms are specific to this document:

Sequence: The set of message packets sent over a session that represent a message **sequence**. A message is associated with a **sequence** number that corresponds to its position within the **sequence**. **Sequence** numbers begin with 1 and increment by 1 with each subsequent message.

Source Journaling: The process of storing copies of outgoing messages on the source computer. Source journaling is configured on a per-message basis and can be used to track messages that were sent successfully, messages that could not be delivered, or both.

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

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.

[IANAPORT] Internet Assigned Numbers Authority, "Port Numbers", November 2006, http://www.iana.org/assignments/port-numbers

[MS-ADTS] Microsoft Corporation, "Active Directory Technical Specification", June 2007.

[MS-ADTS] Microsoft Corporation, "Active Directory Technical Specification", June 2007.

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

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

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

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

[MS-MQDS] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Protocol Specification", July 2007.

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

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

[RFC1319] Kaliski, B., "The MD2 Message-Digest Algorithm", RFC 1319, April 1992, http://www.ietf.org/rfc/rfc1319.txt

[RFC1320] Rivest, R. "The MD4 Message-Digest Algorithm", RFC 1320, April 1992, http://www.ietf.org/rfc/rfc1320.txt

[RFC1321] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, April 1992, http://www.ietf.org/rfc/rfc1321.txt

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

[RFC2212] Shenker, S., Partridge, C., and Guerin, R., "Specification of Guaranteed Quality of Service", RFC 2212, September 1997, http://www.ietf.org/rfc/rfc2212.txt

[RFC2268] Rivest, R., "A Description of the RC2(r) Encryption Algorithm", RFC 2268, March 1998, http://www.ietf.org/rfc/rfc2268.txt

[RFC3110] Eastlake III, D., "RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)", RFC 3110, May 2001, http://www.ietf.org/rfc/rfc3110.txt

[RFC3280] Housley, R., Polk, W., Ford, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002, http://www.ietf.org/rfc/rfc3280.txt

[RFC3962] Raeburn, K., "Advanced Encryption Standard (AES) Encryption for Kerberos 5", RFC 3962, February 2005, http://www.ietf.org/rfc/rfc3962.txt

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[RFC4234] Crocker, D., Ed. and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, http://www.ietf.org/rfc/rfc4234.txt

[RFC4757] Jaganathan, K., Zhu, L., and Brezak, J., "The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows", RFC 4757, December 2006, http://www.ietf.org/rfc/rfc4757.txt

[UNICODE] The Unicode Consortium, "Unicode Home Page", 2006, http://www.unicode.org/

[X509] ITU-T, "Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks", Recommendation X.509, August 2005, http://www.itu.int/rec/T-REC-X.509/en

Note There is a charge to download the specification.

1.2.2 Informative References

[MSDN-MQEIC] Microsoft Corporation, "Message Queuing Error and Information Codes", http://msdn2.microsoft.com/en-gb/library/ms700106.aspx

1.3 Protocol Overview (Synopsis)

The Message Queuing (MSMQ): Message Queuing Binary Protocol is used by a client to reliably transfer a message to a server. The protocol is stateful wherein the client establishes a connection to a server and then sends a variety of packets to transfer messages. The protocol defines additional behaviors such as administration acknowledgments and message **source journaling**. The protocol uses UDP to determine server availability and TCP/SPX to transport the data but augments it with additional levels of acknowledgment that ensure that the messages are reliably transferred regardless of TCP/SPX connection failures, application failures, or node failures.

1.3.1 Message Queuing

Message Queuing is a communications service that provides asynchronous and reliable **message** passing between client applications running on different hosts. In Message Queuing, clients send application messages to a queue and/or consume application messages from a queue. The queue can provide persistence of the messages, enabling them to survive across application restarts, and allowing the sending and receiving client applications to operate asynchronously from each other.

Queues are typically hosted by a communications service called a **queue manager (QM)**.

Implementing the queue manager as a separate service allows client applications to exchange queued messages asynchronously and eliminates the need for the client applications to execute at the same time.

Message Queuing is designed for the possibility of sending messages asynchronously to computers that are temporarily unavailable. When sending a message, the queue manager indicates to the client application that the sending operation has succeeded as soon as the message is created with valid properties and is placed in an outgoing queue, where the message remains until it is delivered to its destination or the message expires. Note that the sending operation does not immediately deliver the message. It just stores the message in a queue to be delivered asynchronously by the QM.

Queue managers handle the delivery of messages by continually checking for messages in all the local outgoing queues and attempting to transmit the messages that are found to their destinations. The queue manager on the sending side does not return any information to the sending application if the message does not reach its destination queue or if the message is discarded before being retrieved by a receiving application. Applications can obtain information from acknowledgment messages sent back from the destination host and can also examine the **dead letter** and journal queues for information on messages sent.

The Message Queuing (MSMQ): Message Queuing Binary Protocol defines a mechanism for reliably transferring messages between queue managers that are located on two different hosts. The protocol does not define the queue manager or its interface to client applications.

1.3.2 User Messages

A typical message exchanged in a message queuing system 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.

1.3.2.1 User Message Types

Messages sent using the Message Queuing (MSMQ): Message Queuing Binary Protocol are either express or recoverable. The choice between the two delivery options is essentially a choice between better performance with minimal resource use (express messaging) and reliability and recovery after a failure (recoverable messaging).

1.3.2.1.1 Express Message

When express messaging is used to send messages, the messages are stored in RAM during transfer and after delivery to the destination queue until they are received. This provides fast performance, but the messages are not recoverable if any computer where the messages reside fails. Notably, this means that express messages can be lost when the queue manager service is stopped. Express messages are not guaranteed to be delivered only once or in order.

Express messages can, like recoverable messages, survive a network failure. For example, if the client sends express messages and the link between the queue manager and the target computer fails, the queue manager continues to store the messages in its memory and will retry the connection. However, if the client process fails before the link is restored, the undelivered express messages are lost. Likewise, express messages on a server will be lost in the event of a process failure.

1.3.2.1.2 Recoverable Message

When recoverable messaging is used to send messages, the messages are written to disk on both the sending and receiving computer. After delivery to the destination queue, recoverable messages are stored on disk until they are consumed by a user application. This makes delivery somewhat slower than express messaging, but it is ideal when persistence through service restart or failure is required. If a computer fails or is shut down while sending messages, the messages are stored on

disk. Then when the computer is restarted and the queue manager service restarts, the sending process is automatically resumed. Recoverable messages are not guaranteed to be delivered only once or in order.

1.3.2.1.3 Transactional Message

A transactional message is a recoverable message that has exactly once and in order (EOIO) delivery guarantees. When delivering transactional messages, the protocol utilizes an additional level of acknowledgment to guarantee that messages arrive only once and in the correct order.

Transactional messaging is intended to be used in situations where the queue manager has captured one or more messages under a transaction and subsequently uses the Message Queuing (MSMQ): Message Queuing Binary Protocol to transfer the messages to a queue manager on a different host. A queue manager uses the Message Queuing (MSMQ): Message Queuing Binary Protocol to transfer messages after a transaction has committed. This protocol does not participate in transaction processing on the client or server.

This protocol does not mandate the implementation details of the transactional capture of messages as long as the external behavior of a queue manager is consistent with that specified in this document.

1.3.2.2 Message Security

Messages sent using the Message Queuing (MSMQ): Binary Messaging Protocol can be digitally signed and/or encrypted using a variety of technologies. The MD2 (as specified in [RFC1319]), MD4 (as specified in [RFC1320]), MD5 (as specified in [RFC1321]), and SHA-1 (as specified in [RFC3110]) hashing algorithms are supported for generating digital signatures, and the RC2 (as specified in [RFC2268]), RC4 (as specified in [RFC4757]), and AES (as specified in [RFC3962]) algorithms are supported for encrypting the message.

Sender identity can be specified by including an **X.509** digital certificate in the message. A receiver can authenticate a message by validating the digital signature by using the sender public key.

1.3.3 Queues

A queue is a logical data structure containing an ordered list of zero or more messages. A queue manager maintains a set of queues that hold messages. The queue manager requires a set of predefined or system queues (defined below) that are referenced throughout this document. A queue manager configuration defines a set of user queues that are the typical targets for messages sent via the Message Queuing (MSMQ): Message Queuing Binary Protocol.

Messages transferred using this protocol are addressed to specific queues by name. This protocol identifies queues by using the formats specified in [MS-MQMQ] section 2.1. This protocol does not mandate the implementation details of queues as long as their external behaviors are consistent with those described in this document.

A queue can be transactional or non-transactional. A transactional queue accepts only transactional messages, while a non-transactional queue accepts only express and recoverable messages. A transaction queue requires persistent storage of messages and guaranteed consistency through process or node failure.

1.3.3.1 System Queues

Queues that must be supported by all queue managers are referred to as system queues. System queues include the following types of queues:

Dead-Letter Queues: Contain messages that were sent from a host with a request for negative source journaling and could not be delivered. Dead-letter queues can be implemented as transactional or non-transactional.

Outgoing Queues: Contain messages that must be sent to specific destination addresses. Messages remain in outgoing queues until they can be transferred to their respective destination queues.

Connector Queues: Temporary locations to store messages that are forwarded to foreign messaging systems. Typically, a connector service running on a server waits for messages to arrive in one or more connector queues and forwards them to the foreign messaging systems. A connector service is application-defined and is not specified by this protocol.

Journal Queues: Contain copies of the messages sent from hosts when positive source journaling is requested by message queuing applications.

1.3.4 Source Journaling

Source journaling is the process of storing copies of outgoing messages on a source computer. It is configured on a per-message basis and is implemented as a property set programmatically by a message queuing application. Source journaling can be used to track messages that were sent successfully, messages that could not be delivered, or both. Source journaling is disabled by default.<1>

There are two types of source journaling: positive source journaling and negative source journaling.

1.3.4.1 Positive Source Journaling

Positive source journaling tracks successfully sent messages by placing message copies in the local host journal queue.

1.3.4.2 Negative Source Journaling

Negative source journaling tracks unsuccessfully sent messages by placing message copies in the local host dead-letter queue. When a message queuing application requests negative source journaling, messages are processed differently, depending on the message type.

For non-transactional messages, a copy of the message is placed in the local host dead-letter queue. Failure indicates that the source queue manager on the host cannot transfer the message to the destination host queue manager.

For transactional messages, a copy of the message is placed in the transactional dead-letter queue of the local host only if Message Queuing does not confirm that the message was retrieved from its destination queue.

1.3.5 Acknowledgments

1.3.5.1 Internal Acknowledgments

Internal acknowledgments are system-generated protocol packets sent from a receiving queue manager to a sending queue manager to acknowledge receipt (or other processing) of a user message. Internal acknowledgments are used by the protocol to enforce guarantees such as exactly once and in order delivery. Retransmission of messages may occur when an internal acknowledgment is not received within a specific period of time.

The protocol utilizes the following types of internal acknowledgments:

Session Acknowledgment: This packet acknowledges receipt of express and recoverable user message packets.

Order Acknowledgment: This packet acknowledges in-order receipt of a transactional message. This acknowledgment is required to guarantee exactly once in-order delivery of transactional messages. The packet can represent a positive or negative (for example, quota exceeded) acknowledgment. This acknowledgment is sent from the final destination queue manager to the original sender.

Final Acknowledgment: This packet acknowledges when a delivered transactional message has been removed from the destination queue. Removal from the destination queue could be the result of a user-level application reading the message from the queue or of an administrative action such as deleting the message or queue. The packet can represent a positive or negative acknowledgment. This acknowledgment is sent from the final destination queue manager to the original sender.

1.3.5.2 Administration Acknowledgments

Administration acknowledgment messages are system-generated express messages that are sent to administration queues specified in a packet. These messages can indicate whether a message has reached its destination queue or whether the message has been retrieved. When a message is rejected, an acknowledgment message can indicate the reason for its loss. Administration acknowledgment messages are utilized by application logic to identify the status of sent messages.

1.3.6 Message Tracing

Message tracing is the process of generating report messages when a user message leaves or arrives at a queue manager. Message tracing is an option that can be specified by the original sender of a message. The sender must specify the queue to receive the system-generated report messages. Report messages are utilized by application logic to track the delivery of sent messages.

A report message contains the following information:

- Source queue manager
- Destination queue manager
- Target queue name
- Received or sent time
- Message identifier

1.3.7 Message Routing

Message Queuing always attempts to establish a direct connection, or session, with the destination queue manager using the underlying TCP/SPX network protocol. If a direct connection is not possible due to lack of IP connectivity, Message Queuing servers with routing enabled (routing servers) can temporarily store messages and subsequently forward them to the destination computer or to another routing server.

Routing servers utilize this protocol to forward messages, via direct connections, to queue managers that are part of the route to a destination queue manager. This protocol utilizes the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to process messages that require routing.

1.3.8 Typical Scenario

A typical scenario for Message Queuing is to achieve reliable, asynchronous messaging between a client computer and server application. The client application might be an order application used for entering orders from customers. This application could be installed on a laptop computer, moving with the salesperson from customer site to customer site. Connectivity might not be available from those customer sites to the head office. In these cases, the order application on the salesperson's laptop computer would use Message Queuing to queue a message containing the order information to a local queue on the laptop computer.

When the salesperson returned to his branch office he would establish connectivity with the head office and the queued message would then be transferred using the Message Queuing (MSMQ): Message Queuing Binary Protocol from the local message queue on the laptop computer to the message queue server computer at the head office. At that point, the order would be retrieved from the message queue on the server and processed by the server application.

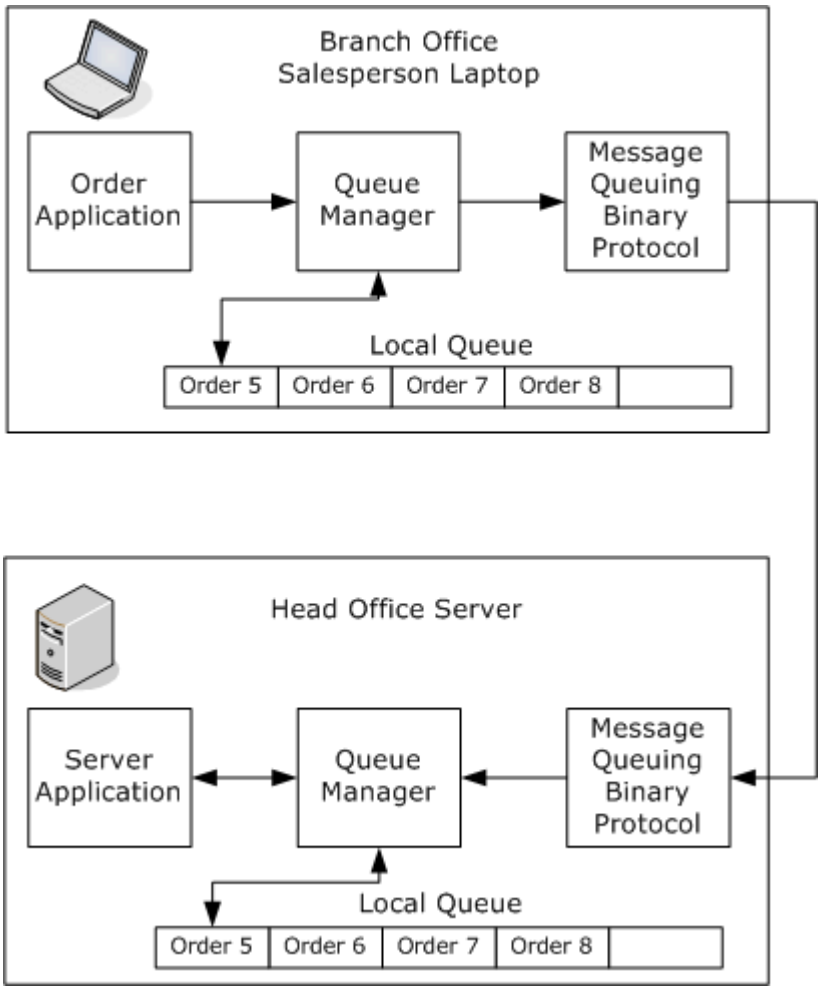


Figure 1: Queues and queue managers

The preceding diagram shows the relationship between the branch office laptop and the head office server. Messages containing orders are transferred from the outgoing queue on the laptop to the destination queue on the server.

1.4 Relationship to Other Protocols

The Message Queuing (MSMQ): Message Queuing Binary Protocol depends upon direct TCP/IP or IPX/SPX to provide a reliable stream-oriented transport for messages. The protocol uses UDP or IPX to determine server availability.

The protocol may rely upon LDAP, as specified in [MS-ADTS], or the Directory Service Protocol, as specified in [MS-MQDS], to look up persistently stored entries in **Active Directory (AD)**.

The protocol may rely upon the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to process messages sent using public and private **format names**.

1.5 Prerequisites/Preconditions

It is assumed that the protocol client has obtained the name of a server computer that supports this protocol and the name of a queue hosted on the server before this protocol is invoked. This specification does not mandate how a client acquires this information.

It is assumed that the protocol client has access to a private encryption key used to decrypt messages. A private key is typically stored in a secure location on the local host.

1.6 Applicability Statement

The server side of the Message Queuing (MSMQ): Message Queuing Binary Protocol is applicable for implementation by a queue manager providing message queuing communication services to clients. The client side of this protocol is applicable for implementation by client libraries providing message queue managers to applications or by a queue manager delegating requests on behalf of a client.

Applicable scenarios include cases in which there are disconnected users or unreliable connectivity, such as when a sales force works remotely, or in which guaranteed delivery is important, as required when sending orders from an entry system to a billing system.

The protocol is not applicable for distributed applications that require message delivery within a predefined amount of time. The protocol is not applicable for scenarios that require message data greater than 4 MB in size.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- **Supported Transports:** The Message Queuing (MSMQ): Message Queuing Binary Protocol can be implemented on top of TCP/UDP or SPX/IPX as specified in section 2.1. The protocol utilizes both TCP/UDP or SPX/IPX simultaneously.
- **Capability Negotiation:** There is a single version of this protocol at this time.

1.8 Vendor-Extensible Fields

The Message Queuing (MSMQ): Message Queuing Binary Protocol does not define any vendor-extensible fields.

1.9 Standards Assignments

This protocol uses the following assignments:

Parameter	Value	Reference
Microsoft-DS	TCP Port 1801 (0x709)	As specified in [IANAPORT]

2 Messages

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

2.1 Transport

A client exchanges messages with a server over a protocol session to perform actions such as session establishment, user message transfer, and message acknowledgment. A client uses a Ping Message to determine if a server is available.

2.1.1 Protocol Session

A protocol session is a TCP or an SPX connection used to send UserMessage Packets, which contain application-defined messages, and internal packets between a local and remote QM. The protocol session begins with an EstablishConnection Packet and ConnectionParameters Packet exchange to initialize the session. From that point, UserMessage Packets can be sent in either direction and are acknowledged by the SessionAck Packet, OrderAck Packet, and FinalAck Packet as appropriate.

Each packet MUST begin with a BaseHeader, which contains information such as packet type, signature, and packet size. The header is followed by one or more headers, depending on the packet type. The packets supported within a protocol session are ConnectionParameters Packet, EstablishConnection Packet, UserMessage Packet, SessionAck Packet, OrderAck Packet, and FinalAck Packet.

The protocol MUST use direct TCP or SPX for a protocol session.<2> The protocol initiator MUST establish a connection to TCP port 1801 or SPX port 876 on the acceptor. The TCP/SPX source port used by the initiator MAY be any TCP/SPX port value.<3> The protocol acceptor MUST listen for connections on TCP port 1801 or SPX port 876.<4>

2.1.2 Ping Request

The Ping Request is sent from an initiator to an acceptor to determine availability. An acceptor responds to a Ping request by sending a Ping Message back to the initiator. A Ping Message is used to determine if an acceptor is available and can accept a binary protocol **sequence** connection.

An initiator MUST broadcast a Ping Message using UDP or IPX port 3527. The source port used by the initiator MAY be any UDP/IPX port value. The acceptor MUST listen for Ping messages on UDP/IPX port 3527.

The acceptor MUST send a Ping response to the same UDP/IPX address and port that the Ping request was sent from, and the initiator MUST listen on that port for the response.

2.2 Packet Syntax

The Message Queuing (MSMQ): Message Queuing Binary Protocol uses **little-endian byte order**.

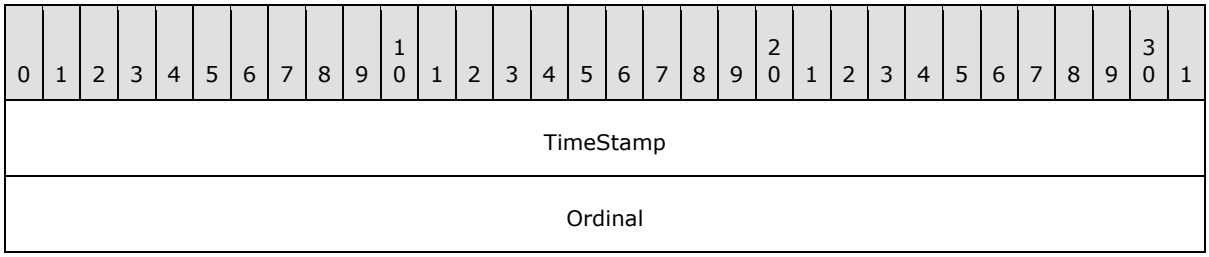
2.2.1 Common Data Types

2.2.1.1 GUID

This specification uses the **globally unique identifier** data type (the **GUID** data type), as specified in [MS-DTYP] section **2.3.4**).

2.2.1.2 TxSequenceID

A 64-bit value that identifies a sequence of transactional messages.

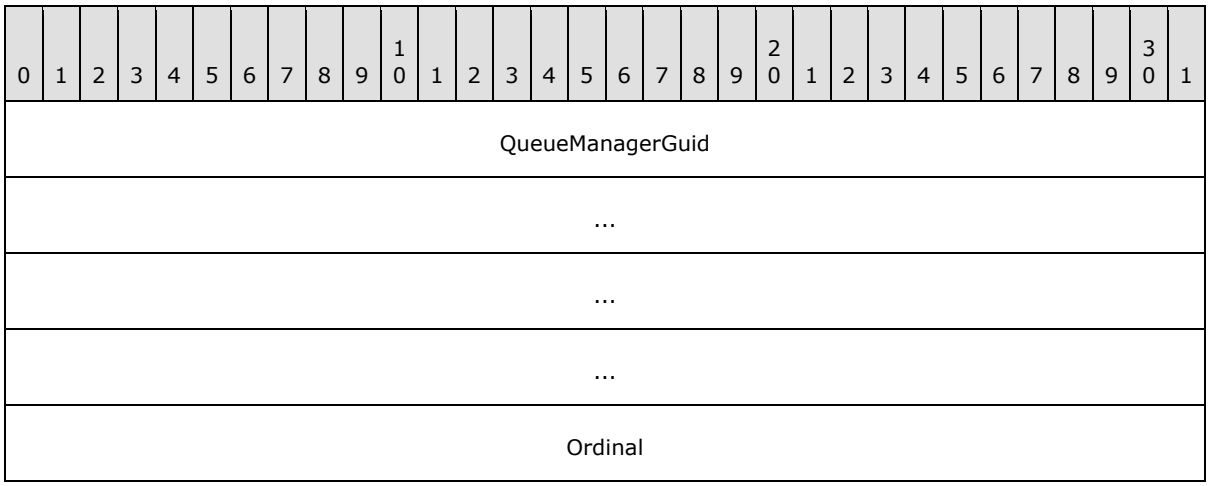


TimeStamp (4 bytes): A 32-bit unsigned long integer field that MUST be set to an implementation-dependent value that is guaranteed to be greater than any previously generated value.<5>

Ordinal (4 bytes): A 32-bit unsigned long integer field containing an increasing counter. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

2.2.1.3 MessageIdentifier

A 20-byte identifier that uniquely identifies a message from a queue manager.

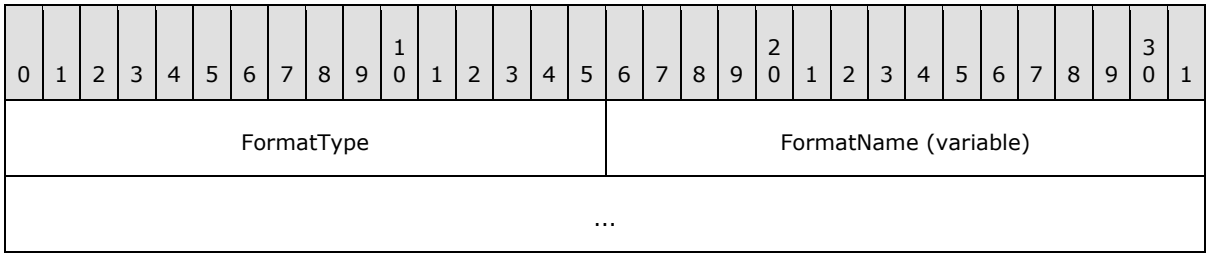


QueueManagerGuid (16 bytes): A **GUID**, as specified in [MS-DTYP], that identifies the sender queue manager.

Ordinal (4 bytes): A 32-bit unsigned long integer ordinal value that identifies the message. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

2.2.1.4 MQFFormatNameElement

The MQFFormatNameElement specifies a queue format name.



FormatType (2 bytes): A 16-bit unsigned short integer field specifying the queue format name type. The field MUST be set to one of the following values:

Value	Meaning
0x0001	Public queue
0x0002	Private queue
0x0003	Direct queue
0x0006	Distribution list

FormatName (variable): A variable-length byte array that contains a queue format name. This field MUST contain the format type specified by the FormatType field. The following table describes the data structure for each FormatType value:

Formattype	Queue type	Data type	Meaning
0x0001	Public	**GUID**	16 bytes. The value contains the **public queue** identifier. The start and end of this field is rounded up to the next 4-byte boundary.
0x0002	Private	**GUID** + **ULONG**	The **GUID**, as specified in [MS-DTYP], of the queue manager plus a **ULONG**, as specified in [MS-DTYP], that contains the **private queue** identifier. The start and end of this field is rounded up to the next 4-byte boundary.
0x0003	Direct	**WCHAR**[]	A null-terminated **WCHAR**, as specified in [MS-DTYP], buffer that contains a direct format name. The end of this field is rounded up to the next 4-byte boundary. The start of this field is rounded up to the next 2-byte boundary.
0x0006	Distribution List	**GUID**	16 bytes. The value contains the **distribution list** identifier. The start of this field is rounded up to the next 4-byte boundary.

Padding bytes in this field MAY be any value <6>.

2.2.1.5 Message Class Identifiers

A message class identifier is used to indicate the type of OrderAck Packet, FinalAck Packet, and type of message contained inside a UserMessage Packet. A message class identifier is stored in the **UserMessage.MessagePropertiesHeader.MessageClass** field.

A message can be a MQMSG_CLASS_NORMAL message that originated from a higher-layer application or a report message. There are message class identifier values used by negative administration acknowledgment messages that can indicate why a message did not arrive or was not retrieved.

The MESSAGE_CLASS_VALUES enumeration identifies the message types being used.

```
typedef  enum
{
  MQMSG_CLASS_NORMAL = 0x0000,
  MQMSG_CLASS_REPORT = 0x0001,
  MQMSG_CLASS_ACK_REACH_QUEUE = 0x0002,
  MQMSG_CLASS_ORDER_ACK = 0x00ff,
  MQMSG_CLASS_ACK_RECEIVE = 0x4000,
  MQMSG_CLASS_NACK_BAD_DST_Q = 0x8000,
  MQMSG_CLASS_NACK_DELETED = 0x8001,
  MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT = 0x8002,
  MQMSG_CLASS_NACK_Q_EXCEED_QUOTA = 0x8003,
  MQMSG_CLASS_NACK_ACCESS_DENIED = 0x8004,
  MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED = 0x8005,
  MQMSG_CLASS_NACK_BAD_SIGNATURE = 0x8006,
  MQMSG_CLASS_NACK_BAD_ENCRYPTION = 0x8007,
  MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q = 0x8009,
  MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG = 0x800A,
  MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER = 0x800B,
  MQMSG_CLASS_NACK_Q_DELETED = 0xC000,
  MQMSG_CLASS_NACK_Q_PURGED = 0xC001,
  MQMSG_CLASS_NACK_RECEIVE_TIMEOUT = 0xC002
} MESSAGE_CLASS_VALUES;
```

MQMSG_CLASS_NORMAL: Messages sent on behalf of higher-layer messaging applications are assigned this value.

A message that originated from a higher-layer application or a report message.

MQMSG_CLASS_REPORT: Indicates a report message used to track delivery of sent messages. For more information, see section 3.1.5.8.9.

MQMSG_CLASS_ACK_REACH_QUEUE: The class is used by administration acknowledgment messages.

Indicates that the original message was delivered to its destination queue.

MQMSG_CLASS_ORDER_ACK: The class is used to acknowledge in-order receipt of a transactional message. This acknowledgment is sent from the final destination queue manager to the original sender.

MQMSG_CLASS_ACK_RECEIVE: The class is used by administration acknowledgment messages.

Indicates that the original message was retrieved by a receiving application from the destination queue.

MQMSG_CLASS_NACK_BAD_DST_Q: Indicates that the destination queue is not available to the sender.

MQMSG_CLASS_NACK_DELETED: Indicates that the message was deleted by an administrative action before reaching the destination queue.

MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT: Indicates that the message did not reach the destination queue. This message can be generated by expiration of either the UserMessage.UserHeader.TimeToBeReceived time or UserMessage.BaseHeader.TimeToReachQueue time before the message reaches the destination queue.

MQMSG_CLASS_NACK_Q_EXCEED_QUOTA: Indicates that the message was not delivered because the destination queue is full. This message is generated only when the final destination queue is full.

When a message requiring routing arrives at an intermediate QM and the outgoing queue is full, the message is disregarded by the QM. This will result in the message being retransmitted at a later time. For more details, see section 3.1.5.8.2. This protocol utilizes the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to process messages that require routing.

MQMSG_CLASS_NACK_ACCESS_DENIED: Indicates that the access rights for placing messages in the destination queue are not allowed for the sender. The sender identity is specified by the SecurityHeader.

MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED: Indicates that the message was rejected because it exceeded the maximum routing hop count. This protocol utilizes the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to process messages that require routing.

MQMSG_CLASS_NACK_BAD_SIGNATURE: Indicates that Message Queuing could not authenticate the original message. The digital signature attached to the original message is not valid.

MQMSG_CLASS_NACK_BAD_ENCRYPTION: Indicates that the destination queue manager could not decrypt a private message.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q: Indicates that a transactional message was sent to a non-transactional queue.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG: Indicates that a **non-transactional message** was sent to a transactional queue.

MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER: Indicates that the requested encryption provider is not supported by the destination.

MQMSG_CLASS_NACK_Q_DELETED: Indicates that the queue was deleted before the message could be read from the queue.

MQMSG_CLASS_NACK_Q_PURGED: Indicates that the queue was purged and the message no longer exists.

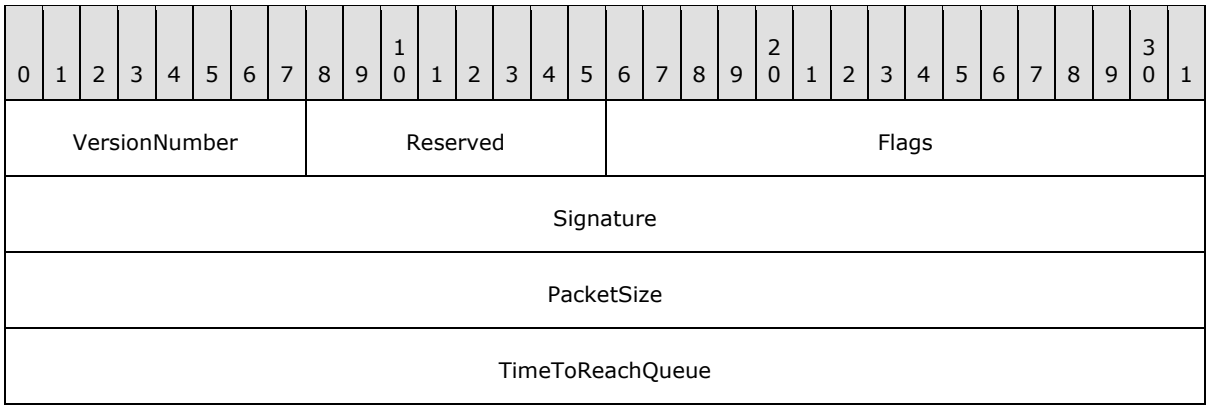
MQMSG_CLASS_NACK_RECEIVE_TIMEOUT: Indicates that the message was placed in the destination queue but was not retrieved from the queue before its time-to-be-received timer expired.

2.2.2 Common Headers

This section contains headers that are common to multiple packets.

2.2.2.1 BaseHeader

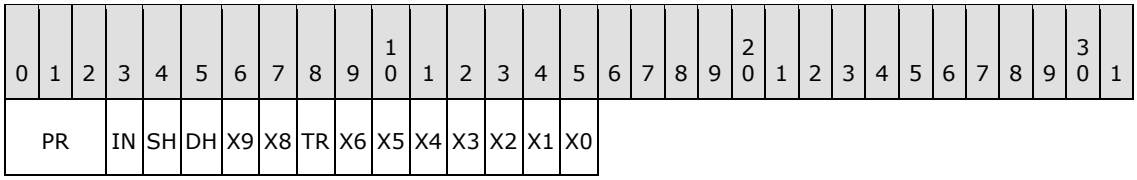
The BaseHeader is the first field of each packet described in this section. The BaseHeader contains information to identify and manage protocol packets.



VersionNumber (1 byte): An 8-bit unsigned integer that is the version of the packet format. This field MUST be set to the value 0x10.

Reserved (1 byte): An 8-bit unsigned integer field that is reserved for future use. The sender MUST set this field to 0x00, and the receiver MUST ignore it on receipt.

Flags (2 bytes): A 16-bit unsigned short integer containing a set of options that provide additional information about the packet. Any combination of these values is acceptable unless otherwise noted in the table below.



PR (3 bits): Specifies the priority of the message in the packet. This field has a valid range from 0x0 to 0x7, with 0x7 being the highest priority. The default is 0x3. A message with a higher priority MUST be placed closer to the front of the queue. This field MUST be set to a value of 0x0 if the packet contains a transactional message. For more details, see **UserHeader.Flags.TH** in section 2.2.2.3.

IN (1 bit): Type of message in the packet. MUST be set if the packet is an EstablishConnection Packet, a ConnectionParameters Packet, or a SessionAck Packet.

MUST not be set if the packet is a UserMessage Packet, OrderAck Packet, or FinalAck Packet.

SH (1 bit): Specifies if a SessionHeader is present in the packet. If set, the packet MUST contain a SessionHeader.

DH (1 bit): Specifies if a DebugHeader is present in the packet. If set, the packet MUST include a DebugHeader.

X9 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X8 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

TR (1 bit): Specifies if message tracing is enabled for this packet. MUST be set to enable messaging tracing.

X6 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X5 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X4 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X3 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X2 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X1 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X0 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

Signature (4 bytes): A 32-bit unsigned long integer that is the packet signature value. MUST be set to 0x524F494C (**big-endian** order).

PacketSize (4 bytes): A 32-bit unsigned long integer that indicates the packet size. MUST be set to the size, in bytes, of the entire packet including the base header. This field has a maximum value of 0x00400000.

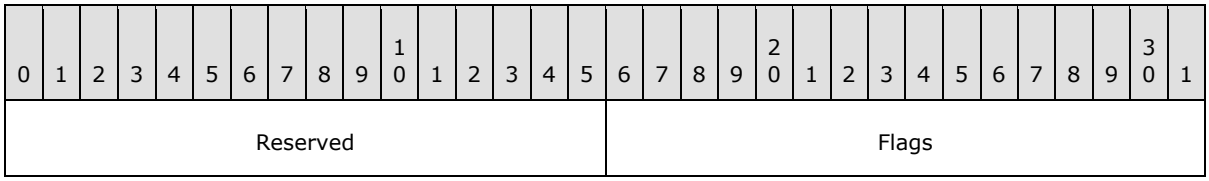
TimeToReachQueue (4 bytes): A 32-bit unsigned long integer that indicates the length of time, in seconds, that a UserMessage Packet has to reach its destination queue manager. This field has a valid range from 0x00000000 to 0xFFFFFFFF. The value 0xFFFFFFFF indicates an infinite time.

When a UserMessage Packet is sent or received, this value MUST be evaluated against the current system time and the **UserMessage.UserHeader.SentTime** field. If CURRENT_TIME - **UserMessage.UserHeader.SentTime** is greater than the value of this field, then the UserMessage Packet has expired and MUST be deleted by a sender and ignored by a receiver.

For the purpose of this section, CURRENT_TIME is defined as the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (**Coordinated Universal Time**).

2.2.2.2 InternalHeader

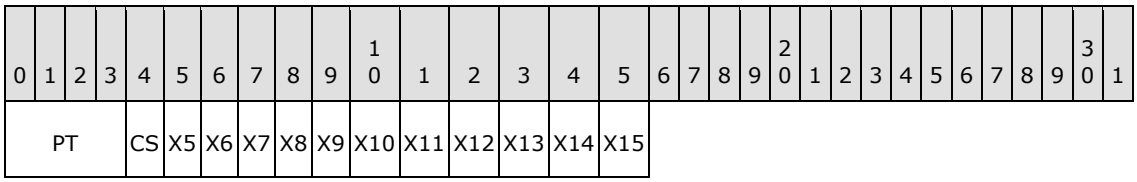
The InternalHeader contains packet type and state information for the session. This header is used by internal packets.



Reserved (2 bytes): A 16-bit unsigned short integer reserved for future use. MUST be set to 0x0000 by the sender, and MUST be ignored by the receiver.

Flags (2 bytes): A 16-bit unsigned short integer that contains a set of options that provide additional information about the packet. Any combination of these values is acceptable unless otherwise noted below.

Where the bits are defined as:



PT (4 bits): Specifies the packet type. These fields MUST be set to one of the following value combinations.

Value	Meaning
0x1	SessionAck Packet
0x2	EstablishConnection Packet
0x3	ConnectionParameters Packet

Any value not specified in the preceding table MUST be treated as an error by closing the session.

CS (1 bit): Specifies connection state. This field is used during the session negotiation to establish a connection. When set indicates that the connection is refused. This field MUST NOT be set in any packet other than an EstablishConnection Packet or a ConnectionParameters Packet.

X5 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X6 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X7 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X8 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X9 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X10 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X11 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X12 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

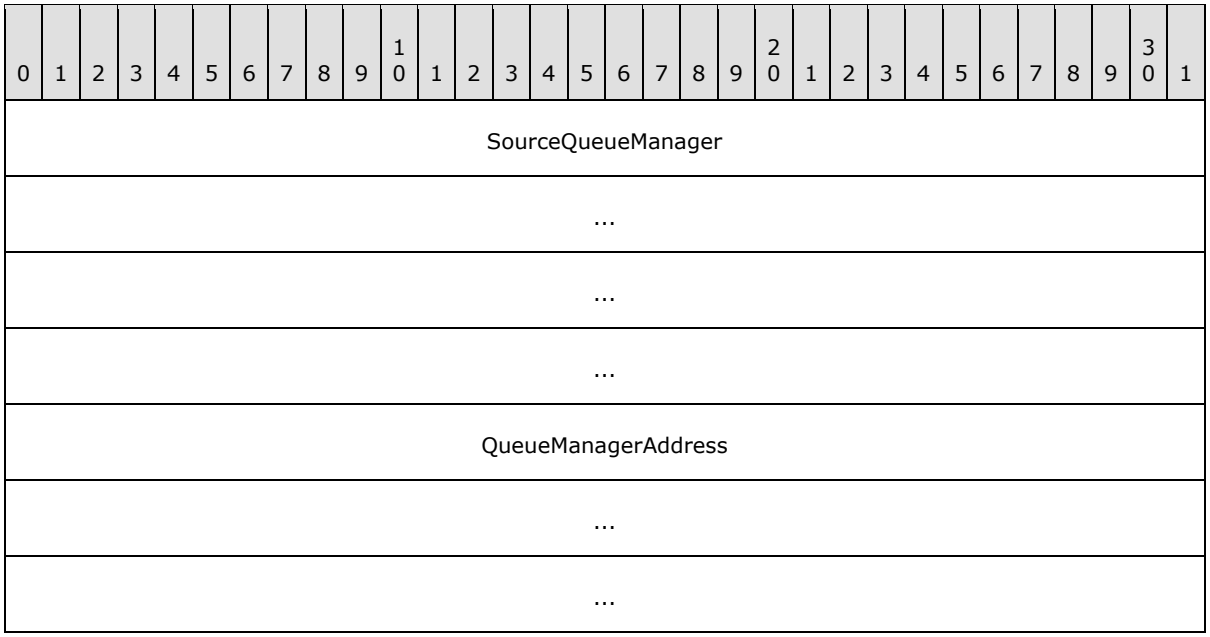
X13 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X14 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

X15 (1 bit): Reserved. SHOULD NOT be set when sent, and MUST be ignored when received.

2.2.2.3 UserHeader

The UserHeader contains source and destination information for the message in a UserMessage Packet.



...
TimeToBeReceived
SentTime
MessageID
Flags
DestinationQueue (variable)
...
AdminQueue (variable)
...
ResponseQueue (variable)
...
ConnectorType (optional)
...
...
...

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

QueueManagerAddress (16 bytes): A **GUID**, as specified in [MS-DTYP], that identifies the address of the destination queue manager.

If the message is sent to a public or private queue, this field MUST be set to the **GUID**, as specified in [MS-DTYP], of the destination queue manager. If the message is sent to a queue that uses a direct format name, then each byte of this field MUST be filled with the value 0x00.

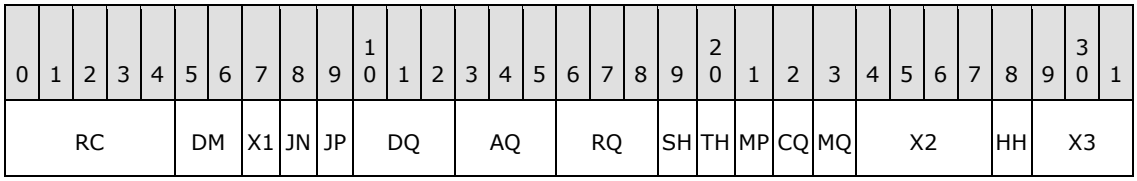
TimeToBeReceived (4 bytes): A 32-bit unsigned long integer that indicates the length of time, in seconds, that the message in the packet has before it expires. This field has a valid range from 0x00000000 to 0xFFFFFFFF. The value 0x00000000 indicates less than one second and 0xFFFFFFFF indicates an infinite time.

This time is measured from when the sending protocol receives the message. If the value is exceeded, the message MUST be removed from the destination queue. For more details on message expiration see section 3.1.5.8.5.

SentTime (4 bytes): A 32-bit unsigned long integer that MUST be set to the time when the packet was sent. This value represents the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time).

MessageID (4 bytes): A 32-bit unsigned long integer that is the message identifier specified by the protocol. The protocol MUST generate a unique identifier for each message it sends. For more details, see the MessageIdOrdinal value in 3.1.1.1.2.

Flags (4 bytes): A 32-bit unsigned long integer that contains a set of options that provide additional information about the packet. Any combination of these values is acceptable unless otherwise noted below.



RC (5 bits): The number of routing servers that have processed the UserMessage Packet. This value MUST be set to 0x00 when the packet is generated. This field has a valid range from 0x00 to 0x1D. A routing server MUST increment this value by 1 when it processes the packet. If the incremented value exceeds the valid range, then the packet MUST be deleted, as specified in section 3.1.5.8.2.

This protocol utilizes the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to process messages that require routing. For more details see section 3.1.5.8.8.

DM (2 bits): The delivery mode of the packet. The field MUST be set to one of the following values:

Value	Meaning
0x0	Express messaging. Express messages are not written to disk.
0x1	Recoverable messaging (including transactional). Recoverable messages are written to disk.

Note: a transactional message is a recoverable message that has UserHeader.Flags.TH set to 0x1, which indicates the presence of a TransactionHeader header.

X1 (1 bit): Reserved bit field. MUST NOT be set when sent, and MUST be ignored when received.

JN (1 bit): Specifies if negative source journaling is enabled.<7> If set, the protocol SHOULD log a record locally in the event of message delivery failure.<8>

JP (1 bit): Specifies if positive source journaling is enabled. If set, the protocol SHOULD log a record locally if the message is successfully delivered.<9>

DQ (3 bits): Type of destination queue. The field MUST be set to one of the following values:

Value	Meaning
0x3	Private queue on destination host.
0x5	Public queue.
0x6	Private queue.
0x7	Direct queue.

AQ (3 bits): Type of administration queue. The field MUST be set to one of the following values:

Value	Meaning
0x0	None.
0x2	Private queue on source host.
0x3	Private queue on destination host.
0x5	Public queue.
0x6	Private queue on host other than source or destination host.
0x7	Direct queue.

RQ (3 bits): Type of response queue. The field MUST be set to one of the following values:

Value	Meaning
0x0	None.
0x1	Same as administration queue.
0x2	Private queue on source host.
0x3	Private queue on destination host.
0x4	Private.
0x5	Public queue.
0x7	Direct queue.

SH (1 bit): Specifies if a SecurityHeader is present in the UserMessage packet. If set, the packet MUST contain a SecurityHeader; otherwise, it MUST NOT.

TH (1 bit): Specifies if a TransactionHeader is present in the UserMessage packet. If set, the packet MUST contain a TransactionHeader; otherwise, it MUST NOT. A transactional

message MUST contain a TransactionHeader. An express message MUST NOT contain a TransactionHeader.

MP (1 bit): Specifies if a MessagePropertiesHeader is present in the UserMessage packet. If set, the packet MUST contain a MessagePropertiesHeader; otherwise, it MUST NOT.

CQ (1 bit): Specifies if the ConnectorType field is present in the packet. If set, the packet MUST contain a ConnectorType field; otherwise, it MUST NOT.

MQ (1 bit): Specifies if a MultiQueueFormatHeader is present in the UserMessage packet. If set, the packet MUST contain a MultiQueueFormatHeader; otherwise, it MUST NOT.

X2 (4 bits): Reserved bit field. MUST NOT be set when sent, and MUST be ignored when received.

HH (1 bit): Specifies if a SoapHeader is present in the packet. If set, the UserMessage packet MUST contain a SoapHeader.

X3 (3 bits): Reserved bit field. MUST NOT be set when sent, and MUST be ignored when received.

DestinationQueue (variable): The destination queue specifies the final destination of the message that is contained inside the UserMessage packet. This field MUST be set to one of the queue types specified below. The format of the destination queue name varies depending on the queue type specified in the **Flags.DQ** field. The size of this field also depends on the queue type, as shown here:

Flags.DQ field	Queue type	Data type	Meaning
0x3	Private queue on destination host.	**ULONG**	A 4-byte private queue identifier.
0x5	Public	**GUID**	16 bytes. The value contains the public queue identifier.
0x6	Private	**ULONG**	4 bytes. The value contains the private queue identifier.
0x7	Direct	**USHORT** + **WCHAR[]**	The **USHORT** value specifies the length in bytes of the null-terminated **WCHAR** buffer that contains a direct format name that follows. The end of this field is rounded up to the next 4-byte boundary, so that the **AdminQueue** field that follows will begin on a 4-byte boundary. The **USHORT** length value includes the null terminator but does not include rounding bytes.

When the **Flags.DQ** field is set to 0x5 or 0x6 the **QueueManagerAddress** field MUST be set to the GUID of the destination queue manager.

Padding bytes in this field MAY be any value.<10>

Any value not specified in the preceding table MUST be treated as an error by closing the session. Details on direct format names are as specified in [MS-MQMQ] section 2.1.2.

AdminQueue (variable): The name of the administration queue. This field specifies the response queue where administration acknowledgment messages are sent. An administration response queue MUST be specified if a MessagePropertiesHeader is included and any bits are set in the **MessagePropertiesHeader.Flags** field; otherwise, this field MUST not be specified. Details on administration acknowledgments are as specified in sections 1.3.5.2 and 3.1.5.8.10.

The format of the administration queue varies depending on the queue type specified in the **Flags.AQ** field. The size of this field also depends on the queue type, as shown in the following table.

Flags.AQ field	Queue type	Data type	Meaning
0x2	Private queue on source host	**ULONG**	A 4-byte private queue identifier.
0x3	Private queue on destination host	**ULONG**	A 4-byte private queue identifier.
0x5	Public	**GUID**	A 16-byte value that contains the public queue identifier.
0x6	Private queue on host other than the source or destination host	**GUID** + **ULONG**	The **GUID**, as specified in [MS-DTYP], of the queue manager plus a ULONG that contains the private queue identifier.
0x7	Direct	**USHORT** + **WCHAR[]**	The **USHORT** value, as specified in [MS-DTYP], specifies the length in bytes of the null-terminated **WCHAR** buffer, as specified in [MS-DTYP], that contains a direct format name that follows. The end of this field is rounded up to the next 4-byte boundary, so that the **ResponseQueue** field that follows will begin on a 4-byte boundary. The **USHORT** length value includes the null terminator but does not include rounding bytes.

Padding bytes in this field MAY be any value.<11>

Any value not specified in the preceding table MUST be treated as an error by closing the session. Details on direct format names are as specified in [MS-MQMQ] section 2.1.2.

ResponseQueue (variable): A variable-length array of bytes containing the name of the response queue. The response queue is an application-defined value that specifies a queue that a receiving application could use to send a reply message.

The format of the response queue varies depending on the queue type specified in the **Flags.RQ** field. The size of this field also depends on the queue type, as shown here:

Flags.RQ field	Queue Type	Data Type	Meaning
0x1	Same as administration	None	0 bytes.

Flags.RQ field	Queue Type	Data Type	Meaning
	queue		
0x2	Private queue on source host	**ULONG**	A 4-byte private queue identifier.
0x3	Private queue on destination host	**ULONG**	A 4-byte private queue identifier.
0x4	Private queue on host other than the source or destination host	**GUID** + **ULONG**	A **GUID**, as specified in [MS-DTYP], of the queue manager plus a **ULONG**, as specified in [MS-DTYP], that contains the private queue identifier.
0x5	Public	**GUID**	A 16-byte value that contains the public queue identifier.
0x7	Direct	**USHORT** + **WCHAR[]**	The **USHORT**, as specified in [MS-DTYP], value specifies the length in bytes of the null-terminated **WCHAR**, as specified in [MS-DTYP], buffer that contains a direct format name that follows. The end of this field is rounded up to the next 4-byte boundary, so that the **ConnectorType** field that follows will begin on a 4-byte boundary. The **USHORT** length value includes the null terminator but does not include rounding bytes.

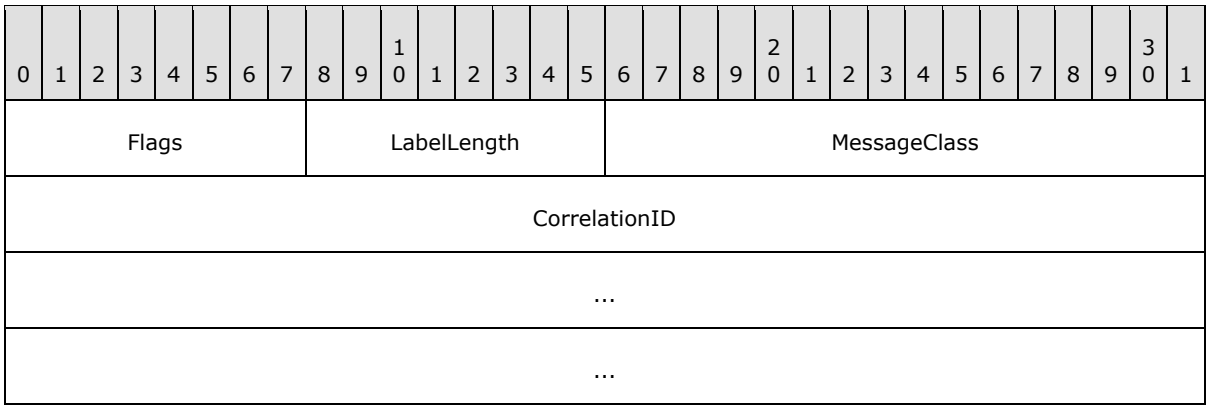
Padding bytes in this field MAY be any value.<12>

Any value not specified in the preceding table MUST be treated as an error by closing the session. Details on direct format names are as specified in [MS-MQMQ] section 2.1.2.

ConnectorType (16 bytes): An optional field that represents an application-defined **GUID** as specified in [MS-DTYP]. This field MUST be present when the **Flags.CQ** field is set. The field can be used by application-level logic to associate a message in a connector queue with a connector service.

2.2.2.4 MessagePropertiesHeader

The MessagePropertiesHeader contains property information about a UserMessage Packet.



...
...
BodyType
ApplicationTag
MessageSize
AllocationBodySize
PrivacyLevel
HashAlgorithm
EncryptionAlgorithm
ExtensionSize
MessageData (variable)
...

Flags (1 byte): An 8-bit unsigned integer field that specifies administration acknowledgments. Any combination of these values is acceptable unless otherwise noted in the table below.

For more details on administration acknowledgments, see Sending Administration Acknowledgments (section 3.1.5.8.10).

0	1	2	3	4	5	6	7
PA	PR	NA	NR	X	X	X	X

Where the bits are defined as:

Value	Description
PA	If the message is delivered to the destination queue, the server MUST send a positive acknowledgment.
PR	If the message is received from the destination queue, the server MUST send a positive acknowledgment.

Value	Description
NA	If the message is not delivered to the destination queue, the server MUST send a negative acknowledgment.
NR	If the message is not received from the destination queue, the server MUST send a negative acknowledgment.
X	Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.
X	Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.
X	Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.
X	Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

LabelLength (1 byte): An 8-bit unsigned integer field that MUST be set to the number of elements in the Label value contained inside the **MessageData** field. This field has a valid range from 0x00 to 0xFA. This value includes the end-of-string character.

MessageClass (2 bytes): A 16-bit unsigned short integer that specifies the class of the message in the packet. The value MUST be set to a value specified in section 2.2.1.5.

CorrelationID (20 bytes): This field is an application-defined buffer that specifies a correlation value for the message contained in the UserMessage Packet. A receiving application can use this field to sort or filter messages. Each byte of this field MUST be set to a value between 0x00 and 0xFF.

If this header appears inside an administration acknowledgment message, as specified in section 3.1.5.8.10, then this field MUST be set to a MessageIdentifier consisting of **UserMessage.UserHeader.MessageID** and **UserMessage.UserHeader.SourceQueueManager** of the message being acknowledged. See section 2.2.1.3 for details of the MessageIdentifier type.

BodyType (4 bytes): A 32-bit unsigned long integer that specifies the type of data that is contained in the message body. This value MUST be set to a **PROPVARIANT** type constant as specified in [MS-MQMQ] section **2.2.12**.

ApplicationTag (4 bytes): A 32-bit unsigned long integer that specifies an application-defined value that can be used to organize messages. The field MUST be set to a value between 0x00000000 to 0x00400000, inclusive.<13>

MessageSize (4 bytes): A 32-bit unsigned long integer that MUST be set to the size, in bytes, of the **MessageBody** contained inside the **MessageData** field. The field MUST be set to a value between 0x00000000 to 0x00400000, inclusive.<14>

AllocationBodySize (4 bytes): A 32-bit unsigned long integer field that MUST be set to the size, in bytes, of the data allocated for the **MessageData.MessageBody** field. This size can

be larger than the actual Message Body size; for example, an encrypted message body might be larger than the original unencrypted message body.

PrivacyLevel (4 bytes): A 32-bit unsigned long integer field that specifies the privacy level of the message in the UserMessage Packet. The privacy level determines what part of the message is encrypted. The field SHOULD be set to one of the following values:<15>

Value	Meaning
0x00000000	No encryption. The message body is sent as clear text.
0x00000001	The **MessageData.MessageBody** field is encrypted using 40-bit end-to-end encryption.
0x00000002	The **MessageData.MessageBody** field is encrypted using 40-bit end-to-end encryption.
0x00000003	The **MessageData.MessageBody** field is encrypted using 128-bit end-to-end encryption.
0x00000005	The **MessageData.MessageBody** field is encrypted using Advanced Encryption Standard (AES).

Any value not specified in the preceding table MUST be treated as an authentication failure.

HashAlgorithm (4 bytes): A 32-bit unsigned long integer that specifies the hashing algorithm that is used when authenticating the message. SHOULD be set to a value in the following table.<16>

Value	Meaning
0x00008001	Specifies the MD2 hash algorithm. Details about the algorithm are as specified in [RFC1319].
0x00008002	Specifies the MD4 hash algorithm. Details about the algorithm are as specified in [RFC1320].
0x00008003	Specifies the MD5 hash algorithm. Details about the algorithm are as specified in [RFC1321].
0x00008004	Specifies the SHA1 hash algorithm. For more details, the algorithm specification can be found at [RFC3110].

Any value not specified in the preceding table MUST be treated as an authentication failure.

EncryptionAlgorithm (4 bytes): A 32-bit unsigned long integer that specifies the encryption algorithm used to encrypt the Message Body contained inside the **MessageData** field. This field MUST be set to a value in the following table.

Value	Meaning
0x00006602	Specifies the RC2 algorithm. Details about the algorithm are as specified in [RFC2268].
0x00006801	Specifies the RC4 algorithm. Details about the algorithm are as specified in [RFC4757].

Value	Meaning
0x00006611	Specifies the AES algorithm. Details about the algorithm are as specified in [RFC3962].

Any value not specified in the preceding table MUST be treated as a failed decryption error.

ExtensionSize (4 bytes): A 32-bit unsigned long integer field that MUST be set to the length, in bytes, of the application-defined **ExtensionData** contained inside the **MessageData** field. The field MUST be set to a value between 0x00000000 to 0xFFFFFFFF.

MessageData (variable): This field is a buffer containing message data. This field contains the **Label**, **ExtensionData**, and the **MessageBody**. The field MUST be in the following format:

Offset	Field
0	Label
LabelLength * 2	ExtensionData
(LabelLength * 2) + ExtensionSize	MessageBody

The offset values listed above are offsets in bytes. LabelLength and ExtensionSize refer to the corresponding fields in the MessagePropertiesHeader packet.

The **Label** field is an application-defined **Unicode** string. This field can be used by an application to assign a short descriptive string to the message. This field MUST be in the format specified by the following ABNF rule:

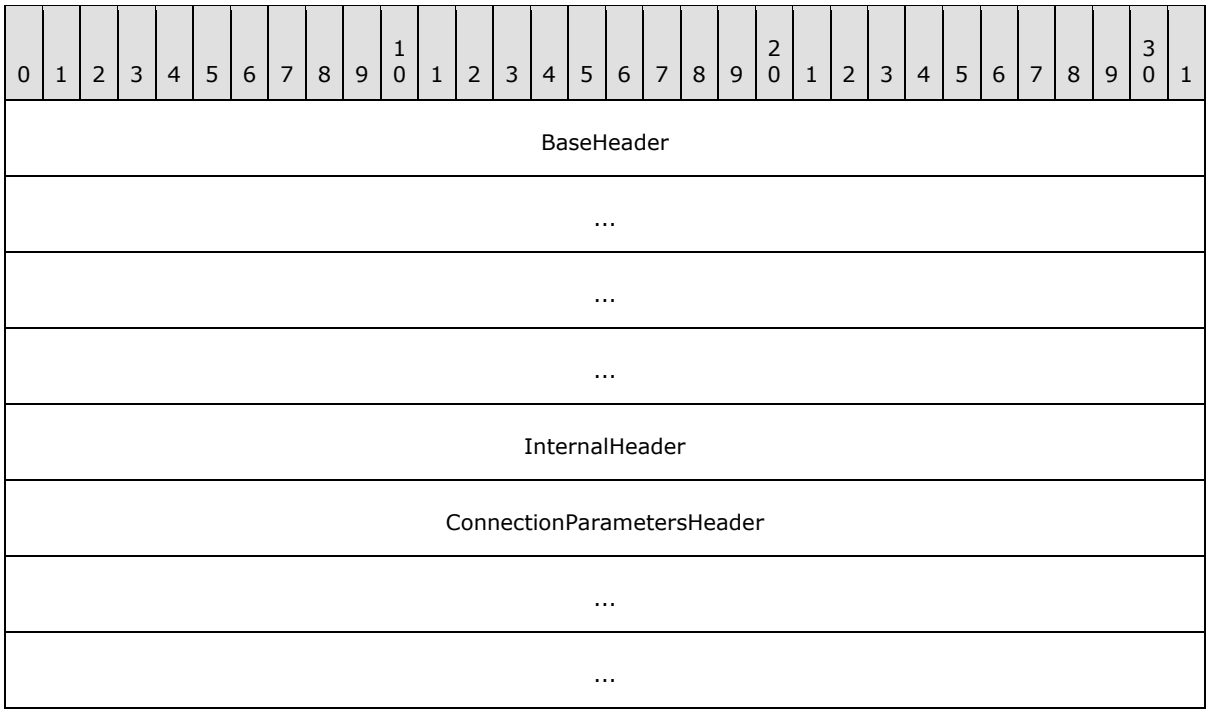
label = 0*249(%x0001-FFFF) 0x0000

The **ExtensionData** field provides a place to put additional application-defined information that is associated with the message.

The **MessageBody** field is the application-defined message payload.

2.2.3 ConnectionParameters Packet

The ConnectionParameters Packet is used to communicate connection parameters from the initiator to the acceptor during session initialization.



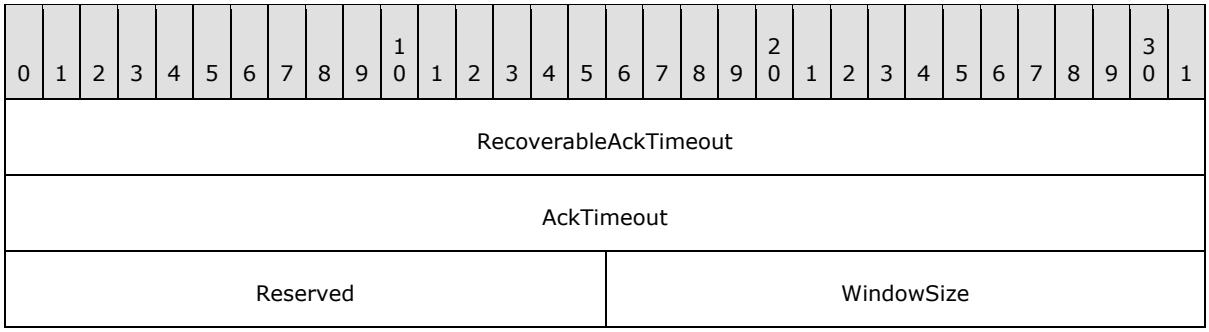
BaseHeader (16 bytes): A BaseHeader. The **BaseHeader.Flags.IN** field MUST be set to indicate that this packet is an internal message.

InternalHeader (4 bytes): An InternalHeader. The **InternalHeader.Flags.PT** field MUST be set to 0x3.

ConnectionParametersHeader (12 bytes): A ConnectionParametersHeader that contains parameters for the acknowledgment timeout and sliding window size.

2.2.3.1 ConnectionParametersHeader

The ConnectionParametersHeader contains parameters for acknowledgment timeout and sliding window size.



RecoverableAckTimeout (4 bytes): A 32-bit unsigned long integer that specifies the minimum time, in milliseconds, that the protocol waits before sending a SessionAck Packet acknowledgment after receiving a recoverable UserMessage Packet. This field has a valid

range from 0x000001F4 to 0x0001D4C0, inclusive. This field MUST be set based on system configuration, which is implementation-dependent.<17>

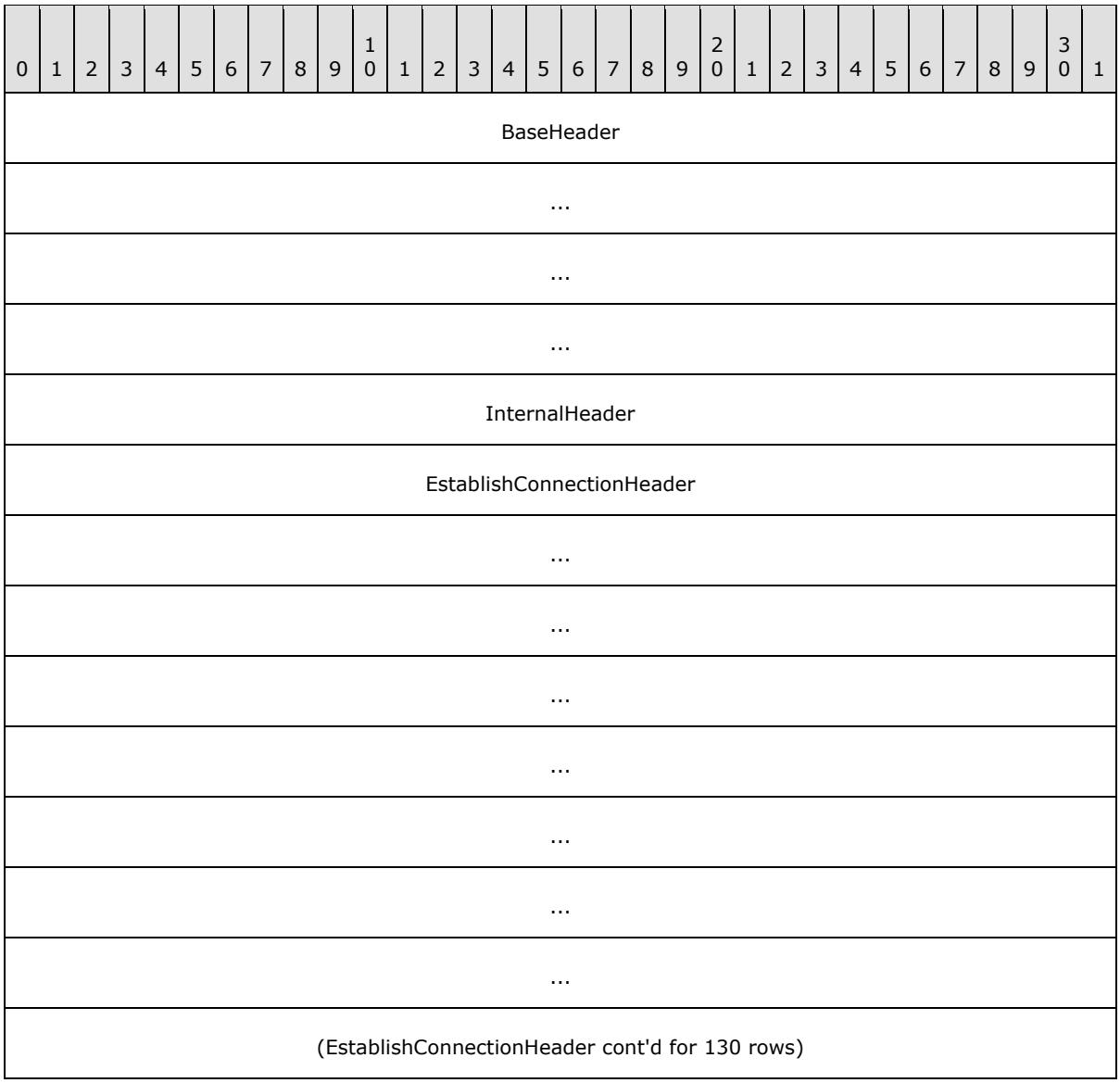
AckTimeout (4 bytes): A 32-bit unsigned long integer that specifies the time, in milliseconds, that the protocol waits for a SessionAck Packet before closing the session. This field has a valid range from 0x00004E20 to 0x0001D4C0, inclusive. This field MUST be set based on system configuration, which is implementation-dependent.<18>

Reserved (2 bytes): A 16-bit unsigned integer reserved for future use. MUST be set to 0x0000 by the sender, and MUST be ignored by the receiver.

WindowSize (2 bytes): A 16-bit unsigned integer containing a sliding window size, based on the number of unacknowledged packets received, for sending session acknowledgments. This field has a valid range from 0x0000 to 0xFFFF. This field SHOULD be set to value 0x0020.<19>

2.2.4 EstablishConnection Packet

The initiator sends an EstablishConnection Packet to an acceptor to initiate a protocol session.



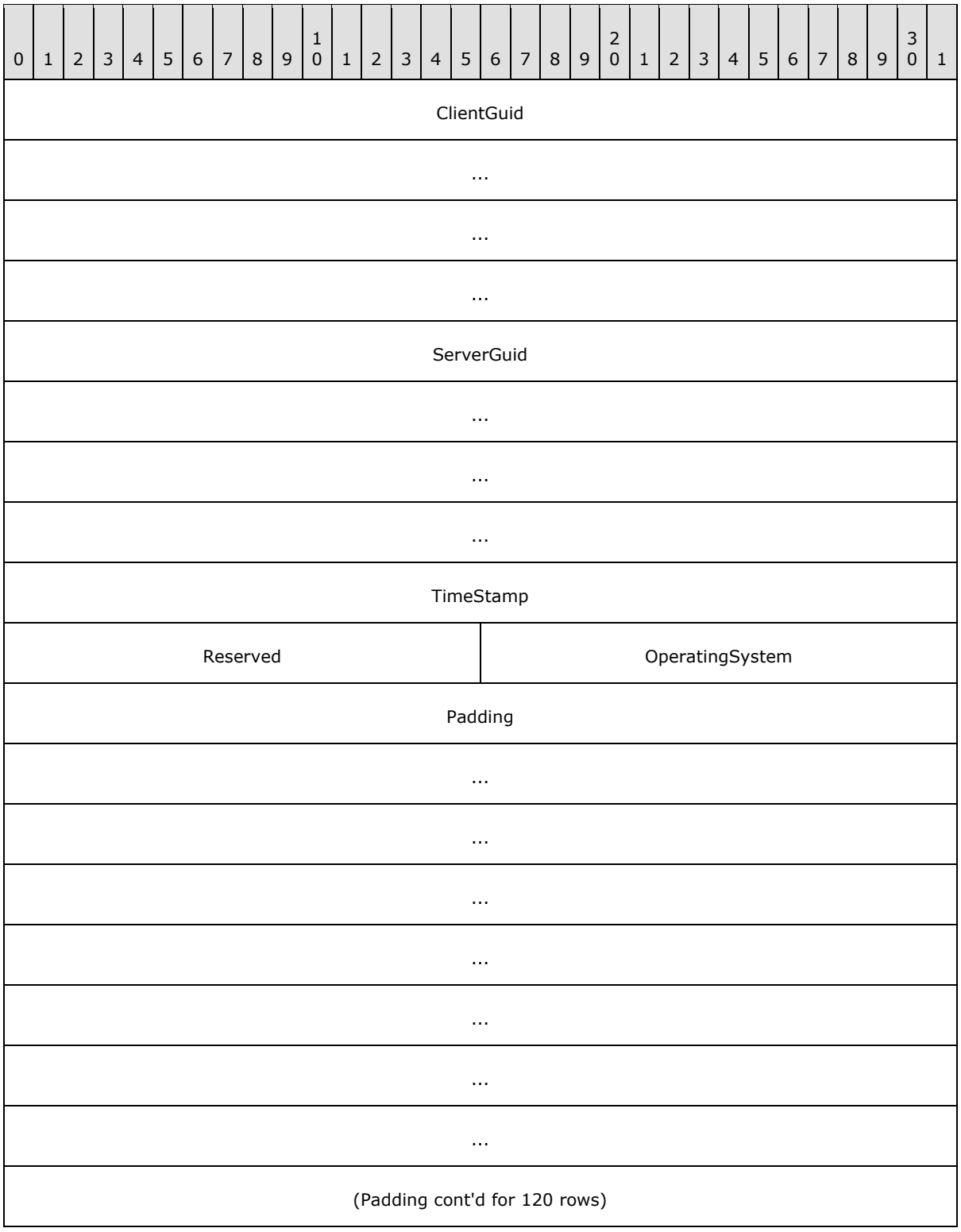
BaseHeader (16 bytes): A BaseHeader. The **BaseHeader.Flags.IN** field MUST be set to indicate that this packet is an internal message.

InternalHeader (4 bytes): An InternalHeader. The **InternalHeader.Flags.PT** field MUST be set to 0x2.

EstablishConnectionHeader (552 bytes): An EstablishConnectionHeader that contains information to identify the initiator and acceptor, a timestamp set by the initiator, and a flags field.

2.2.4.1 EstablishConnectionHeader

The EstablishConnectionHeader contains queue manager **GUIDs**, as specified in [MS-DTYP], that identify the initiator and acceptor, a timestamp set by the initiator, and flags.



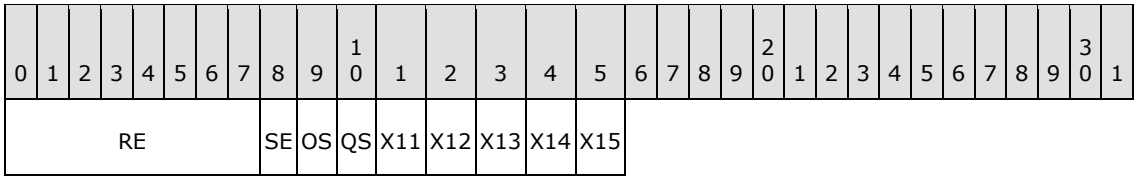
ClientGuid (16 bytes): The **GUID**, as specified in [MS-DTYP], for the queue manager of the initiator that is requesting the connection. In a connection request, the initiator sets this field. In a connection response, the acceptor sets this field to the queue manager **GUID** provided in the connection request.

ServerGuid (16 bytes): The **GUID**, as specified in [MS-DTYP], that identifies the queue manager of the acceptor responding to the connection request. In a connection request, the initiator MUST specify the acceptor queue manager **GUID**, as specified in [MS-DTYP], or zero fill this field if a direct format name is used. In a connection response, the acceptor MUST set this field to the **GUID** provided in the connection request or, if a direct format name was used, sets the field to the acceptor queue manager **GUID**.

TimeStamp (4 bytes): A 32-bit unsigned integer that identifies when the connection request was made. This value represents the number of milliseconds since the device booted. In a connection request, the initiator sets this field. In a connection response, the acceptor sets this field to the timestamp provided in the connection request.

Reserved (2 bytes): A 16-bit unsigned integer field reserved for future use. The initiator MUST set this field to 0x0000, and the acceptor MUST ignore it on receipt.

OperatingSystem (2 bytes): A 16-bit unsigned integer field containing flags related to the connection request. The field MUST be set to a combination of the values below:



RE (1 byte): This field is reserved. MUST be set to 0x0F.

SE (1 bit): Session flag. This bit MUST be set to 1 to indicate a new session.

OS (1 bit): Indicates the type of the initiator operating system. This field MUST be set to a value specified below:<20>

Value	Meaning
0x0	Initiator operating system is not a server.
0x1	Initiator operating system is a server.

QS (1 bit): Quality of Service (QOS) flag. This field SHOULD be set to a value specified below:<21>

Value	Meaning
0x0	None.
0x1	Indicates that the underlying transport supports Guaranteed Quality of Service (GQoS). Details are as specified in [RFC2212].

X11 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X12 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X13 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

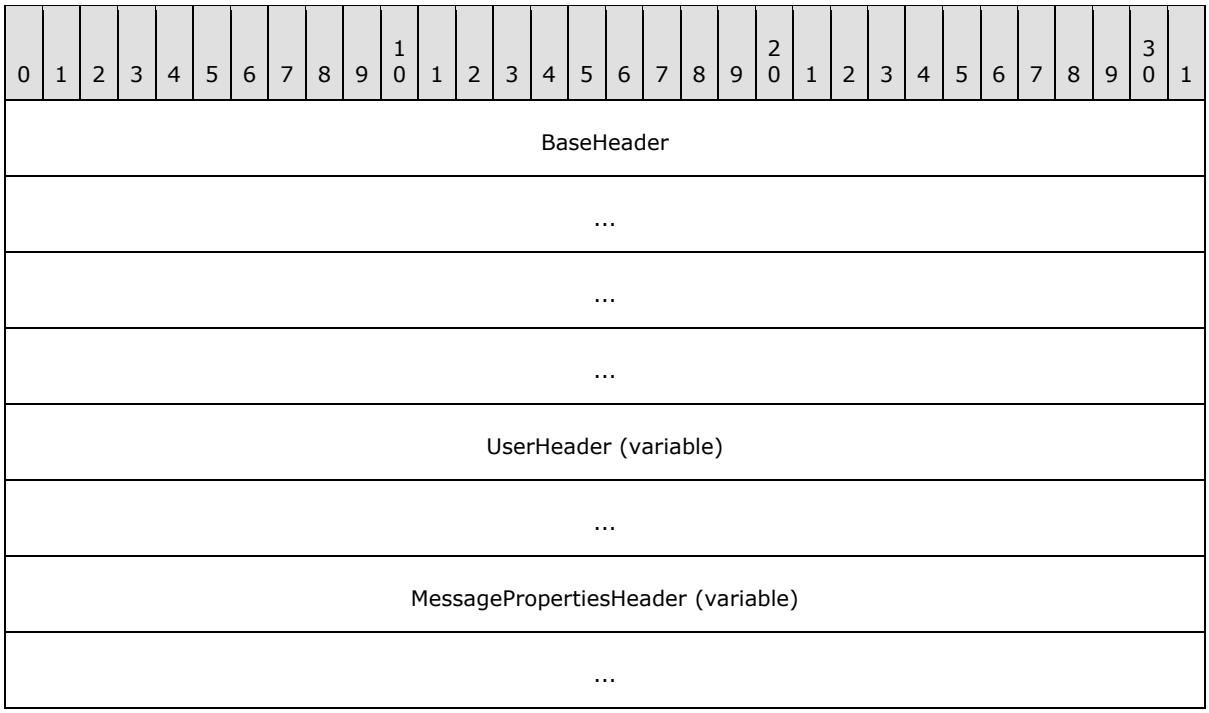
X14 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X15 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

Padding (512 bytes): A fixed-length array of 512 bytes of padding to fill the remainder of the EstablishConnectionHeader packet. Each byte of this array MUST be filled with the value 0x7A.

2.2.5 OrderAck Packet

The OrderAck Packet contains a stand-alone transactional acknowledgment packet. The packet can represent a positive or negative (for example, quota exceeded) acknowledgment that is sent to the original sender when a transactional message is delivered to the destination queue. The MessageClass field of the contained MessagePropertiesHeader defines the type of acknowledgment. The OrderAck Packet is an end-to-end acknowledgment.



BaseHeader (16 bytes): A BaseHeader. The **BaseHeader.Flags** field MUST have all bits set to 0x0.

UserHeader (variable): A UserHeader. The **UserHeader.Flags.DQ** field MUST be set to 0x7 and all other bits set to 0x0. The **DestinationQueue** field MUST be set to a direct format name in the format specified by the following ABNF rules:

```
orderQueueName = ("TCP:" ip-address / "SPX:" ipx-address )
                              "\PRIVATE$\order_queue"
ip-address=(IPv6address / IPv4address) ; as defined in [RFC3986]
ipx-address= 8HEXDIG "." 12HEXDIG ; network, node
```

The use of TCP or SPX will depends on whether a TCP or SPX transport is supported.<22> The value ip-address/ipx-address MUST represent the IP/IPX address of the queue manager to receive the message.

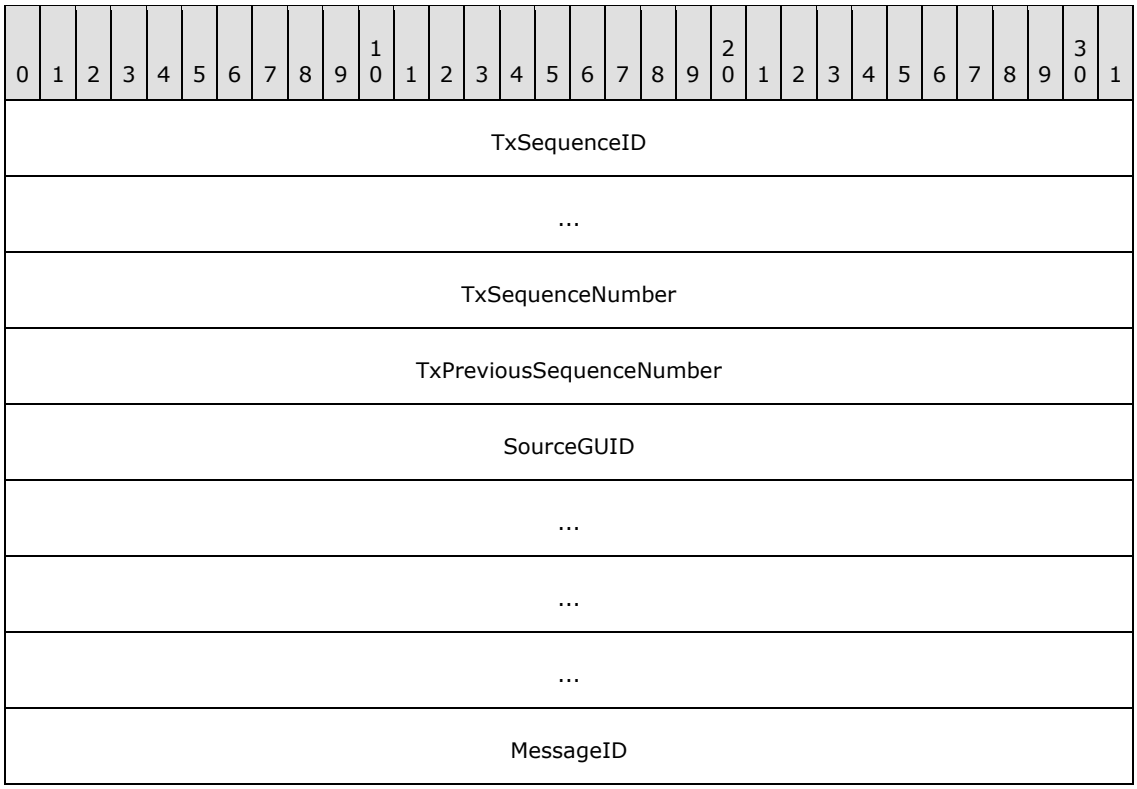
MessagePropertiesHeader (variable): A MessagePropertiesHeader. The **MessageData.Label** field MUST be set to "QM Ordering Ack". The **MessageSize** field MUST be set to 0x00000024. The **Flags** field MUST have all bits set to 0x0.

For a positive acknowledgment, the **MessageClass** field MUST be set to MQMSG_CLASS_ORDER_ACK. For a negative acknowledgment, the **MessageClass** field MUST be set to one of the following message class identifiers:

- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q
- MQMSG_CLASS_NACK_Q_DELETED
- MQMSG_CLASS_NACK_ACCESS_DENIED
- MQMSG_CLASS_NACK_BAD_ENCRYPTION
- MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER
- MQMSG_CLASS_NACK_BAD_SIGNATURE

For more details about message class identifiers, see section 2.2.1.5.

The **BodyType** field MUST be set to the value VT_ARRAY, which is specified in [MS-MQMQ]. The **MessageBody** field MUST be in the format described below:



TxSequenceID (8 bytes): A transactional sequence identifier, TxSequenceID. This value MUST be set to the transactional sequence identifier of the message being acknowledged. For more details, see the definition of TxSequenceID in section 2.2.1.2.

TxSequenceNumber (4 bytes): A 32-bit unsigned integer specifying a transactional sequence number that represents the order of a message within a transactional sequence. This value MUST be set to the transactional sequence number of the message being acknowledged. This field has a valid range from 0x00000001 to 0xFFFFFFFF.

TxPreviousSequenceNumber (4 bytes): A 32-bit unsigned integer specifying a transactional sequence number. This value MUST be set to the transactional sequence number of the previous message received in the transactional sequence. This field has a valid range from 0x00000001 to 0xFFFFFFFF.

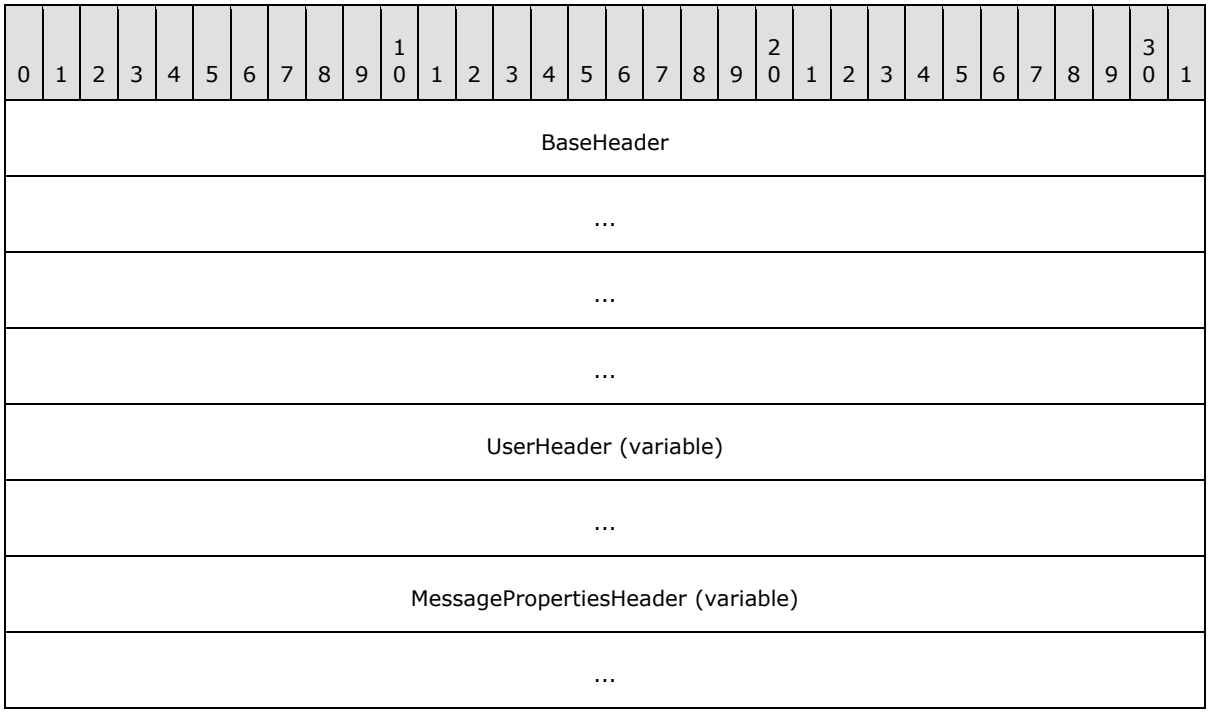
SourceGUID (16 bytes): This field MUST be the **GUID** of the source queue manager, as specified in [MS-DTYP]. This value is the **GUID**, as specified in [MS-DTYP], of the queue manager that originated the message being acknowledged.

MessageID (4 bytes): A 32-bit unsigned integer specifying a message identifier. This value MUST be set to the UserMessage.UserHeader.MessageID of the message being acknowledged.

2.2.6 FinalAck Packet

The FinalAck Packet contains a stand-alone transactional acknowledgment message that is sent to the original sender when a transactional message is removed from the destination queue that has the **UserHeader.Flags.JP** field is set to 0x2.

The packet can represent a positive or negative acknowledgment. The **MessageClass** field of the contained MessagePropertiesHeader packet defines the type of acknowledgment. The FinalAck Packet is an end-to-end acknowledgment.



BaseHeader (16 bytes): A BaseHeader. The **BaseHeader.Flags** field MUST have all bits set to 0x0.

UserHeader (variable): A UserHeader. The **UserHeader.Flags.DQ** field MUST be set to 0x7, the **UserHeader.Flags.DM** field set to 0x1, and all other bits set to 0x0. The **DestinationQueue** field MUST be set to a direct format name in the format specified by the following ABNF rule:

```
orderQueueName = ("TCP:" ip-address / "SPX:" ipx-address )
                 "\PRIVATE$\order_queue"
ip-address=(IPv6address / IPv4address) ; as defined in [RFC3986]
ipx-address= 8HEXDIG "." 12HEXDIG ; network.node
```

The use of TCP or SPX depends on whether TCP or SPX transport is supported.<23> The value ip-address/ipx-address MUST represent the IP/IPX address of the queue manager to receive the message.

MessagePropertiesHeader (variable): A MessagePropertiesHeader. The **MessageData.Label** field MUST be set to "QM Final Ack". The **MessageSize** field MUST be set to 0x00000024. The **Flags** field MUST have all bits set to 0x0.

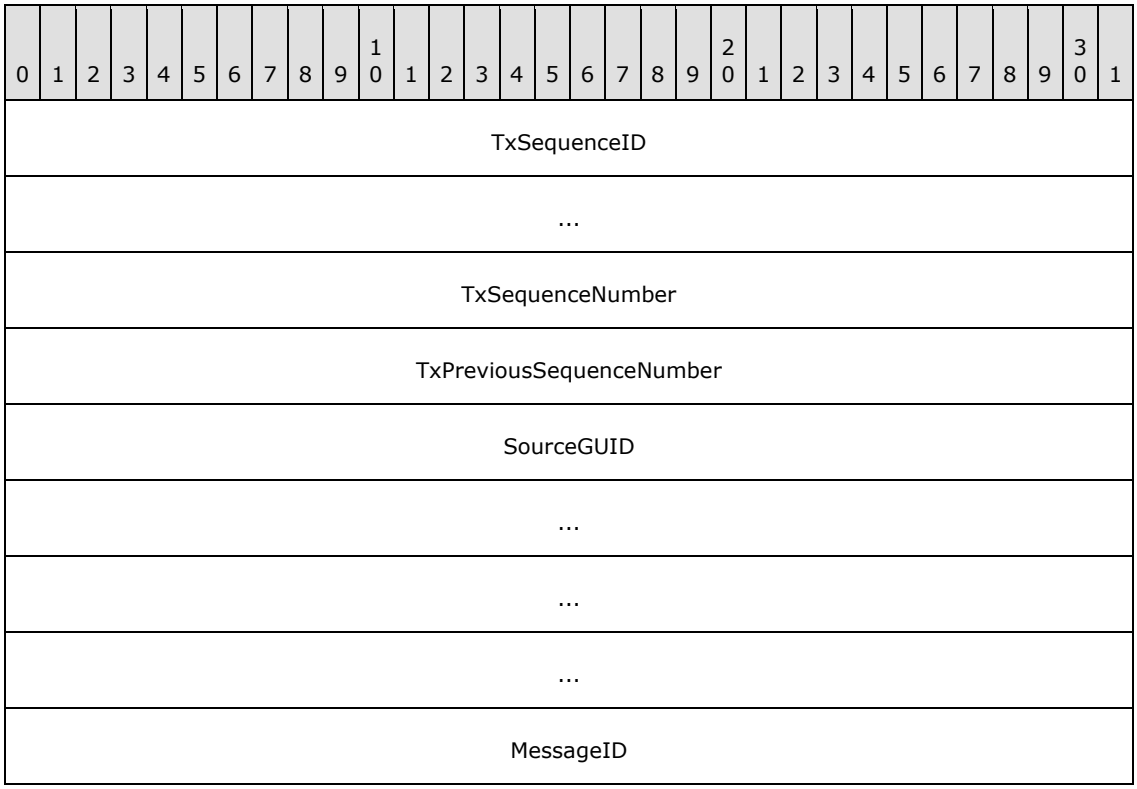
The **Flags** field MUST have all bits set to 0x0.

For a positive acknowledgment, the **MessageClass** field MUST be set to MQMSG_CLASS_ACK_RECEIVE. For a negative acknowledgment, the **MessageClass** field MUST be set to one of the following message class identifiers:

- MQMSG_CLASS_NACK_Q_DELETED
- MQMSG_CLASS_NACK_Q_PURGED
- MQMSG_CLASS_NACK_RECEIVE_TIMEOUT

For more details on message class identifiers, see section 2.2.1.5.

The **BodyType** field MUST be set to the value VT_ARRAY, which is specified in [MS-MQMQ]. The **MessageBody** field MUST be in the format described below:



TxSequenceID (8 bytes): A TxSequenceID specifying a transactional sequence identifier. This value MUST be the transactional sequence identifier of the message being acknowledged. For more details, see the definition of TxSequenceID in section 2.2.1.2.

TxSequenceNumber (4 bytes): A 32-bit unsigned integer specifying a transactional sequence number that represents the order of a message within a transactional sequence. This value MUST be set to the transactional sequence number of the message

being acknowledged. This field has a valid range from 0x00000001 to 0xFFFFFFFF, inclusive.

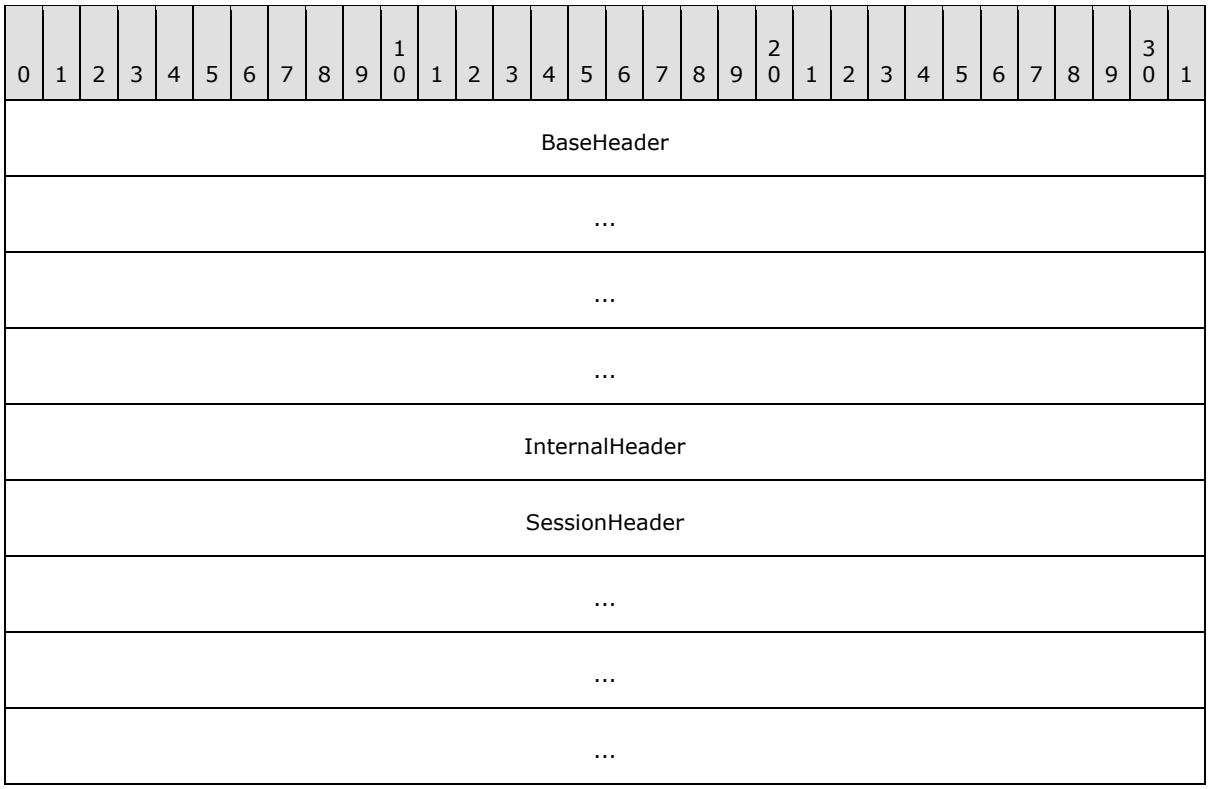
TxPreviousSequenceNumber (4 bytes): A 32-bit unsigned integer specifying a transactional sequence number. This value MUST be set to the transactional sequence number of the previous message received in the transactional sequence. This field has a valid range from 0x00000001 to 0xFFFFFFFF, inclusive.

SourceGUID (16 bytes): The **GUID** for the source queue manager GUID.

MessageID (4 bytes): A 32-bit unsigned integer specifying a message identifier. This value MUST be set to the **UserMessage.UserHeader.MessageID** of the message being acknowledged. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

2.2.7 SessionAck Packet

The SessionAck Packet contains a session acknowledgment. This packet acknowledges express and recoverable UserMessage Packets that have been received on the session.



BaseHeader (16 bytes): A BaseHeader. The **BaseHeader.Flags.IN** and **BaseHeader.Flags.SH** MUST be set.

InternalHeader (4 bytes): An InternalHeader. The **InternalHeader.Flags.PT** field MUST be set to 0x1.

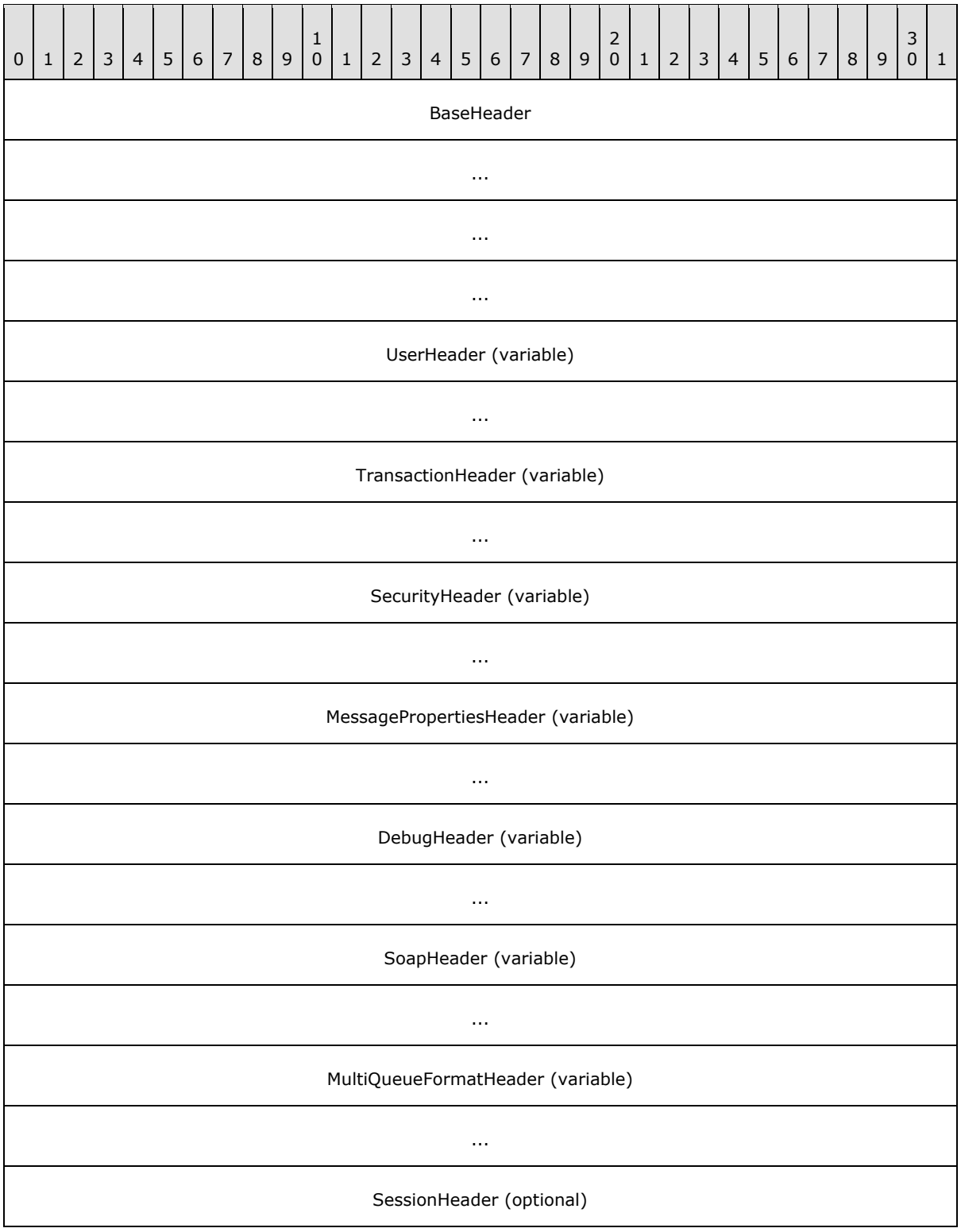
SessionHeader (16 bytes): A SessionHeader that contains acknowledgment and window size information.

A session acknowledgment can also be included within a UserMessage Packet. For more information about session acknowledgments, see section 3.1.1.4.1.

2.2.8 UserMessage Packet

A UserMessage Packet always contains an entire message. The UserMessage Packet is used to communicate application-defined and administration acknowledgment messages between a sender and receiver.

A UserMessage Packet contains a number of required headers and can contain additional optional headers. The required headers that MUST appear in all UserMessage Packets are: BaseHeader, UserHeader, and MessagePropertiesHeader. Optional headers include: TransactionHeader, SecurityHeader, DebugHeader, SoapHeader, and MultiQueueFormatHeader.



...
...
...

BaseHeader (16 bytes): A BaseHeader packet. The **BaseHeader.Flags.IN** field MUST NOT be set.

UserHeader (variable): A UserHeader that contains source and destination queue information.

TransactionHeader (variable): An optional TransactionHeader that contains flags and sequence information for the packet. This header MUST be present when a message is transactional. This header MUST be present when **UserHeader.Flags.TH** is set.

SecurityHeader (variable): An optional SecurityHeader. This header MUST be present when the sender of the packet requests authentication. This header MUST be present when **UserHeader.Flags.SH** is set.

MessagePropertiesHeader (variable): A MessagePropertiesHeader. This header MUST be present when **UserHeader.Flags.MP** is set.

DebugHeader (variable): A optional DebugHeader. This header specifies the queue where trace messages are sent. This header MUST be included when the UserMessage Packet packet contains a SoapHeader or MultiQueueFormatHeader. If this packet is present, then **BaseHeader.Flags.DH** MUST be set.

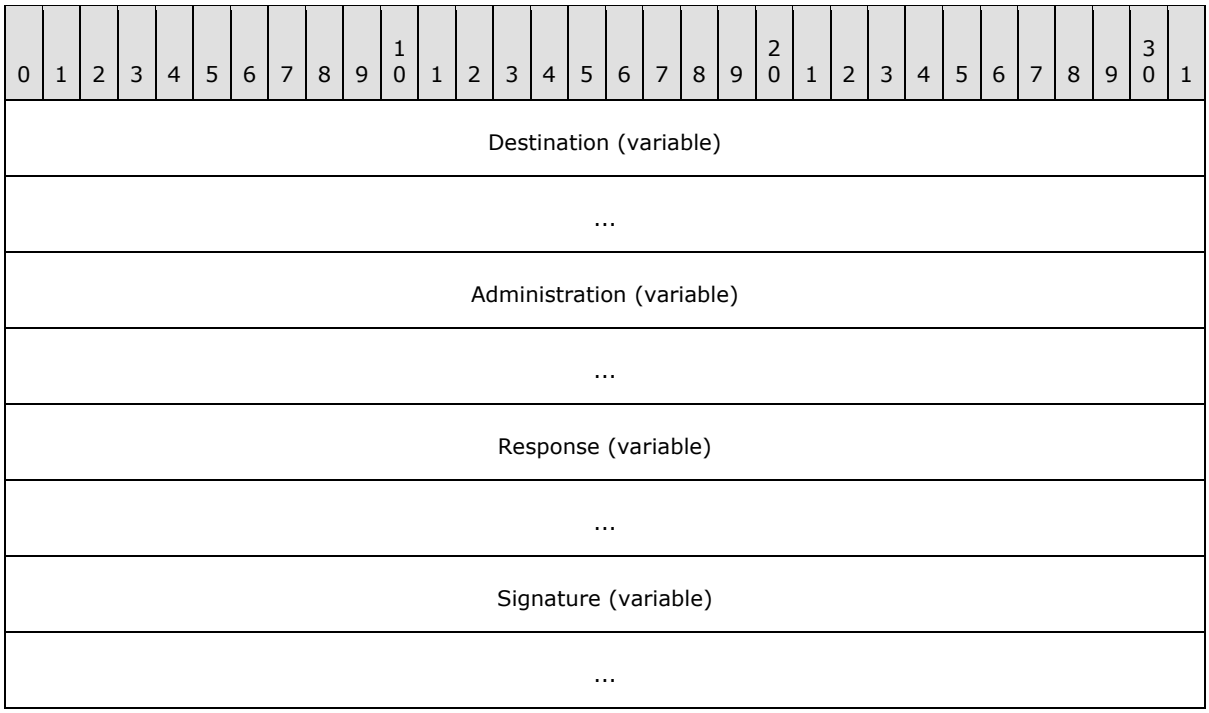
SoapHeader (variable): A optional SoapHeader.

MultiQueueFormatHeader (variable): An optional MultiQueueFormatHeader packet that is included when a message is destined for multiple queues. This header MUST be present when **UserHeader.Flags.MQ** is set.

SessionHeader (16 bytes): An optional SessionHeader. The inclusion of this header is an optimization in which the SessionHeader is placed in an outgoing UserMessage Packet, rather than sending it in a separate SessionAck Packet. A sender MAY choose to include this header to reduce the number of SessionAck Packets sent. A receiver MUST process a SessionHeader that is included in a UserMessage Packet. This header MUST be present when **BaseHeader.Flags.SH** is set.

2.2.8.1 MultiQueueFormatHeader

The optional MultiQueueFormatHeader is used when a message is destined for multiple queues.<24> When an application-layer message is sent using a multiple-element format name, this header is added to the packet to list the destinations. The sending queue manager creates a separate UserMessage Packet for each destination and specifies the packet address in the UserHeader. The information in this header provides a list of all destinations that were sent the message in addition to associated administration and response queues.



Destination (variable): An MQFAddressHeader that specifies the name of one or more destination queues. This field provides a list of all queues that were sent a copy of this UserMessage Packet.

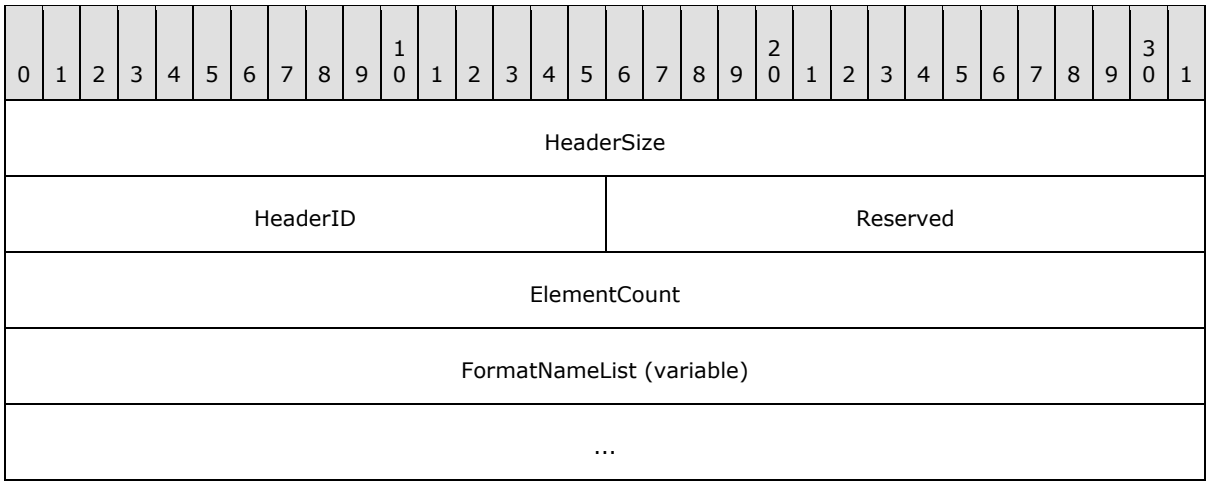
Administration (variable): An MQFAddressHeader that specifies the name of one or more administration queues.

Response (variable): An MQFAddressHeader that specifies the name of one or more response queues.

Signature (variable): An MQFAddressHeader that specifies a signature for the packet.

2.2.8.2 MQFAddressHeader

The MQFAddressHeader is used to specify multiple destination queue format names.



HeaderSize (4 bytes): A 32-bit unsigned integer that specifies the size of the header. This value MUST contain the size, in bytes, of this header including the variable data. This field has a valid range from 0x00000020 to 0xFFFFFFFF, inclusive.

HeaderID (2 bytes): A 16-bit unsigned integer that specifies an identifier for this header. This field MUST be set to one of the following values based on the header designation:

Value	Meaning
0x0064	Destination
0x00C8	Admin
0x012C	Response
0x015E	Signature

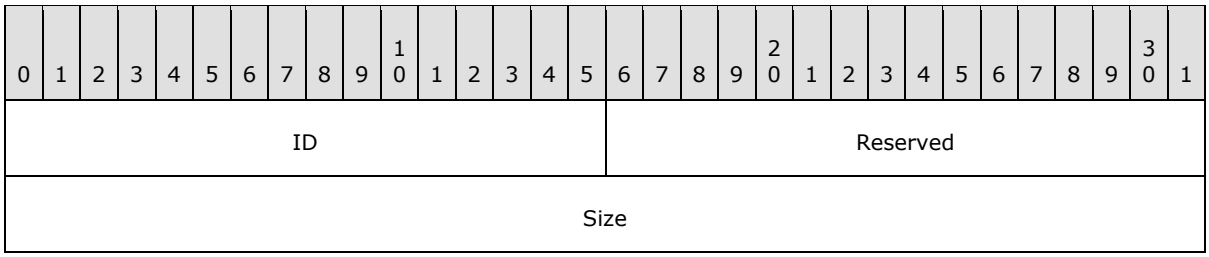
Reserved (2 bytes): A 16-bit unsigned integer field that is reserved for alignment. The sender SHOULD set this field to 0x0000, and the receiver MUST ignore it on receipt.

ElementCount (4 bytes): A 32-bit unsigned integer field that MUST be set to the number of elements in the **FormatNameList** field. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

FormatNameList (variable): An MQFFormatNameElement that contains a list of queue format names. This field MUST contain a list of MQFFormatNameElement data structures. The array MUST contain the number of elements specified by the **ElementCount** field.

2.2.8.3 MQFSignatureHeader

The MQFSignatureHeader is a signature used in the MultiQueueFormatHeader.



ID (2 bytes): A 16-bit unsigned short integer value that MUST be set to 0x015E.

Reserved (2 bytes): A 16-bit unsigned short integer field that is reserved for alignment. The sender SHOULD set this field to 0x0000, and the receiver MUST ignore it on receipt.

Size (4 bytes): A 32-bit unsigned integer field that MUST be set to the value 0x00000000.

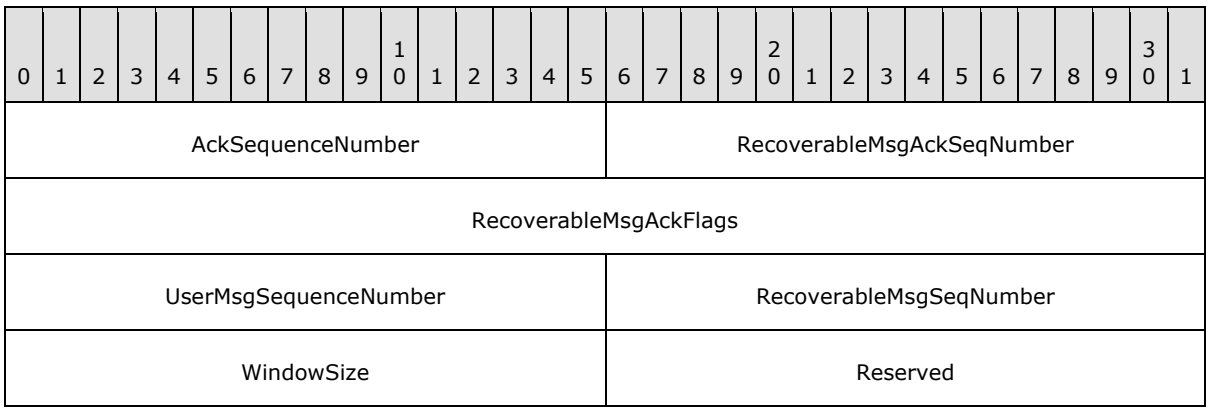
2.2.8.4 SessionHeader

The SessionHeader is used to acknowledge express and recoverable UserMessage Packets received by the protocol. This header is present in stand-alone SessionAck Packet and is optional in a UserMessage Packet.

This header contains a session acknowledgment. For more details, see sections 3.1.1.5, and 3.1.1.4.1.

The set of UserMessage Packets sent on a session represent a message sequence. There is a local-to-remote and remote-to-local sequence. These message sequences exist for the lifetime of the session. The local and remote protocols maintain counts of the UserMessage Packets sent and received. A message is associated with a sequence number that corresponds to its position within the sequence. Sequence numbers begin with 1 and increment by 1 with each subsequent message. For example, the third message sent on a session has a sequence number of 3.

The protocol also maintain a count of recoverable UserMessage Packets sent and associates recoverable sequence numbers with those messages. For example, the fifth recoverable message sent on a session has a sequence number of 5.



AckSequenceNumber (2 bytes): A 16-bit unsigned short integer that specifies a count of messages received. This field MUST be set to the count of UserMessage Packets received on this session. This field acknowledges all messages up to and including the specified sequence number. This field has a valid range from 0x0001 to 0xFFFF, inclusive.

RecoverableMsgAckSeqNumber (2 bytes): A 16-bit unsigned short integer that specifies a recoverable message sequence number. This field MUST be set to the lowest unacknowledged recoverable message sequence number that has been written to disk. If no recoverable messages have been received by the receiver since the last SessionHeader was sent, this field MUST be set to 0. This field has a valid range from 0x0000 to 0xFFFF, inclusive.

RecoverableMsgAckFlags (4 bytes): A 32-bit unsigned long integer bit field representing messages. This bit field represents up to 32 recoverable messages that are being acknowledged as written to disk. Bit 0 of this field represents the message whose sequence number is specified in the **RecoverableMsgAckSeqNumber** field. A given bit k of this field represents a recoverable message with a sequence number of **RecoverableMsgAckSeqNumber** + k. The corresponding bit for a message that has been successfully written to disk MUST be set in the bit field.

UserMsgSequenceNumber (2 bytes): A 16-bit unsigned short integer that is the count of messages sent. This field MUST be set to the count of UserMessage Packets sent on this session. This value MUST be 0 if no UserMessage Packets have been sent. This field has a valid range from 0x0000 to 0xFFFF.

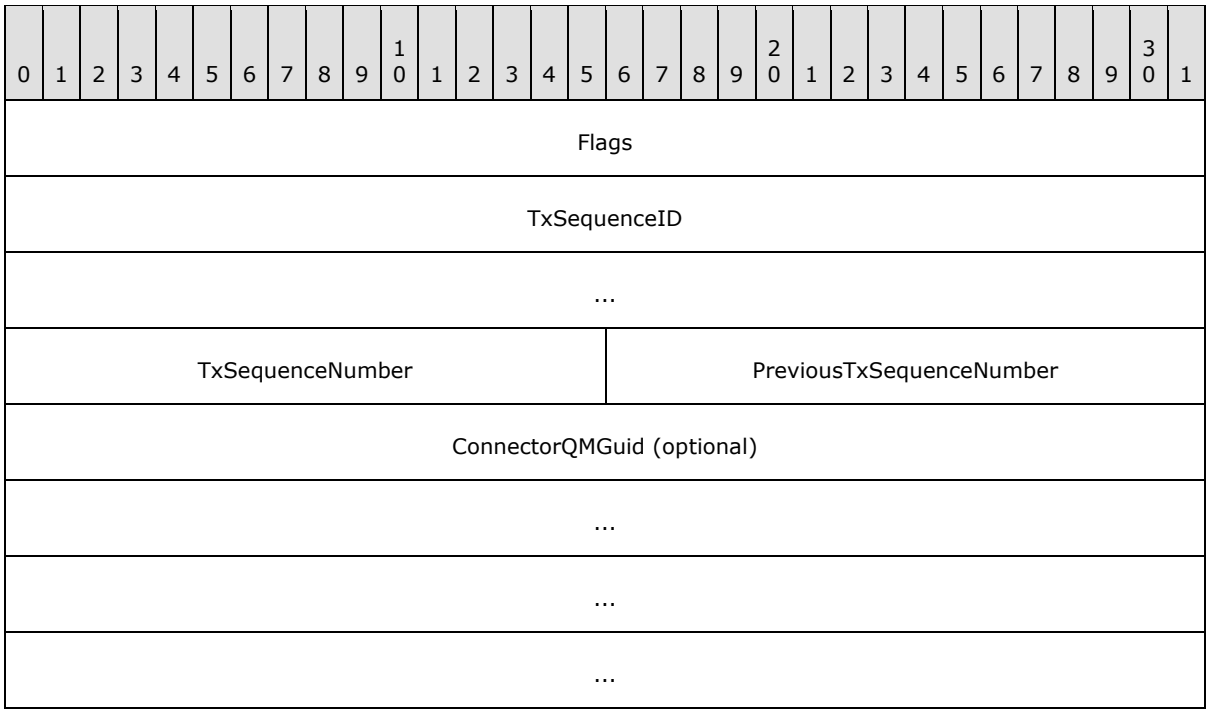
RecoverableMsgSeqNumber (2 bytes): A 16-bit unsigned short integer that is the count of recoverable messages sent. This field MUST be set to the count of recoverable UserMessage Packets sent on this session. This value MUST be 0 if no recoverable UserMessage Packets have been sent. This field has a valid range from 0x0000 to 0xFFFF.

WindowSize (2 bytes): A 16-bit unsigned short integer field that specifies the acknowledgment window size. The window size controls the frequency at which the protocol sends acknowledgment packets. The value of **WindowSize** SHOULD be set to value 0x0020.<25> The window helps avoid network collisions and can be used to throttle a low-resource condition.<26> This field has a valid range from 0x0001 to 0xFFFF.

Reserved (2 bytes): A 16-bit unsigned short integer field that is reserved for future use. The sender MUST set this field to 0x0000, and the receiver MUST ignore it on receipt.

2.2.8.5 TransactionHeader

The TransactionHeader packet contains sequence information for a transactional message. The presence of this packet in a UserMessage Packet indicates that the message contained in the packet is transactional.

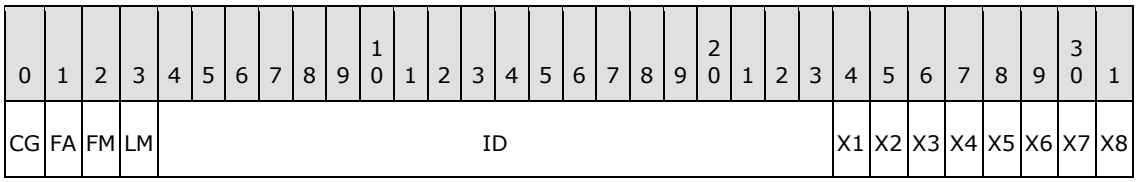


Flags (4 bytes): A 32-bit unsigned long integer that contains a set of options that provide additional information about the packet. Any combination of these values is acceptable unless otherwise noted in the table below.

Note that transactional capture of messages is an implementation detail of a queue manager. This protocol does not mandate the implementation details of the transactional capture of messages as long as the external behavior of a queue manager is consistent with that specified in this document. The **Flags.FM**, **Flags.LM**, and **Flags.ID** fields correspond to values specific to that implementation.

Any value not specified in the table MUST be treated as an error by closing the session.

The value SHOULD be set to a combination of the following values:<27>



CG (1 bit): A bit that specifies if the **ConnectorQMGuid** field contains a connector queue manager GUID. If set, the **ConnectorQMGuid** field MUST contain a GUID.

FA (1 bit): A bit that specifies if a FinalAck Packet is required. If set, a FinalAck Packet MUST be sent to the original sender when the message is removed from the destination queue.

FM (1 bit): A bit that specifies whether the message is the first one sent within the context of a transaction. If set, this value indicates the message is the first one in a transaction.

LM (1 bit): A bit that specifies whether the message is the last one sent within the context of a transaction. If set, this value indicates the message is the last one in a transaction.

ID (20 bits): An array of 20 bits that specifies an identifier to correlate this packet to the transaction under which it was captured. The protocol MUST generate a unique identifier for the transaction and assign all packets captured under the transaction to the value.

X1 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X2 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X3 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X4 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X5 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X6 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X7 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X8 (1 bit): An unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

TxSequenceID (8 bytes): A transactional sequence identifier, as specified in TxSequenceID. This value identifies the transactional sequence that the **TxSequenceNumber** and **PreviousTxSequenceNumber** are within. This value MUST be set to the current transaction sequence for the session. For more details, see section 2.2.1.2.

TxSequenceNumber (2 bytes): A 16-bit unsigned short integer that is the message sequence number within the TxSequenceID sequence. This field MUST be set to the value that represents the message position within the transactional sequence. The first message within a sequence is set to the value 1. This field has a valid range from 0x0001 to 0xFFFF.

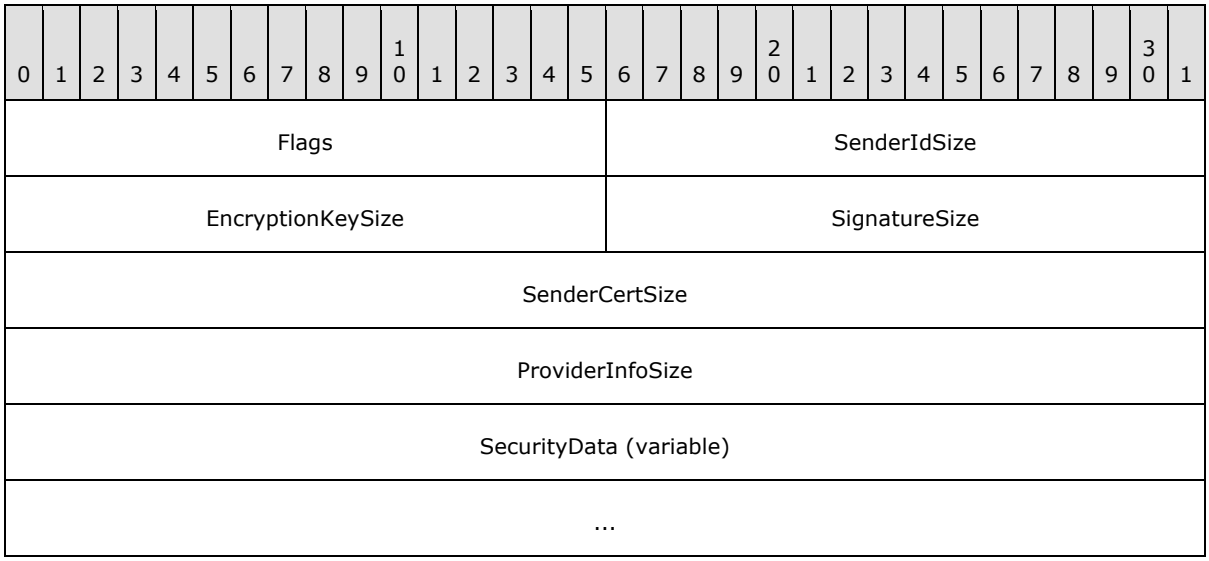
PreviousTxSequenceNumber (2 bytes): An 16-bit unsigned short integer that is the sequence number of the previous message in the TxSequenceIDsequence. This field MUST be set to the sequence number of the message that precedes this message in the transactional sequence. This value MUST be set to 0 if there is no previous message. This field has a valid range from 0x0001 to 0xFFFF.

ConnectorQMGuid (16 bytes): An optional field containing an application-defined **GUID**, as specified in [MS-DTYP]. If **Flags.CG** is set, this field MUST be present and contain an

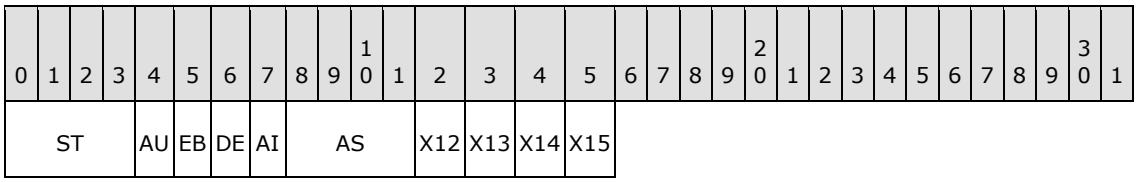
application-defined **GUID** value, as specified in [MS-DTYP]. The field can be used by application-level logic to associate a message in a connector queue with a connector service.

2.2.8.6 SecurityHeader

The SecurityHeader contains optional security information.



Flags (2 bytes): A 16-bit unsigned short integer that contains a set of options that provide additional information about the packet. Any combination of these values is acceptable unless otherwise noted in the table below.



ST (4 bits): Specifies the type of sender ID in the **SecurityData** field. This field MUST be set to one of the following values:

Value	Meaning
0x0	None. The **SecurityData.SecurityID** field is not present and the SenderIdSize field MUST be set to 0x0000.
0x1	SID. The **SecurityData.SecurityID** field MUST contain the sender application security identifier (SID).
0x2	**GUID**. The **SecurityData.SecurityID** field MUST contain the queue manager **GUID**.

AU (1 bit): Indicates whether the message is authenticated or not. If set, the **SecurityData.Signature** field MUST be set by the sender, and the receiver MUST authenticate the packet by verifying the **SecurityData.Signature** field.

Sender: The sender MUST compute a hash of the fields specified by the **Flags.AS** field by using the hash algorithm specified by the **MessagePropertiesHeader.HashAlgorithm** field. The **Signature** field MUST be set to the value of the hash encrypted using RSA and QMPrivateKey private key.

Receiver: The receiver MUST authenticate the signature by re-computing the hash and comparing it to the decrypted signature value. The sender MUST compute a hash of the fields specified by the **Flags.AS** field by using the hash algorithm specified by the **MessagePropertiesHeader.HashAlgorithm** field. This value MUST be compared to the **Signature** field value after it has been decrypted using RSA and the public key contained in the **SecurityData.SenderCert** certificate. If the values are not identical the packet MUST be ignored.

EB (1 bit): Indicates whether the body of the message is encrypted or not. If set, the **MessagePropertiesHeader.MessageData.MessageBody** field MUST be encrypted by the sender and decrypted by the receiver.

Sender: The sender MUST encrypt the **MessagePropertiesHeader.MessageData.MessageBody** field by using the encryption algorithm specified by **MessagePropertiesHeader.EncryptionAlgorithm** and a randomly generated encryption key. The **MessagePropertiesHeader.PrivacyLevel** field specifies the encryption key size that MUST be used. The size of the **MessageBody** field MUST be adjusted if the encrypted data is a different size than the original field. The **MessagePropertiesHeader.AllocationBodySize** MUST be set to the size of the encrypted data. The encryption key above MUST be encrypted using RSA and the public key of the destination queue manager in RemoteQMPublicKey.<28> The **SecurityData.EncryptionKey** field MUST be set to the resulting encrypted data and the **EncryptionKeySize** field set to the size of that data.

Receiver: The receiver MUST decrypt the **SecurityData.EncryptionKey** field by using RSA and the local queue manager private key in QMPrivateKey. Using the decrypted key, the receiver MUST decrypt the **MessagePropertiesHeader.MessageData.MessageBody** field by using the encryption algorithm specified by **MessagePropertiesHeader.EncryptionAlgorithm** and the decrypted **SecurityData.EncryptionKey** value. If decryption fails, the packet MUST be ignored.

DE (1 bit): Indicates whether the default cryptographic provider is used. When clear and **SignatureSize** is nonzero, the **SecurityData.ProviderName** MUST specify the name of the alternate provider.

AI (1 bit): Indicates if the **SecurityData** field is present. If set, the header MUST include a **SecurityData** field.

AS (4 bits): AS

Indicates the authentication signature type. This field MUST be set to one of the following values.

Value	Meaning
0x0	None.
0x1	Specifies the **SecurityData.EncryptionKey** value; MUST be a hash of the following

Value	Meaning
	packet field values in the specified order: 1. MessagePropertiesHeader.CorrelationID 2. MessagePropertiesHeader.ApplicationTag 3. MessagePropertiesHeader.MessageData.MessageBody 4. MessagePropertiesHeader.Label 5. UserHeader.ResponseQueue 6. UserHeader.AdminQueue
0x3	Specifies the SecurityData.EncryptionKey value; MUST be a hash of the following packet field values in the specified order: 1. MessagePropertiesHeader.CorrelationID 2. MessagePropertiesHeader.ApplicationTag 3. MessagePropertiesHeader.MessageData.MessageBody 4. MessagePropertiesHeader.Label 5. UserHeader.ResponseQueue 6. UserHeader.AdminQueue 7. UserHeader.SourceQueueManager 8. (**BYTE**)UserHeader.Flags.DM 9. (**BYTE**)BaseHeader.Flags.PR 10.(**BYTE**)((UserHeader.Flags.JP << 1) | UserHeader.Flags.JN) 11.(**BYTE**)(MessagePropertiesHeaders.Flags & 0x0F) 12.(**USHORT**)MessagePropertiesHeader.MessageClass 13.(**ULONG**)MessagePropertiesHeader.BodyType 14.UserHeader.ConnectorType 15.UserHeader.DestinationQueue
0x5	Specifies the **SecurityData.EncryptionKey** value; MUST be a hash of the following packet field values in the specified order: 1. MessagePropertiesHeader.CorrelationID 2. MessagePropertiesHeader.ApplicationTag 3. MessagePropertiesHeader.MessageData.MessageBody

Value	Meaning
	4. MessagePropertiesHeader.Label 5. UserHeader.ResponseQueue 6. UserHeader.AdminQueue 7. UserHeader.SourceQueueManager 8. (**BYTE**)UserHeader.Flags.DM 9. (**BYTE**)BaseHeader.Flags.PR 10.(**BYTE**)((UserHeader.Flags.JP << 1) | UserHeader.Flags.JN) 11.(**BYTE**)(MessagePropertiesHeaders.Flags & 0x0F) 12.(**USHORT**)MessagePropertiesHeader.MessageClass 13.(**ULONG**)MessagePropertiesHeader.BodyType 14.UserHeader.ConnectorType 15.MultiQueueFormatHeader.FormatNameList 16.MessagePropertiesHeader.MessageData.ExtensionData
0xB	Specifies that the **SecurityData.EncryptionKey** value MUST be a hash of the following packet field values in the specified order: 1. MessagePropertiesHeader.CorrelationID 2. MessagePropertiesHeader.ApplicationTag 3. MessagePropertiesHeader.MessageData.MessageBody 4. MessagePropertiesHeader.Label 5. UserHeader.ResponseQueue 6. UserHeader.AdminQueue The hash algorithm used to compute the **SecurityData.EncryptionKey** field is specified by the **MessagePropertiesHeader.HashAlgorithm** field. Fields in the preceding list that are cast to a (**BYTE**) MUST be hashed as separate **BYTE** values.

X12 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X13 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X14 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

X15 (1 bit): Unused bit field. SHOULD NOT be set when sent, and MUST be ignored when received.

SenderIdSize (2 bytes): A 16-bit unsigned short integer that specifies the size of the **SecurityData.SecurityID** field. This value MUST be set to the size, in bytes, of the security identifier in the **SecurityData.SecurityID** field. This field has a valid range from 0x0000 to 0xFFFF.

EncryptionKeySize (2 bytes): A 16-bit unsigned short integer that specifies the size of the **SecurityData.EncryptionKey** field. This value MUST be set to the size, in bytes, of the encryption key in the **SecurityData.EncryptionKey** field. This field has a valid range from 0x0000 to 0xFFFF.

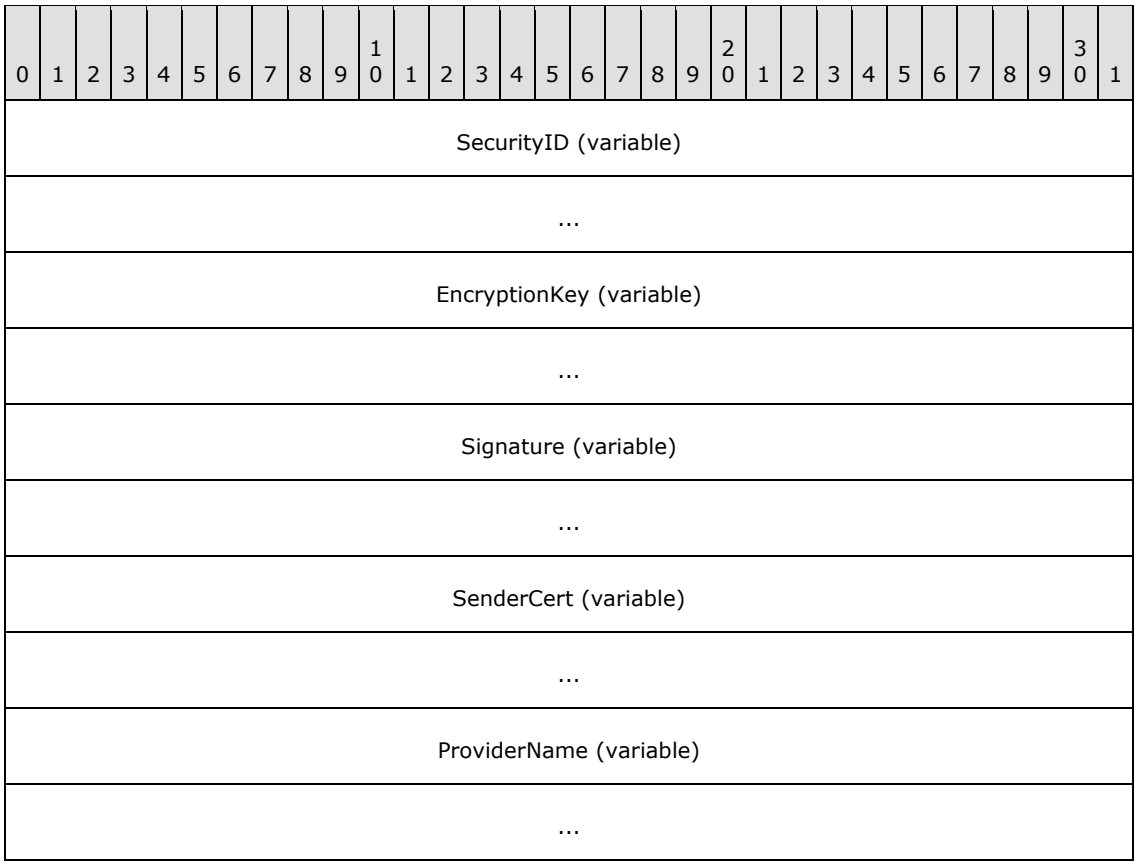
SignatureSize (2 bytes): A 16-bit unsigned short integer that specifies the size of the **SecurityData.SenderCert** field. This value MUST be set to the size, in bytes, of the sender certificate in the **SecurityData.SenderCert** field. This field has a valid range from 0x0000 to 0xFFFF.

SenderCertSize (4 bytes): A 32-bit unsigned long integer that specifies the size of the **SecurityData.Signature** field. This value MUST be set to the size, in bytes, of the sender signature in the **SecurityData.Signature** field. This field has a valid range from 0x00000000 to a value 0x0000FFFF.

ProviderInfoSize (4 bytes): A 32-bit unsigned long integer that specifies the size of the **SecurityData.ProviderInfo** field. This value MUST be set to the size, in bytes, of the security provider information in the **SecurityData.ProviderInfo** field. This field has a valid range from 0x00000000 to 0xFFFFFFFF.<29>

SecurityData (variable): An optional variable-length array of bytes containing additional security information. This field MUST contain the security information specified in the **Flags** field.

The data appears in the order specified below. Each field MUST be aligned up to the next 4-byte boundary. The size of each field is specified by the corresponding **SenderIdSize**, **EncryptionKeySize**, **SignatureSize**, **SenderCertSize**, and **ProviderInfoSize** fields. An item with a size of zero occupies no space in the SecurityData array. If all items have a size of 0, then the SecurityData field is 0 size.



SecurityID (variable): Sender security identifier (SID).

EncryptionKey (variable): Sender symmetrical encryption key.

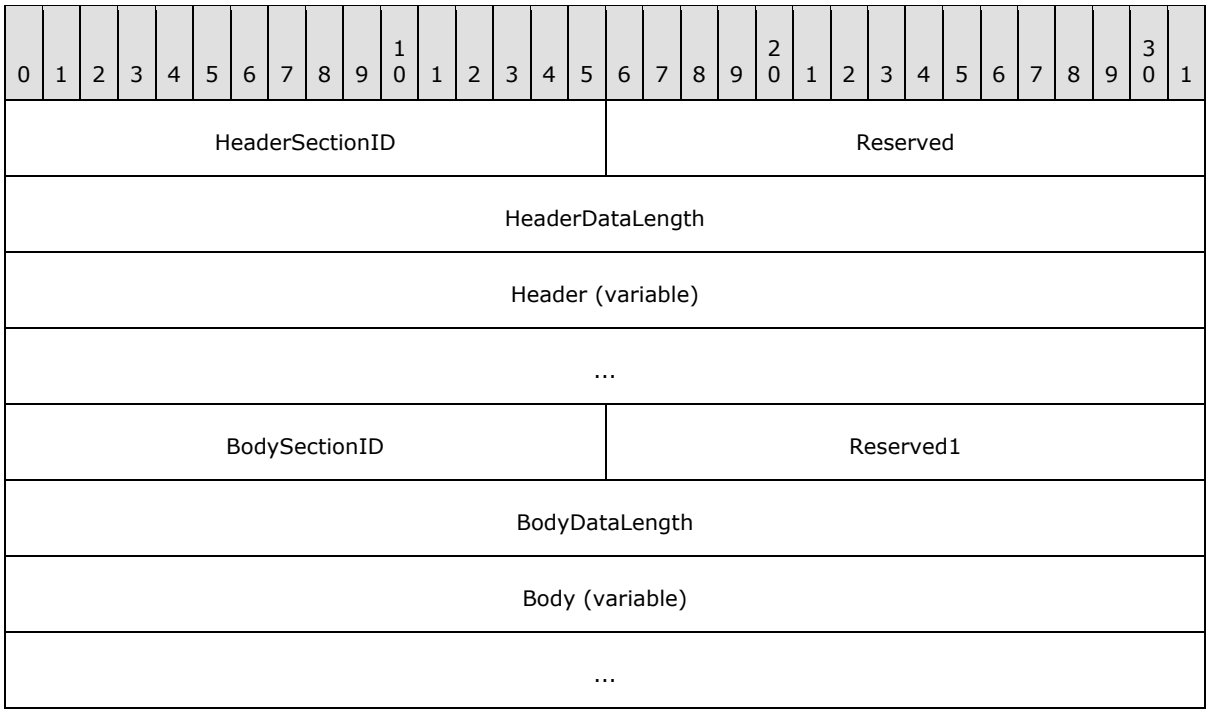
Signature (variable): The packet digital signature. The type of signature is specified by the **SecurityHeader.Flags.AS** field and the hash algorithm is specified by the **MessagePropertiesHeader.HashAlgorithm** field.

SenderCert (variable): Sender X.509 digital certificate. Details are as specified in [RFC3280].

ProviderName (variable): An application-defined value that specifies the name of the cryptographic provider that is used to generate the digital signature attached to the message. This value MAY be used when an application-defined signature is included in the header.

2.2.8.7 SoapHeader

The optional SoapHeader packet contains application-defined information. If the **UserHeader.Flags.HH** field is set, this field MUST be present in a UserMessage Packet. This field SHOULD NOT be used by the protocol.<30>



HeaderSectionID (2 bytes): A 16-bit unsigned short integer field that MUST be set to 0x0320.

Reserved (2 bytes): A 16-bit unsigned integer field that is reserved for future use. The sender SHOULD set this field to 0x0000, and the receiver MUST ignore it on receipt.

HeaderDataLength (4 bytes): A 32-bit unsigned long integer that specifies the length of the **Header** field. This field MUST be set to the number of elements in the Unicode **Header** field, including the null terminator. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

Header (variable): A null-terminated Unicode string. This field contains an application-defined string.

BodySectionID (2 bytes): A 16-bit unsigned short integer that MUST be set to 0x0384.

Reserved1 (2 bytes): A 16-bit unsigned short integer field reserved for future use. The sender SHOULD set this field to 0x0000, and the receiver MUST ignore it on receipt.

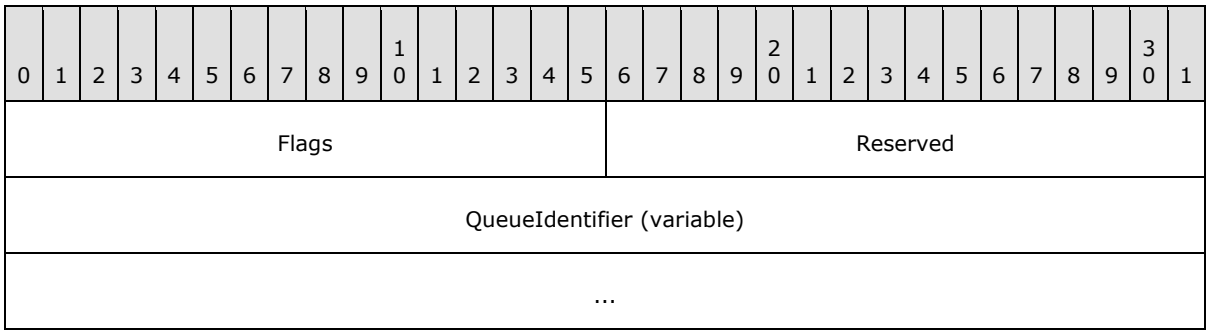
BodyDataLength (4 bytes): A 32-bit unsigned long integer that specifies the length of the Body field. This field MUST be set to the number of elements in the Unicode Body field, including the null terminator. This field has a valid range from 0x00000000 to 0xFFFFFFFF.

Body (variable): A null-terminated Unicode string. This field contains an application-defined string.

2.2.8.8 DebugHeader

The DebugHeader specifies the queue to receive trace messages for this UserMessage Packet. When tracing is enabled and a report queue is specified, report messages are sent to the report queue specified in this header each time the UserMessage Packet leaves or arrives at a queue manager.

The DebugHeader is optional, except if a UserMessage Packet contains a SoapHeader or MultiQueueFormatHeader, in which case the DebugHeader MUST be included.<31> The **BaseHeader.Flags.DH** MUST be set only if the DebugHeader is present.



Flags (2 bytes): A 16-bit unsigned short integer field that provides bitflags containing additional information about the packet.

QT (2 bits): Specifies the queue type. This field MUST be set to one of the following values:

Value	Meaning
0x0	No queue. The length of the **QueueIdentifier** field is 0.
0x1	Public queue. The **QueueIdentifier** field contains a 16-byte queue **GUID**, as specified in [MS-DTYP].

X2 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X3 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X4 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X5 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X6 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X7 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X8 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X9 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X10 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X11 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X12 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X13 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

X14 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

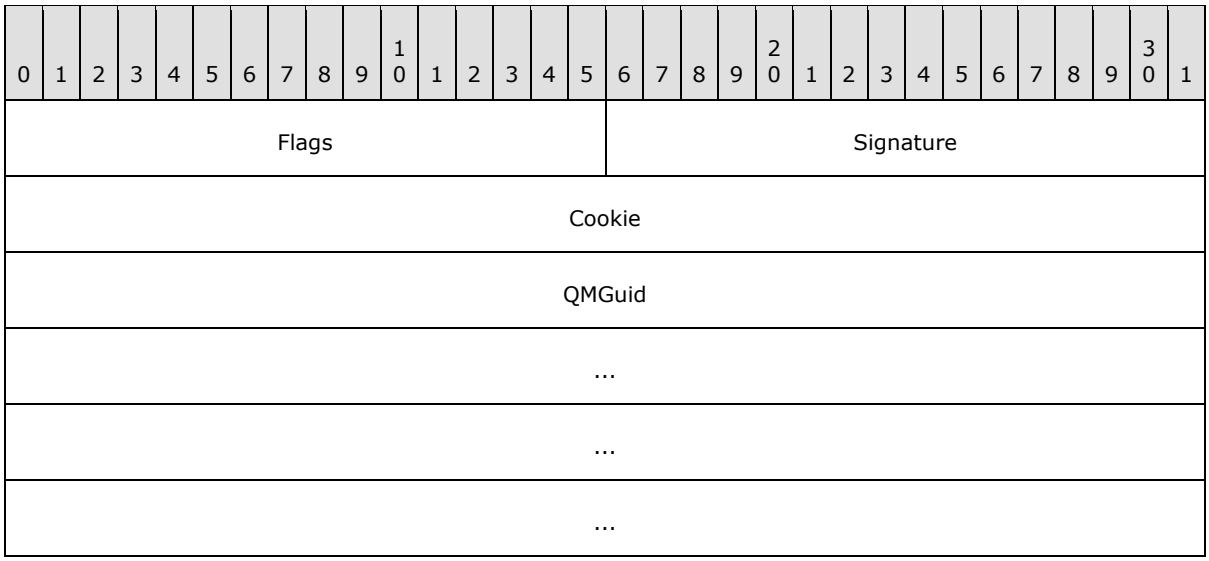
X15 (1 bit): Unused bitfield. SHOULD NOT be set when sent, and MUST be ignored when received.

Reserved (2 bytes): A 16-bit unsigned integer field that is reserved for future use. The sender SHOULD set this field to 0x0000, and the receiver MUST ignore it on receipt.

QueueIdentifier (variable): A variable-length byte array containing the identifier of the queue that is used to store trace messages. This field's length depends on the queue type specified in the **Flags** field.

2.2.9 Ping Message

The Ping Message is sent from an initiator to an acceptor to determine availability. An acceptor responds to a Ping request by sending a Ping Message back to the initiator. A Ping Message is used to determine if an acceptor is available and can accept a binary protocol sequence connection.



Flags (2 bytes): A 16-bit unsigned short integer field that provides additional information about the packet.

Fields marked X are unused. They MAY be set when sent. They MUST be ignored when received.<32>

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5
RC	RF	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Where the bits are defined as:

Value	Description
RC	Specifies if the initiator is an **independent client**. An initiator MUST set this field if it is an independent client; otherwise, it MUST NOT be set. An acceptor MUST set this field to the value of this field in the Ping request from the initiator.
RF	An acceptor MUST set this field if it would currently refuse a protocol session over TCP/SPX from this initiator; otherwise, the field MUST be clear if a protocol session would be accepted. An initiator MUST clear this field.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.

Value	Description
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.
X	Unused bit field. MAY be set when sent, and MUST be ignored when received.

Signature (2 bytes): A 16-bit unsigned short integer field that identifies the packet as a Ping Message packet. This value MUST be set to 0x5548. The signature is transmitted as an unsigned short in big-endian order. A receiver MUST ignore the packet if the signature is not set to this value.

Cookie (4 bytes): A 32-bit unsigned long integer that specifies a value used to correlate request and reply Ping Message packets. This value is generated by the initiator to uniquely identify the Ping request. The initiator SHOULD set this to a value that provides uniqueness across outstanding Ping requests.<33> This field has a valid range from 0x00000000 to 0xFFFFFFFF.

When sending a Ping Message response, an acceptor MUST set this field to the Cookie value from the received Ping request packet. When an initiator receives a Ping Message response it uses the Cookie field to correlate it to a Ping Message request. An initiator MUST disregard a Ping Message response packet that contains a Cookie that does not correspond to the Ping Message request Cookie it has sent.

QMGuid (16 bytes): The **GUID**, as specified in [MS-DTYP], of the sender's queue manager. This value MUST be set to the queue manager **GUID**, as specified in [MS-DTYP], of the sender, whether that be the initiator or acceptor.

3 Protocol Details

The Message Queuing (MSMQ): Message Queuing Binary Protocol is often described as a communication between a "client" and "server"; however, for the purpose of this section the terms "local host" and "remote/destination host" are used to refer to these roles, respectively. During session initialization, these roles are referred to as "initiator" and "acceptor," respectively. After initialization, the protocol behaves in a typical peer-to-peer mode where either participant sends and receives messages over the established protocol session.

3.1 Common 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.

3.1.1.1 Protocol State

3.1.1.1.1 State Diagrams

3.1.1.1.1.1 Protocol State - Sender

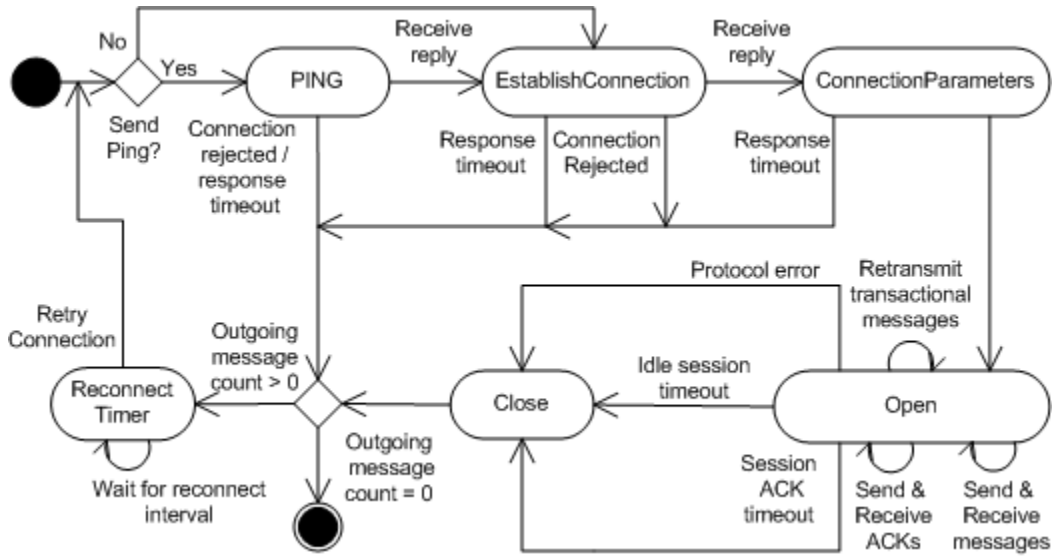


Figure 2: Sender protocol state

3.1.1.1.1.2 Protocol State - Receiver

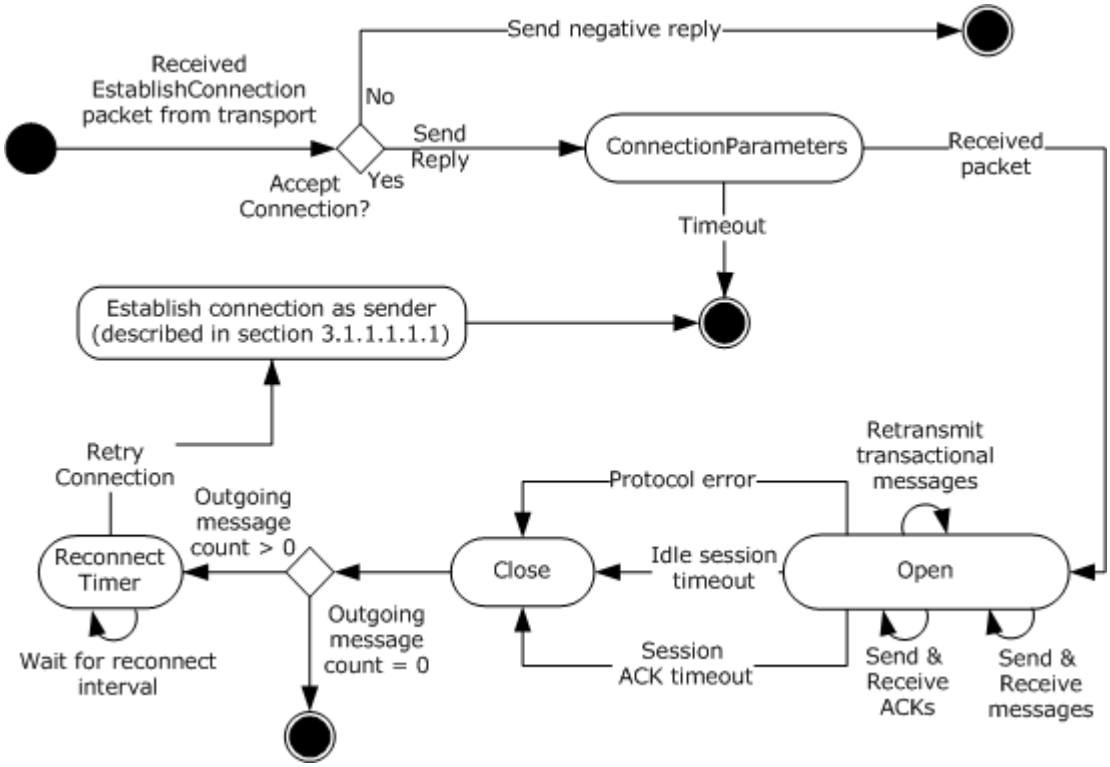


Figure 3: Receiver protocol state

3.1.1.1.1.3 Express Message State - Sender

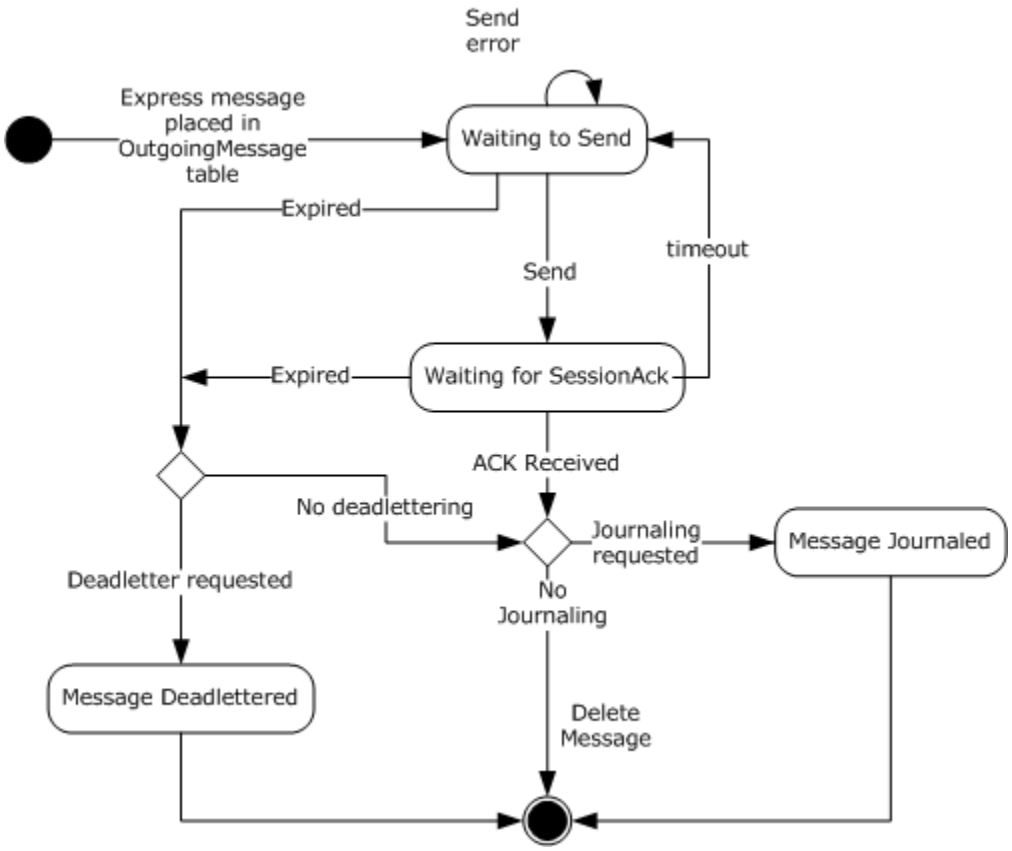


Figure 4: Sender express message state

3.1.1.1.1.4 Express Message State - Receiver

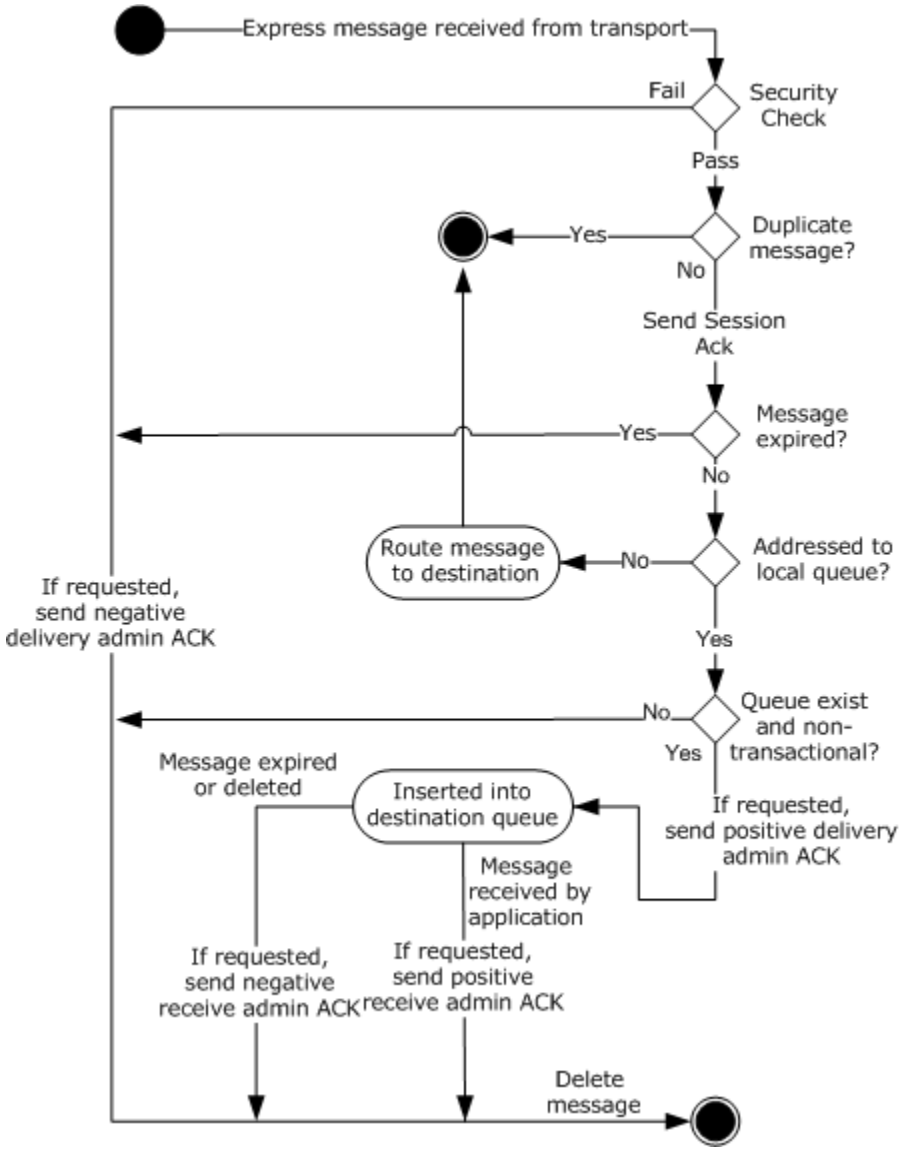


Figure 5: Receiver express message state

3.1.1.1.1.5 Recoverable Message State - Sender

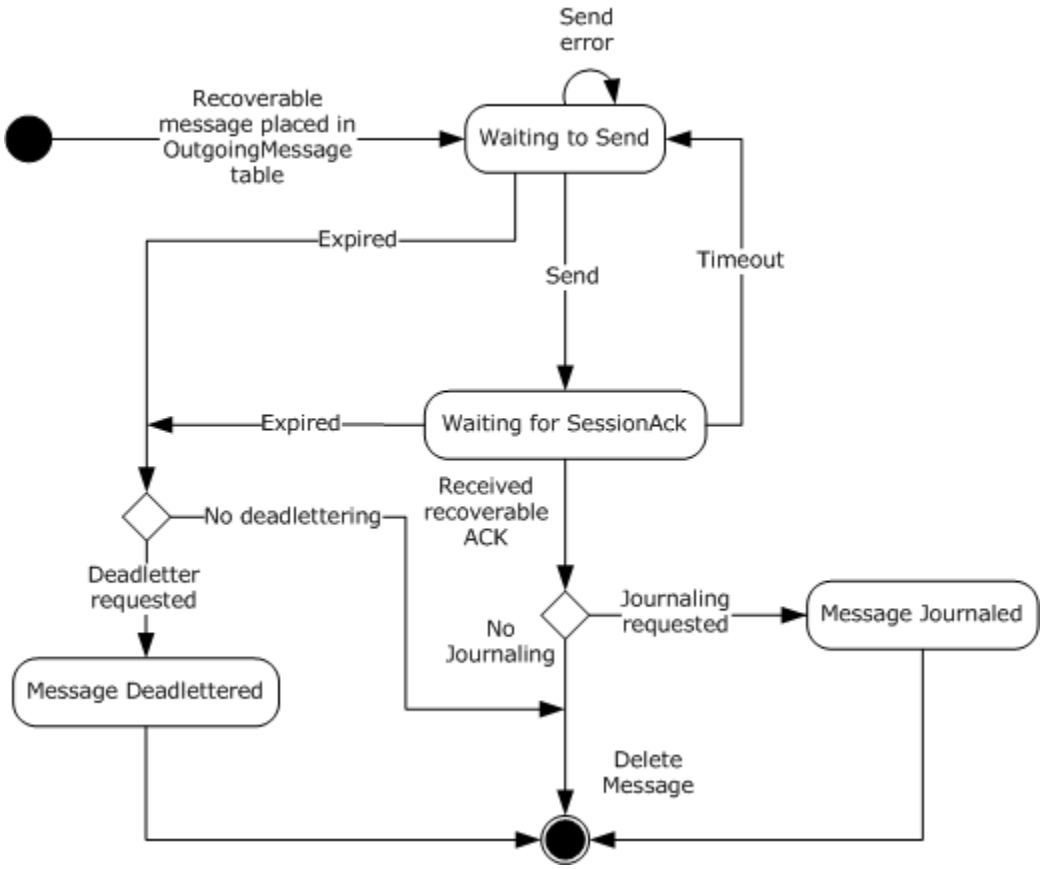


Figure 6: Sender recoverable message state

3.1.1.1.1.6 Recoverable Message State - Receiver

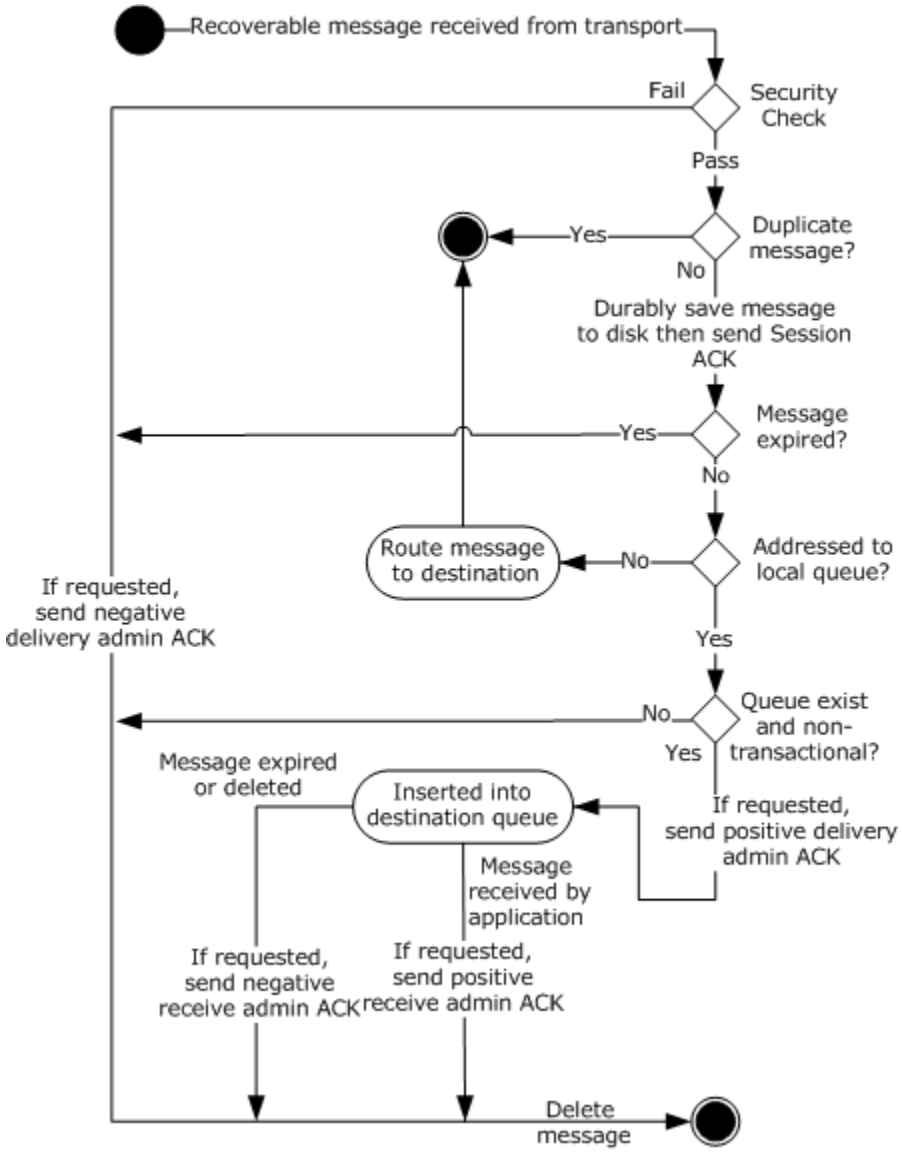


Figure 7: Receiver recoverable message state

3.1.1.1.1.7 Transactional Message State - Sender

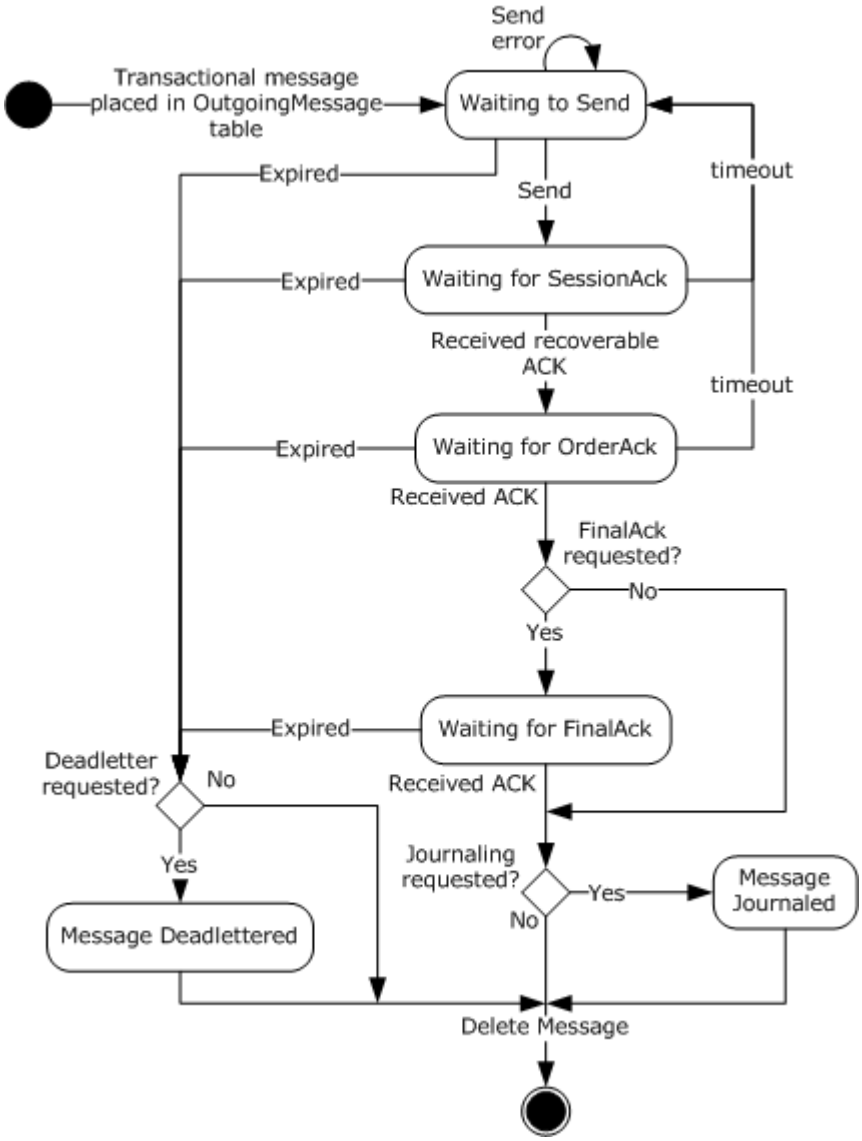


Figure 8: Sender transactional message state

3.1.1.1.1.8 Transactional Message State - Receiver

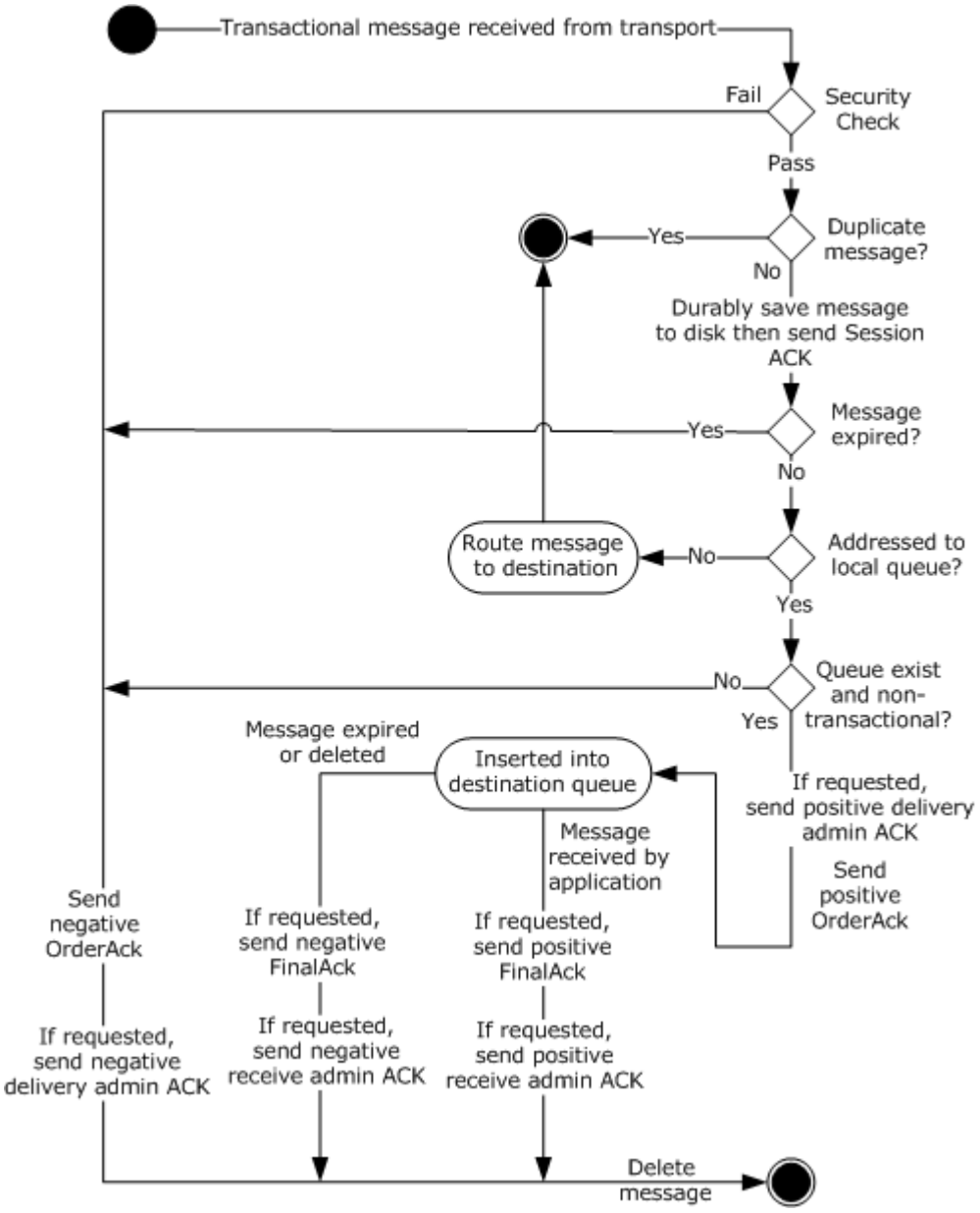


Figure 9: Receiver transactional message state

3.1.1.1.2 Global State

The protocol MUST maintain the following global data elements:

QMGuid: GUID of the local queue manager. This value uniquely identifies the local host.<34> This value MUST be saved to persistent storage.

QMPrivateKey: The private encryption key of the local queue manager.

QueueTable: A table of queues deployed at the host, keyed by name. Each entry MUST contain:

- The name of the queue. Details on queue naming conventions are as specified in [MS-MQMQ] section 2.1.
- A Boolean transaction flag indicating if the queue supports transactional messages.
- A Message List.

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

- A MessageIdentifier that uniquely identifies the message in the queue.
- A UserMessage Packet.
- The time when the message arrived to the queue.

MessageIDHistoryTable: A table that contains a history of MessageIdentifier values from UserMessage packets that have been received by the protocol host. This table provides a lightweight duplicate elimination mechanism. When a UserMessage Packet arrives, the UserMessage.UserHeader.MessageID value is checked against this table. If the value exists then the packet MUST be rejected as a duplicate. The length of history this table maintains is implementation-dependent; however, it MUST NOT contain more than 4,294,967,296 entries because that is the point at which the MessageIdOrdinal value rolls over, and values may be reused. This value SHOULD be saved to persistent storage.<35>

MessageIdOrdinal: A monotonically increasing value used in MessageIdentifier.<36> This value MUST be incremented by 1 for each UserMessage Packet sent by the protocol. This value MUST be unique only within the scope of the local queue before a rollover occurs. When a rollover occurs, values MAY<37> be reused. Rollover of this value will not affect message delivery guarantees, provided that the MessageIDHistoryTable maximum history length is not exceeded. This value MUST be saved to persistent storage.

PingCookie: An integer value that identifies Ping packet requests from this host. For more information, see section Ping Message (section 2.2.9).

3.1.1.1.3 Session State

The sender and receiver MUST independently maintain the following data elements for each session:

RemoteQMGuid: GUID (as specified in [MS-DTYP], section **GUID**) of the remote queue manager. This value represents the destination queue manager if a direct connection is possible or the next hop if routing is required. This value uniquely identifies the remote host.<38>

RemoteQMPublicKey: The public encryption key of the remote queue manager.

MessageSentCount: A count of all UserMessage Packets sent on this session. This value is incremented by 1 for each express, recoverable, and transacted UserMessage Packet sent.

RecoverableMessageSentCount: A count of recoverable UserMessage Packets sent on this session. This value is incremented by 1 for each recoverable message sent.

MessageReceivedCount: A count of all UserMessage Packets received on this session.

RecoverableMessageReceivedCount: A count of recoverable UserMessage Packets received on this session.

TxSequenceID: A TxSequenceID that identifies the current outgoing sequence of transactional messages. Only one sequence is valid at a given time. This value consists of timestamp and an ordinal as specified in TxSequenceID (section 2.2.1.2). This value MUST be saved to persistent storage.

TxSequenceNumber: The sequence number of the last outgoing transactional UserMessage Packet sent on this session. The value 0 indicates no transactional UserMessage Packets have been sent on the current sequence. This value MUST be saved to persistent storage.

IncomingTxSequenceID: A value that identifies the last incoming transactional message sequence on this session. The value null indicates no transactional sequence has existed on this session. This value MUST be saved to persistent storage.

IncomingTxSequenceNumber: A value that identifies the sequence number of the last transactional UserMessage Packet received on this session. This value MUST be saved to persistent storage.

WindowSize: This field represents the session acknowledgment window size.<39> The window size controls when the protocol sends session acknowledgments for received messages and sets a limit on the number of unacknowledged outgoing messages.

SessionState: A value that indicates the current state of the protocol. Valid values are:

Value	Meaning
NONE	Indicates that session initialization has not begun.
OPEN	The protocol has completed session initialization.
CLOSED	Indicates the session is closed.
WAITING_CP_MSG	The protocol is waiting for a ConnectionParameters Packet.
WAITING_CPR_MSG	The protocol is waiting for a ConnectionParameters Packet response packet.
WAITING_EC_MSG	The protocol is waiting for an EstablishConnection Packet.
WAITING_ECR_MSG	The protocol is waiting for an EstablishConnection Packet response packet.

SessionActive: A Boolean value that is set to TRUE when there is activity on the session. This value is used by the SessionCleanup timer to identify when there has been message activity since last timer event.

AckWaitTimeout: Time, in milliseconds, that the protocol waits before closing the session because its messages are not acknowledged.

RecoverableAckSendTimeout: Time, in milliseconds, that the protocol waits before transmitting a session acknowledgment.

RemoteQMAddress: The address of the remote queue manager. This value MUST be a NetBIOS name, a DNS name, or a textual IPv4 or IPv6 address. This value MUST be saved to persistent storage.

UnAckedMessageCount: The count of sent UserMessage Packets packets that have not been acknowledged by the remote queue manager.

OutgoingMessage table: An ordered list of MessageEntry data elements. This table contains unsent messages and/or messages awaiting acknowledgment. MessageEntry elements containing recoverable or transactional messages MUST be saved to persistent storage. This table MUST generate the "Outgoing Message Event" event as described in Other Local Events (section 3.1.7).

AwaitingFinalACK table: A list of MessageEntry data elements. This table contains a list of messages that have been successfully delivered but are awaiting final acknowledgment. Messages in this table have UserMessage Packet.UserHeader.Flag.JP set to a value other than 0x0. This value MUST be saved to persistent storage.

MessageEntry: A data structure containing the following fields:

- **Message**: A UserMessage Packet.
- **SequenceNumber**: A session sequence number.
- **RecoverableSequenceNumber**: A recoverable sequence number. A value 0 indicates that the message is not recoverable.
- **TxSequenceNumber**: A transactional sequence number. The value 0 indicates that the message is not transactional.
- **AwaitingAck**: A Boolean value indicating if the message has been sent and is awaiting session acknowledgement.
- **ReceivedSessionAck**: A Boolean value indicating that the message has received a session acknowledgment.
- **ReceivedOrderAck**: A Boolean value indicating that the message has received an order acknowledgment.
- **Transmitted**: A Boolean value indicating that the message has been sent at least once.

Note The TxSequenceID, TxSequenceNumber, IncomingTxSequenceID, and IncomingTxSequenceNumber state values only apply to transactional messages that originate from this host or are addressed to a final destination queue on this host. Transactional messages forwarded through this host are not processed as part of the incoming or outgoing transactional sequence.

The preceding conceptual data can be implemented by using a variety of techniques.

3.1.1.1.4 Persistent State Storage

Some protocol data elements MUST be saved in a persistent location that will survive process and node failure. A persistent storage requirement is indicated with a "This value MUST be saved to persistent storage" note in the element description.

3.1.1.2 Session Message Sequence

The set of UserMessage Packets sent over a session represent a message sequence. There is a local-to-remote and remote-to-local sequence. These bidirectional message sequences exist for the lifetime of the session. Both the local and remote host maintain a count of the UserMessage Packets sent and received. A message is associated with a sequence number that corresponds to its position

within the sequence. Sequence numbers MUST begin with 1 and increment by 1 with each subsequent message. For example, the third message sent on the session will have a sequence number of 3.

The protocol also maintains a count of recoverable UserMessage Packets sent and associates recoverable sequence numbers to those messages. For example, the fifth recoverable message sent on a session will have a sequence number of 5.

Transactional messages are recoverable and are included in the recoverable sequence message count.

The protocol maintains the following sequence message counts:

- **MessageSentCount:** A count of all messages sent.
- **RecoverableMessageSentCount:** A count of recoverable messages sent.
- **MessageReceivedCount:** A count of all messages received.
- **RecoverableMessageReceivedCount:** A count of recoverable messages received.

A UserMessage Packet does not contain a field that specifies its sequence number. Instead, the protocol associates sequence numbers with UserMessage Packets based on the order in which they are sent and received, respectively.

The protocol utilizes session sequence numbers when acknowledging receipt of express and recoverable messages. Sequence numbers are specified in the SessionHeader, which can appear in a stand-alone SessionAck Packet or as part of a UserMessage Packet.

3.1.1.3 Transactional Message Sequence

To provide exactly once and in order (EOIO) guarantees for transactional messages the protocol organizes transactional UserMessage Packets into transactional sequences. A transactional message sequence is independent of the session sequence. A transactional sequence is identified by a transactional sequence identifier, as specified in section 2.2.1.2, and a 32-bit transactional sequence number. The first message within a transactional sequence is assigned the value 1. Only one transactional sequence is active at a given time.

The protocol maintains the following transactional sequence state:

- **TxSequenceID:** A value that identifies the current outgoing transactional message sequence.
- **TxSequenceNumber:** The sequence number of the last transactional UserMessage Packet sent on this session.
- **IncomingTxSequenceID:** A value that identifies the last incoming transactional message sequence.
- **IncomingTxSequenceNumber:** The sequence number of the last transactional UserMessage Packet received on this session.

A transactional UserMessage Packet contains a TransactionHeader that specifies the message sequence ID, sequence number, and the sequence number of the previous message in the sequence. This information allows the remote host to determine if a message is in order and to identify duplicates.

Because messages can expire, gaps are allowed in the transactional sequence numbers. The TransactionHeader includes the previous sequence number so the remote host can determine if the received message follows the prior transactional message that was received.

When all the messages within a transactional sequence have been acknowledged, the protocol SHOULD increment the SequenceID.Ordinal by 1 and reset TxSequenceNumber to 0.<40> This process creates a new transactional sequence that MUST be used with subsequent transactional messages. Messages MUST NOT be sent on prior transactional sequences.

The receiver utilizes transactional sequence numbers when acknowledging receipt of transactional messages. Transactional sequence ID, and sequence number values are specified in the OrderAck Packet to acknowledge receipt of transactional messages.

The transactional message sequence mechanism exists alongside the session message sequence specified in section 3.1.1.2. Because transactional messages are recoverable, they are treated as recoverable messages in the session message sequence.

Transactional sequences are end-to-end. Processing of transactional sequences MUST only be done by the original sender queue manager and the final destination queue manager as defined by the queue manager identifier in the UserMessage Packet. An intermediate queue manager that receives a transactional message MUST pass the TransactionHeader to the next destination but perform no processing related to the transactional sequence.

3.1.1.4 Acknowledgments

The Message Queuing (MSMQ): Message Queuing Binary Protocol augments the underlying transport with additional levels of acknowledgment that ensure messages are reliably transferred regardless of transport connection failures, application failures, or node failures.

Message acknowledgment provides a mechanism for the receiver to notify the sender that it has received a message and, optionally, whether it has been save to disk. When the sender receives an acknowledgment, it can discard the acknowledged message or messages that it has stored locally.

The sender will retransmit unacknowledged messages if it does not receive an acknowledgment within a timeout. This protocol implements message acknowledgments at both the session sequence and transactional sequence layers.

3.1.1.4.1 Session Acknowledgment

Session acknowledgments related to the session message sequence are specified in Session Message Sequence (section 3.1.1.5).

A session acknowledgment is sent from the receiver to the sender either as a stand-alone SessionAck Packet or as a SessionHeader included inside a UserMessage Packet. The purpose of a session acknowledgment is to notify the sender that the receiver has received, and stored to disk in the case of recoverable messages, messages received from the sender.

The SessionHeader.AckSequenceNumber field specifies the total number of messages that have been received on the session. The sender SHOULD discard its local copy of express messages up to the position in the sequence specified by the receiver.<41>

The SessionHeader.RecoverableMsgAckSeqNumber and SessionHeader. RecoverableMsgAckFlags fields specify the recoverable messages that the receiver has successfully saved to disk since the last session acknowledgment. The sender SHOULD discard its local copy of the specified recoverable messages if they are not transactional.<42>

The receiver sends session acknowledgments to the sender at intervals defined by an acknowledgment timer or based on message count and session window size.

3.1.1.4.2 Transactional Acknowledgment

Transactional acknowledgments related to the Transactional Message Sequence are specified in Transactional Message Sequence (section 3.1.1.3).

Transactional acknowledgments are end-to-end acknowledgments. Processing of transactional sequences MUST only be done by the original sender queue manager and the final destination queue manager. An intermediate queue manager that receives a transactional message MUST pass the TransactionHeader to the next destination but perform no processing related to the transactional sequence.

A transactional acknowledgment is sent from the final destination to the sender in the form of an OrderAck Packet. The purpose of a transactional acknowledgment is to notify the original sender that the final destination has received and successfully stored to disk a transactional message.

An OrderAck Packet includes a TxSequenceID and TxSequenceNumber that specify the transactional message being acknowledged. The receiver MUST acknowledge transactional messages in sequence order. For example, if it has received message 1, 2, and 4 within a sequence, it cannot send an acknowledgment for message 4 until it has received, saved to disk, and acknowledged message 3.

A transactional acknowledgment MAY acknowledge multiple messages if there are multiple messages to acknowledge when the OrderAck Packet is sent.<43> For example, if the last message acknowledged was 5 and the receiver has received and saved to disk messages 6, 7, and 8, then the receiver MAY set the TxSequenceID field to the value 8.

The TxSequenceID field specifies to the sender the highest message sequence number that has been received by the receiver and saved to disk. The sender SHOULD discard its local copy of the acknowledged transactional messages up to the position in the sequence specified by the sender.<44>

Transactional acknowledgments are independent of session acknowledgments. Although transactional messages are processed by the session acknowledgment mechanism as recoverable messages, they MUST NOT be discarded by the sender as a result of a session acknowledgment. Transactional messages MUST be retained by the sender at least until the sender receives a matching transactional OrderAck Packet. If the sender requests a final acknowledgment, the sender MUST retain the message until it receives the FinalAck Packet.

The receiver MUST send transactional acknowledgments to the sender after durably storing a transactional message to disk. Transactional acknowledgments MAY be delayed for the purpose of batching acknowledgment of multiple transactional messages.

3.1.1.5 Sequence Diagrams

This section contains sequence diagrams that illustrate several common scenarios.

3.1.1.5.1 Session with Express Messages Sent

The following sequence diagram demonstrates session initialization and then the sending of express messages between two queue managers.

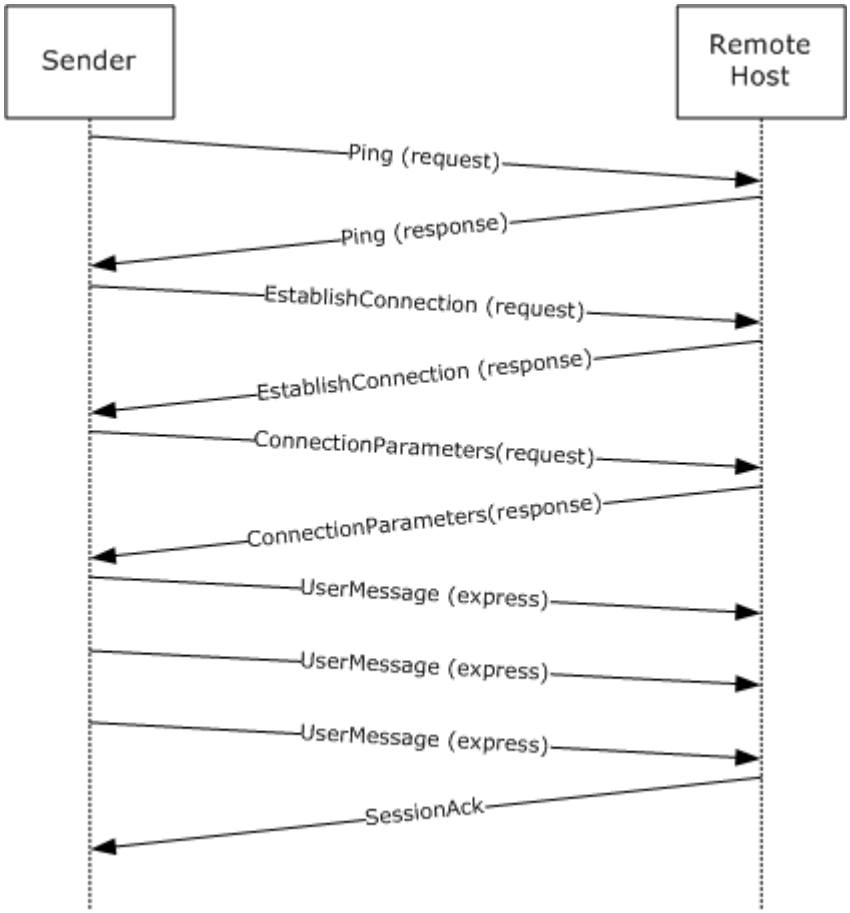


Figure 10: Sequence for express messages

The sender begins by sending a Ping Message packet on the UDP transport to the remote host to determine if it is available and can accept a connection.<45> The receiver responds with a Ping Message response packet indicating it is available.

The sender next initiates a protocol session by sending an EstablishConnection Packet to the remote host. The remote host accepts the connection and responds with an EstablishConnection Packet. The sender sends a ConnectionParameters Packet to the remote host to communicate session parameters such as timeouts and window size. The remote host confirms the session parameters by responding with a ConnectionParameters Packet response packet.

The sender sends three express UserMessage Packets to the remote host. The remote host acknowledges receipt of the UserMessage Packets by sending a SessionAck Packet. The SessionAck Packet is sent after a delay to allow batching of the session acknowledgments.

After an inactivity timeout, the session is closed by either side. A session is closed by closing the underlying TCP/SPX transport connection. The protocol does not exchange packets as part of session closure.

3.1.1.5.2 Session with Transactional Messages Sent

The following sequence diagram demonstrates session initialization and then the sending of a transactional message between two queue managers with positive source journaling enabled.

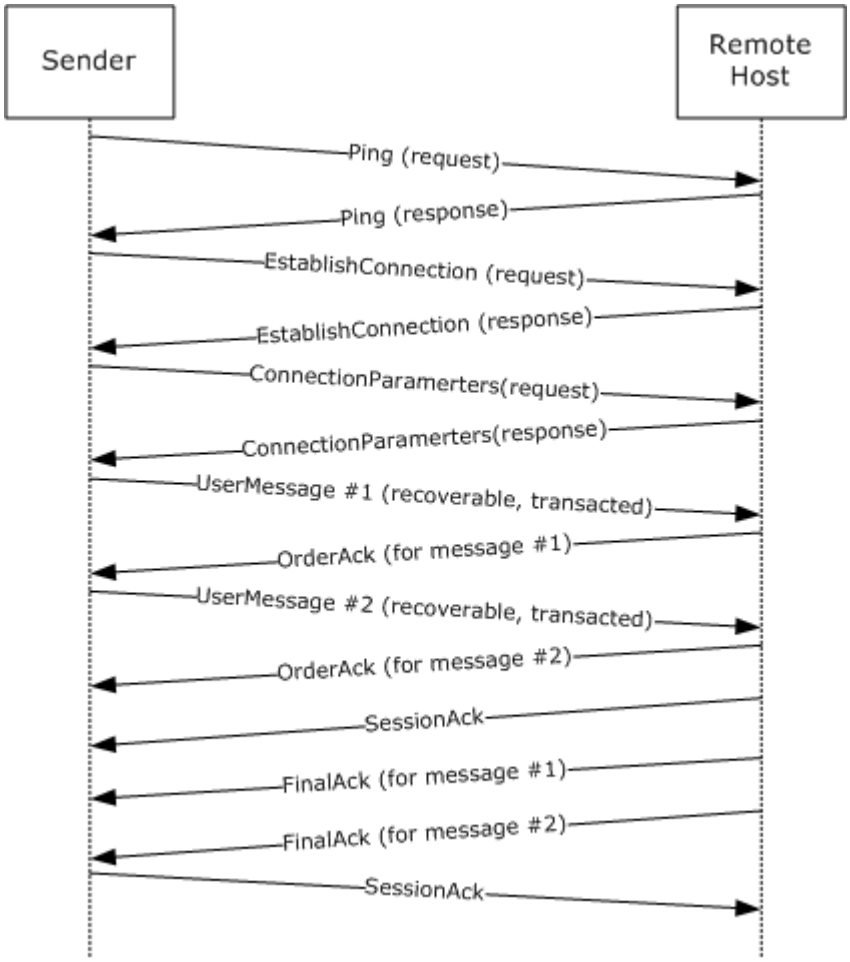


Figure 11: Sequence for transactional messages

The sender begins by sending a Ping packet on the UDP transport to the remote host to determine if it is available and can accept a connection. The receiver responds with a Ping Message response packet indicating it is available.

The sender next initiates a protocol session by sending an EstablishConnection Packet to the remote host. The remote host accepts the connection and responds with an EstablishConnection Packet. The sender sends a ConnectionParameters Packet to the remote host to communicate session parameters such as timeouts and window size. The remote host confirms the session parameters by sending a ConnectionParameters Packet response packet.

The sender sends a transactional UserMessage Packet to the remote host with positive source journaling enabled. The remote host responds by sending an OrderAck Packet. The purpose of the OrderAck Packet is to acknowledge the transactional message was received in the correct order and was not a duplicate. The sender sends another transactional UserMessage Packet and the remote host acknowledges it with an OrderAck Packet response.

The remote host sends a SessionAck Packet that contains a session acknowledgement of both UserMessage Packet packets. It is important to note that session acknowledgements and transactional acknowledgements are separate mechanisms that serve different purposes.

The remote host sends a FinalAck Packet to the sender for each of the messages. A FinalAck Packet is sent when the message is consumed from the destination queue by a higher-layer application.

After an inactivity timeout, the session is closed by either side. A session is closed by closing the underlying TCP/SPX transport connection. The protocol does not exchange packets as part of session closure.

3.1.2 Timers

The Message Queuing (MSMQ): Message Queuing Binary Protocol MUST maintain the following timers, described in the following sections:

- Session Initialization Timer
- Session Cleanup Timer
- Session Retry Connect Timer
- Session Ack Wait Timer
- Session Ack Send Timer
- Transactional Ack Wait Timer

3.1.2.1 Session Initialization Timer

This timer regulates the amount of time that the host waits for the remote host to respond to session initialization messages. This timer is started after sending a Ping Message or EstablishConnection Packet message. Session initialization must complete before this timer expires or the session is closed. The duration of this timer MUST be set based on the system configuration, which is implementation-dependent.<46>

3.1.2.2 Session Cleanup Timer

This timer regulates the amount of time the protocol waits before closing an idle protocol session. If the value of SessionActive is FALSE when the timer expires, the session is closed. SessionActive is set to TRUE when a UserMessage Packet is sent or received by the protocol. The duration of this timer MUST be set based on the system configuration, which is implementation-dependent.<47>

3.1.2.3 Session Retry Connect Timer

This timer regulates the amount of time that the protocol waits until it tries to re-establish a connection to a remote host. The duration of this timer MAY be set based on system configuration, which is implementation-dependent.<48> The protocol MAY implement an adaptive mechanism for reconnection timeouts.

3.1.2.4 Session Ack Wait Timer

This timer regulates the amount of time the protocol waits for a session acknowledgment before closing the session. Closing the session will cause the queue manager to establish a new session and retransmit the unacknowledged messages. This timer is armed after sending a UserMessage Packet. The duration of this timer MUST be set to AckWaitTimeout.

3.1.2.5 Session Ack Send Timer

This timer regulates the amount of time the protocol waits before sending a session acknowledgment to the remote host. This timer is stated after receiving a UserMessage Packet. The duration of this timer MUST be set to AckWaitTimeout / 2.

3.1.2.6 Transactional Ack Wait Timer

This timer regulates the amount of time the protocol waits for an OrderAck Packet before resending transactional messages to the remote host. This timer is started after sending a transactional UserMessage Packet. The duration of this timer SHOULD be set based on system configuration, which is implementation-dependent.<49>

3.1.3 Initialization

3.1.3.1 Global Initialization

The Session Retry Connect Timer MUST be started.

The Session Cleanup Timer MUST be started.

3.1.3.2 Session Initialization

The following values MUST be initialized for each session:

- The value of MessageSentCount MUST be set to 0x00000000.
- The value of RecoverableMessageSentCount MUST be set to 0x00000000.
- The value of MessageReceivedCount MUST be set to 0x00000000.
- The value of RecoverableMessageReceivedCount MUST be set to 0x00000000.
- The value of SessionState MUST be set to UNINITIALIZED.
- The value TxSequenceID MUST be loaded from persistent storage. If the value does not exist in persistent storage, then TxSequenceID.Ordinal MUST be set to 0x00000001, TxSequenceID.Timestamp MUST be set to an implementation-dependent value that is guaranteed to be greater than any previously generated value, and TxSequenceNumber MUST be set to 0x00000000.<50> For more information, see TxSequenceID (section 2.2.1.2).
- The value TxSequenceNumber MUST be loaded from persistent storage. If the value does not exist in persistent storage then it MUST be set to 0.
- The value MessageIDHistoryTable MUST be loaded from persistent storage. If the value does not exist in persistent storage then it MUST be set to an empty table.
- The Session Ack Wait Timer MUST be disabled.
- The Session Ack Send Timer MUST be disabled.
- The Transactional Ack Wait Timer MUST be disabled.
- The value of IncomingTxSequenceID MUST be loaded from persistent storage. If the value does not exist in persistent storage, it MUST be set to 0x0000000000000000.

- The value of IncomingTxSequenceNumber MUST be loaded from persistent storage. If the value does not exist in persistent storage, it MUST be set to 0x00000000.
- The OutgoingMessage table MUST be loaded from persistent storage if it does not exist in memory from a previous protocol instance. If the table does not exist in memory or persistent storage, it MUST be initialized to an empty table. The MessageEntry AwaitingAck, ReceivedSessionAck, ReceivedOrderAck, and Transmitted fields of each table entry MUST be set to FALSE.
- The value of MessageIdOrdinal MUST be loaded from persistent storage. If the table does not exist in persistent storage, it MUST be initialized to the value 0x00000000.
- The value UnAckedMessageCount MUST be set to the count of entries in OutgoingMessage table where ReceivedSessionAck is false.
- The value of WindowSize SHOULD be set to value 32.<51>
- The value of SessionActive MUST be set to FALSE.
- The value of AckWaitTimeout MUST be set based on the system configuration, which is implementation-dependent.<52>
- The value of RecoverableAckSendTimeout MUST be set based on system configuration, which is implementation-dependent.<53>
- The value of QMPrivateKey MUST be set based on system configuration, which is implementation dependent.<54>
- The value of RemoteQMPublicKey MUST be set to the public key of the remote queue manager.<55>
- The value of QMGuid MUST be globally unique and set based on system configuration, which is implementation-dependent.<56>
- The value of PingCookie SHOULD be set to the value 0x00000000. <57>
- The value of WaitingForEC MUST be set to TRUE.
- The value of WaitingForCP MUST be set to FALSE.

3.1.4 Higher-Layer Triggered Events

The operation of the Message Queuing (MSMQ): Message Queuing Binary Protocol is initiated and subsequently driven by the following higher-layer triggered events:

- Queue Manager Service Started
- Queue Manager Service Stopped
- Queue Manager Inserts Message into OutgoingMessage Table
- The Administrator Purges a Queue

3.1.4.1 Queue Manager Service Started

The queue manager service on startup MUST restore the protocol persistent state. The protocol MUST create a session to the remote queue manager if there are messages in the OutgoingMessage table. Protocol session creation is specified in Create a Protocol Session (section 3.1.5.2).

3.1.4.2 Queue Manager Service Stopped

When the queue manager service is stopped the protocol MUST be closed as specified in Closing a Session (section 3.1.5.10).

3.1.4.3 Queue Manager Inserts Message into OutgoingMessage Table

This event occurs when the queue manager inserts a MessageEntry into the OutgoingMessage table. The MessageEntry MUST have the following values:

- The Message field set to a UserMessage Packet.
- The AwaitingAck, ReceivedSessionAck, ReceivedOrderAck, and Transmitted fields set to FALSE.
- The SequenceNumber field set to 0x00000000.
- The RecoverableSequenceNumber field set to 0x00000000.

The queue manager MUST construct the UserMessage Packet by using the steps described in UserMessage Packet (section 2.2.8).

For more information on how the message is processed, see Outgoing Message Event (section 3.1.7.1).

3.1.4.4 Administrator Purges a Queue

This event occurs when the administrator purges the messages from a queue hosted by the queue manager. All messages MUST be removed from a purged queue. Each removed message MUST be processed as specified in Message Removed from Destination Queue (section 3.1.7.2).

3.1.5 Message Processing Events and Sequencing Rules

3.1.5.1 Receiving Any Packet

Unless specifically noted in a subsequent section, the following actions MUST be applied to any session message received:

- Identifying Packet Type
- Verifying the Signature
- Handling Incorrectly Formatted Messages

3.1.5.1.1 Identifying Packet Type

A packet is identified by inspecting the BaseHeader and possibly subsequent packet headers. The following list describes how to identify each packet type.

EstablishConnection Packet**:** The BaseHeader.Flags.IN field MUST be set and the InternalHeader.Flags.PT field MUST be set to 0x2.

ConnectionParameters Packet**:** The BaseHeader.Flags.IN field MUST be set and the InternalHeader.Flags.PT field MUST be set to 0x3.

SessionAck Packet**:** The BaseHeader.Flags.IN and BaseHeader.Flags.SH fields MUST be set and the InternalHeader.Flags.PT field MUST be set to 0x1.

OrderAck Packet: The UserMessage Packet.DestinationQueue field MUST address the message to the local private queue named "order_queue". All bits in the BaseHeader.Flags field MUST be set to 0x0. MessageData.Label MUST be set to "QM Ordering Ack". The UserMessage.MessagePropertiesHeader.MessageSize field MUST be set to 0x00000024. The UserMessage.MessagePropertiesHeader.MessageClass field MUST NOT be set to MQMSG_CLASS_NORMAL.

FinalAck Packet: The UserMessage.DestinationQueue field MUST address the local private queue named "order_queue". All bits in the BaseHeader.Flags field MUST be set to 0x0. MessageData.Label MUST be set to "Final Ordering Ack". The UserMessage. MessagePropertiesHeader.MessageClass field MUST NOT be set to MQMSG_CLASS_NORMAL.

UserMessage Packet: BaseHeader.Flags.IN MUST be set and UserMessage. MessagePropertiesHeader.MessageClass field MUST be set to MQMSG_CLASS_NORMAL.

Ping Packet: The Ping Message packet is the only packet type sent over the UDP transport.

3.1.5.1.2 Verifying the Signature

A packet signature is validated by evaluating the BaseHeader.VersionNumber and BaseHeader.Signature fields. A packet is valid when the BaseHeader.VersionNumber field is set to 0x10 and BaseHeader.Signature is set to 0x524F494C (big-endian order). Any other value indicates an invalid packet.

If signature validation fails, the protocol MUST discard the received packet and perform no further processing for it and then close the session as specified in Closing a Session (section 3.1.5.10). If signature verification succeeds, the protocol continues processing on the packet as specified in subsequent sections.

A Ping Message packet is valid if the Ping.Signature field has the value 0x5548. An invalid packet MUST be ignored.

3.1.5.1.3 Handling Incorrectly Formatted Messages

If the protocol receives a request that does not conform to the structures outlined in Messages (section 2), the protocol MUST discard the received packet and perform no further processing for it and then close the session as specified in Closing a Session (section 3.1.5.10).

3.1.5.2 Create a Protocol Session

The queue manager creates a session to a remote queue manager. This could be the result of the queue manager receiving a message from a higher-layer application or the queue manager retrying a connection to a remote queue manager.

Creating a session to a remote host consists of the following sequence of operations:

- Resolve Host Address
- Send Ping Packet
- Session Initialization

3.1.5.2.1 Resolve Host Address

The queue manager MUST provide a queue format name that specifies the destination queue manager and queue.

If a **direct format name** is specified, the protocol MUST parse the queue format name and identify the host component. The host component MUST be resolved to a **NetBIOS** name, a DNS name, or a textual IPv4 or IPv6 address. The RemoteQMGuid state variable MUST be set to all zero bytes.

If a private format name or public format name is specified, the protocol MUST perform the following steps to compute the destination host:

- Parse the queue format name and resolve the host component to the associated destination queue manager GUID.
- Use the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to determine where to send the message. The queue manager GUID computed by the algorithm represents the destination queue manager, if a direct connection is possible, or the next hop, if routing is required. The algorithm takes as input the value of the QMGuid state variable and the destination queue manager GUID that was computed in the previous step.
- The RemoteQMGuid state value MUST be set to the queue manager GUID computed by the Binary Reliable Message Routing Algorithm.
- The queue manager GUID computed by the Binary Reliable Message Routing Algorithm MUST be resolved to a NetBIOS name, a DNS name, or a textual IPv4 or IPv6 address.

For information on the structure of a queue format name, see [MS-MQMQ] section 2.1. For information on locating host addresses and attributes such as queue manager GUID, see Directory Service as specified in [MS-MQMA], Directory Service Schema as specified in [MS-MQMQ], and Directory Service Protocol as specified in [MS-MQDS]. The host associated with a public queue name can be queried using the directory service.<58>

3.1.5.2.2 Send Ping Packet

The protocol MAY send a Ping Message packet to the remote host before attempting to establish a session.<59> If a Ping Message packet is not sent, skip to Session Initialization (section 3.1.5.2.3).

If a Ping Message packet is sent, the following steps MUST be performed:

- The protocol MUST start the Session Initialization Timer.
- The Ping.QMGuid field MUST be set to QMGuid.
- The value of PingCookie MUST be incremented by 1.
- The Ping.Cookie value MUST be set to PingCookie.
- A Ping Message packet MUST be sent to the remote host using the transport settings specified in Ping Request (section 2.1.2).

See Receiving a Ping Packet (section 3.1.5.9) for the next step in session initialization.

3.1.5.2.3 Session Initialization

The protocol MUST restart the Session Initialization Timer.

An EstablishConnection Packet MUST be sent to the remote host using the transport settings specified in Ping Request (section 2.1.2). The following fields MUST be set in the EstablishConnection Packet:

- BaseHeader.Flags.SH MUST be set.

- InternalHeader.Flags.PT field MUST be set to 0x2.
- InternalHeader.Flags.CS MUST be set to 0x0.
- EstablishConnectionHeader.ClientGuid MUST be set to QMGuid.
- EstablishConnectionHeader.ServerGuid MUST be set to RemoteQMGuid.

The SessionState value must be set to WAITING_ECR_MSG.

See Receiving an EstablishConnection Packet (section 3.1.5.3) for the next step in session initialization.

3.1.5.3 Receiving an EstablishConnection Packet

If SessionState is not set to WAITING_EC_MSG or WAITING_ECR_MSG, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

An EstablishConnection Packet is a connection request from a remote host or a response to an EstablishConnection Packet sent from this protocol as specified in Session Initialization (section 3.1.5.2.3).

3.1.5.3.1 Request Packet

The packet is processed as a connection request if SessionState is WAITING_EC_MSG.

The packet is valid if EstablishConnectionHeader.ServerGuid is equal to QMGuid or EstablishConnectionHeader.ServerGuid is 0. If the packet is invalid, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

The value of RemoteQMGuid MUST be set to EstablishConnectionHeader.ClientGuid.

The protocol MUST reply to a connection request by sending an EstablishConnection Packet response with the following values:

- BaseHeader.Flags.IN MUST be set.
- InternalHeader.Flags.PT MUST be set to 0x2.
- InternalHeader.Flags.CS MUST be set to 0x0.
- EstablishConnectionHeader.ClientGuid MUST be set to the EstablishConnectionHeader.ClientGuid value in the original packet.
- EstablishConnectionHeader.ServerGuid MUST be set to QMGuid.

The SessionState value MUST be set to WAITING_CP_MSG.

3.1.5.3.2 Response Packet

The packet is processed as a connection response if SessionState is WAITING_ECR_MSG.

The packet is valid if EstablishConnectionHeader.ClientGuid is equal to QMGuid and EstablishConnectionHeader.ServerGUID is equal to RemoteQMGuid and InternalHeader.Flags.CS MUST be set to 0x0. If the packet is invalid, the protocol MUST discard the packet and close the session as specified in Closing a Session (section 3.1.5.10).

The protocol MUST reply to a connection response by sending a ConnectionParameters Packet with the following values:

- BaseHeader.Flags.IN MUST be set.
- InternalHeader.Flags.PT MUST be set to 0x3.
- InternalHeader.Flags.CS MUST be set to 0x0.
- ConnectionParametersHeader.RecoverableAckTimeout field MUST be set to RecoverableAckSendTimeout.
- ConnectionParametersHeader.AckTimeout field MUST be set to AckWaitTimeout.
- ConnectionParametersHeader.WindowSize field MUST be set to WindowSize.

The SessionState value MUST be set to WAITING_CPR_MSG.

See Receiving a ConnectionParameters Packet (section 3.1.5.4) for the next step in session initialization.

3.1.5.4 Receiving a ConnectionParameters Packet

If SessionState is not set to WAITING_CP_MSG or WAITING_CPR_MSG, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

A ConnectionParameters Packet is either a request from a remote host or a response to a ConnectionParameters Packet sent from this protocol as specified in Response Packet (section 3.1.5.3.2).

3.1.5.4.1 Request Packet

The packet is processed as a request if SessionState is WAITING_CP_MSG.

- WindowSize MUST be set to the value ConnectionParametersHeader.WindowSize.
- AckWaitTimeout MUST be set to the value ConnectionParametersHeader.AckTimeout.
- The duration of the Session Ack Wait timer MUST be set to AckWaitTimeout.
- The duration of the Session Ack Send Timer MUST be set to AckWaitTimeout / 2.
- RecoverableAckSendTimeout MUST be set to the value ConnectionParametersHeader.RecoverableAckTimeout.

The protocol MUST reply to a connection request by sending a ConnectionParameters Packet response with the following values:

- BaseHeader.Flags.IN MUST be set.
- InternalHeader.Flags.PT MUST be set to 0x3.
- InternalHeader.Flags.CS MUST be set to 0x0.
- ConnectionParametersHeader.RecoverableAckTimeout field MUST be set to the value of ConnectionParametersHeader.RecoverableAckTimeout in the received packet.

- ConnectionParametersHeader.AckTimeout field MUST be set to the value of ConnectionParametersHeader.AckTimeout in the received packet.
- ConnectionParametersHeader.WindowSize field SHOULD be set to the value of ConnectionParametersHeader.WindowSize in the received packet.<60>

The Session Initialization timer MUST be stopped. The value of SessionState MUST be set to OPEN. The session is now initialized and is ready to send or receive UserMessage Packets.

3.1.5.4.2 Response Packet

The packet is processed as a response if SessionState is WAITING_CPR_MSG.

WindowSize MUST be set to the value ConnectionParametersHeader.WindowSize. The Session Initialization Timer MUST be stopped. The value of SessionState MUST be set to OPEN. The session is now initialized and is ready to send or receive UserMessage Packets.

3.1.5.5 Receiving a SessionAck Packet

If SessionState is not set to the value OPEN, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

A SessionHeader contains a session acknowledgment that acknowledges express and recoverable messages. A SessionHeader can appear in a SessionAck Packet or be piggy-backed onto a UserMessage Packet. A SessionHeader is present in the packet when BaseHeader.Flags.SH is set.

The protocol MUST perform the following steps to process a SessionHeader:

- Mark Acknowledged Messages
- Delete Acknowledged Express Messages
- Delete Acknowledged Recoverable Messages
- Source Journaling
- Validate Message Counts

3.1.5.5.1 Mark Acknowledged Messages

The protocol MUST set the MessageEntry.ReceivedSessionAck field to TRUE for all elements in the OutgoingMessage table where MessageEntry.SequenceNumber is less than or equal to SessionHeader.AckSequenceNumber.

UnAckedMessageCount MUST be decremented by the number of elements where MessageEntry.ReceivedSessionAck field was set to TRUE by the above operation.

3.1.5.5.2 Delete Acknowledged Express Messages

The protocol MUST delete all elements from the OutgoingMessage table where MessageEntry.SequenceNumber is less than or equal to SessionHeader.AckSequenceNumber and MessageEntry.RecoverableSequenceNumber is set to 0x00000000.

3.1.5.5.3 Delete Acknowledged Recoverable Messages

The protocol MUST delete all elements from the OutgoingMessage table where MessageEntry.RecoverableSequenceNumber is less than or equal to SessionHeader.RecoverableMsgAckSeqNumber and MessageEntry.TxSequenceNumber is set to 0x00000000.

The protocol MUST delete all elements from the OutgoingMessage table where MessageEntry.RecoverableSequenceNumber is represented in the SessionHeader.RecoverableMsgAckFlags field with a bit set to 0x1 and MessageEntry.TxSequenceNumber is set to 0x00000000. Details of the SessionHeader.RecoverableMsgAckFlags bit representation are as specified in SessionHeader (section 2.2.8.4).

The protocol MUST delete all elements from the OutgoingMessage table where MessageEntry.RecoverableSequenceNumber is represented in SessionHeader. RecoverableMsgAckFlags with a bit set to 0x1 and MessageTable.ReceivedOrderAck is TRUE. Details of the SessionHeader.RecoverableMsgAckFlags bit representation are as specified in SessionHeader (section 2.2.8.4).

3.1.5.5.4 Source Journaling

An acknowledged message that is deleted from the OutgoingMessage table with UserMessage.UserHeader.Flags.JP set SHOULD be logged locally.<61>

3.1.5.5.5 Validate Message Counts

If SessionHeader.UserMsgSequenceNumber is not equal to MessageReceivedCount or SessionHeader.RecoverableMsgSeqNumber is not equal to RecoverableMessageReceivedCount, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

If the OutgoingMessage table contains any message where AwaitingAck is true, the Session Ack Wait timer MUST be restarted.

Each transactional message, which is recoverable, MUST be retained until receipt of the corresponding OrderAck Packet.

3.1.5.6 Receiving an OrderAck Packet

If SessionState is not set to the value OPEN, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

The protocol MUST delete all elements from the OutgoingMessage table where MessageEntry.TxSequenceNumber is greater than 0x00000000 and is less than or equal to OrderAck.MessagePropertiesHeader.MessageData.TxSequenceNumber and MessageTable.ReceivedSessionAck is set to TRUE.

A message that is deleted from the OutgoingMessage table with UserMessage.UserHeader.Flags.JP set SHOULD be copied to the AwaitingFinalAck table if OrderAck.MessagePropertyHeader.MessageClass is equal to MQMSG_CLASS_ORDER_ACK.<62>

A message that is deleted from the OutgoingMessage table with UserMessage.UserHeader.Flags.JP set SHOULD be logged locally if OrderAck.MessagePropertyHeader.MessageClass is not equal to MQMSG_CLASS_ORDER_ACK.<63>

The dead-letter queue is a system-generated queue and is implementation-dependent.<64>

If the OutgoingMessage table contains no MessageEntry elements where MessageEntry.TxSequenceNumber is nonzero, the protocol SHOULD increment TxSequenceID.Ordinal by 1 and set TxSequenceNumber to the value 0x00000000.<65>

3.1.5.7 Receiving a FinalAck Packet

If SessionState is not set to the value OPEN, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

A FinalAck Packet indicates that the message represented by FinalAck.MessagePropertiesHeader.MessageData.MessageID has been removed from the destination queue.

The message where UserMessage.UserHeader.MessageID is equal to FinalAck.MessagePropertiesHeader.MessageData.MessageID in AwaitingFinalAck table MUST be moved to the system journal queue if FinalAck.MessagePropertyHeader.MessageClass is equal to MQMSG_CLASS_ORDER_ACK.

The message where UserMessage.UserHeader.MessageID is equal to FinalAck.MessagePropertiesHeader.MessageData.MessageID in AwaitingFinalAck table MUST be moved to the system transactional dead-letter queue if FinalAck.MessagePropertyHeader.MessageClass is not equal to MQMSG_CLASS_ORDER_ACK.

The journal and dead-letter queues are system-generated and are implementation-dependent.<66>

3.1.5.8 Receiving a UserMessage Packet

A UserMessage Packet contains an application-defined message sent from the remote host. A received message can be addressed to a local queue or to a queue on a remote host.

If SessionState is not set to the value OPEN, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

Processing a UserMessage Packet consists of the following sequence of operations. The protocol MUST perform the following steps to process a UserMessage Packet:

- Duplicate Detection
- General Processing
- Security
- SessionHeader Processing
- Message Expiration
- Transactional Message Processing
- Recoverable Message Processing
- Inserting a Message into a Local Queue
- Sending a Trace Message
- Sending Administration Acknowledgments

3.1.5.8.1 Duplicate Detection

If the value UserMessage.UserHeader.MessageID exists in the MessageIDHistoryTable table the protocol MUST discard this message and perform no further processing.

3.1.5.8.2 General Processing

If the value of UserMesage.UserHeader.QueueManagerAddress is not equal to QMGuid and is not filled with the value 0x00 and the protocol is unable to save the message locally (for example, insufficient disk space), then the protocol MUST disregard the message and perform no further processing.

The value of MessageReceivedCount MUST be incremented by 1.

If UserMessage.UserHeader.Flags.DM is set to 0x1, the value of RecoverableMessageReceivedCount MUST be incremented by 1.

A MessageIdentifier consisting of UserMessage.UserHeader.MessageID and UserMessage.UserHeader.SourceQueueManager MUST be inserted into the MessageIDHistoryTable.

The Session Ack Send timer MUST be started if it is in a stopped state.

If the value of UserMesage.UserHeader.QueueManagerAddress is equal to QMGuid, the protocol MUST perform the following logic:

- The protocol MUST disregard a message if it is addressed to a nonexistent queue. If the UserMessage.UserHeader.DestinationQueue field does not correspond to a queue in QueueTable, the protocol MUST disregard the message and perform no further processing.
- If a message is rejected for the reason above and administration acknowledgments are enabled, the protocol MUST send a **MQMSG_CLASS_NACK_BAD_DST_Q** negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).
- The protocol MUST disregard a message and perform no further processing if it is unable to store the message because the destination queue is full.
- If a message is rejected for the reason above and administration acknowledgments are enabled, the protocol MUST send an **MQMSG_CLASS_NACK_Q_EXCEED_QUOTA** negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).
- A transactional message can only be delivered to a transactional queue. If the UserMessage Packet contains a TransactionHeader and UserMessage.UserHeader.DestinationQueue corresponds to a non-transactional queue in QueueTable, the protocol MUST disregard the message and perform no further processing.
- If a message is rejected for the reason above and administration acknowledgments are enabled, the protocol MUST send a **MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q** negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).
- A non-transactional message can only be delivered to a non-transactional queue. If the UserMessage Packet contains a TransactionHeader and UserMessage.UserHeader.DestinationQueue corresponds to a transactional queue in QueueTable, the protocol MUST disregard the message and perform no further processing.

- If a message is rejected for the reason above and administration acknowledgments are enabled, the protocol MUST send a **MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG** negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

- If the packet is rejected for any reason other than being a duplicate and contains a TransactionHeader and UserMessage.UserHeader.QueueManagerAddress is equal to QMGuid, the protocol MUST send a negative OrderAck Packet. Details on sending an OrderAck Packet are as specified in Transactional Message Processing (section 3.1.5.8.6).

If the value of UserMessage.UserHeader.QueueManagerAddress is not equal to QMGuid, the protocol MUST perform the following logic:

- The protocol MUST increment the value of UserHeader.Flags.RC by 1. If the incremented value exceeds the valid range of the field, then the protocol MUST disregard the message and perform no further processing.

- If a message is rejected for the reason above, and administration acknowledgments are enabled, the protocol MUST send an **MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED** negative acknowledgment to the sender, as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

- If a message is rejected for the reason above, and contains a TransactionHeader, the protocol MUST send a negative OrderAck Packet. Details on sending an OrderAck Packet are as specified in Transactional Message Processing (section 3.1.5.8.6).

3.1.5.8.3 Security

Authentication and encryption are not supported when a queue is addressed using a direct format name. If the value UserMesage.UserHeader.QueueManagerAddress is equal to QMGuid and a SecurityHeader is present in the UserMessage Packet, the following logic MUST be applied to the message:

If SecurityHeader.Flags.AU is set, the packet MUST be authenticated. The protocol MUST verify the signature by re-computing the packet hash and comparing it to the decrypted signature value. The sender MUST compute a hash of the fields specified by the SecurityHeader.Flags.AS field by using the hash algorithm specified by the MessagePropertiesHeader.HashAlgorithm field. This value MUST be compared to the SecurityHeader.Signature field value after it has been decrypted using RSA and the public key contained in the SecurityData.SenderCert certificate. If the two values do not match, the protocol MUST disregard the message and perform no further processing.

If a message is rejected for a bad signature and administration acknowledgments are enabled, the protocol MUST send a MQMSG_CLASS_NACK_BAD_SIGNATURE negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

The protocol MUST decrypt the SecurityData.EncryptionKey field using RSA and the local queue manager private key in QMPrivateKey. Using the decrypted key, the receiver MUST decrypt the MessagePropertiesHeader.MessageData.MessageBody field using the encryption algorithm specified by MessagePropertiesHeader.EncryptionAlgorithm and the decrypted SecurityData.EncryptionKey value.

The protocol SHOULD perform an access check to authorize the Security Identifier (SID) specified in SecurityHeader.SecurityData.SecurityID against the queue specified by UserMessage.UserHeader.DestinationQueue.<67> If the security check fails, the protocol MUST disregard the message and perform no further processing.

If a message is rejected and administration acknowledgments are enabled, the protocol MUST send a MQMSG_CLASS_NACK_ACCESS_DENIED negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

3.1.5.8.4 SessionHeader Processing

The remote host can include a SessionHeader in the UserMessage Packet message. A SessionHeader does not contain information about the UserMessage Packet message but instead contains session acknowledgment information for express and recoverable messages.

If a UserMessage Packet contains a SessionHeader, it MUST be processed as specified in Receiving a SessionAck Packet (section 3.1.5.5).

3.1.5.8.5 Message Expiration

The values of UserMessage.UserHeader.TimeToBeReceived and UserMessage.BaseHeader.TimeToReachQueue control the message lifetime. The protocol MUST check a received message for expiration before processing.

CURRENT_TIME represents the current system time. This value is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time) according to the system clock.

If CURRENT_TIME - UserMessage.UserHeader.SentTime is greater than UserMessage.UserHeader.TimeToBeReceived or UserMessage.BaseHeader.TimeToReachQueue, the message has expired. The protocol MUST disregard the message and perform no further processing.

If a message has expired and administration acknowledgments are enabled, the protocol MUST send a MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT negative acknowledgment to the sender as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

If a message has expired that contains a UserMessage.UserHeader.Flags.JN field that is set, the message SHOULD be logged locally.<68>

3.1.5.8.6 Transactional Message Processing

If the UserMessage Packet contains a TransactionHeader and the value UserMesage.UserHeader.QueueManagerAddress is equal to QMGuid, the following logic must be applied:

Transactional messages are only accepted in order and exactly once. The following conditions define when a message is accepted.

If the UserMessage Packet has not been rejected for any reason above, the message MUST be accepted if UserMessage.TransactionHeader.TxSequenceID is equal to IncomingTxSequenceID and UserMessage.TransactionHeader.TxSequenceNumber is greater than IncomingTxSequenceNumber and UserMessage.TransactionHeader.PreviousTxSequenceNumber is less than or equal to TransactionHeader.PreviousTxSequenceNumber.

A message MUST be accepted if UserMessage.TransactionHeader.TxSequenceID.Ordinal is greater than IncomingTxSequenceID.Ordinal and UserMessage.TransactionHeader.PreviousTxSequenceNumber is equal to 0x00000000.

An OrderAck Packet MUST be sent to the remote host to acknowledge the message. The protocol MAY delay sending the OrderAck Packet to provide batching of success acknowledgments for multiple transactional messages.<69> The OrderAck.TxSequenceID field MUST be set to

UserMessage.TransactionHeader.TxSequenceID, OrderAck.TxSequenceNumber MUST be set to UserMessage.TransactionHeader.TxSequenceNumber, OrderAck.TxPreviousSequenceNumber MUST be set to IncomingTxSequenceNumber, and OrderAck.MessageID MUST be set to UserMessage.UserHeader.MessageID.

The protocol MUST set IncomingTxSequenceID to UserMessage.TransactionHeader.TxSequenceID and set IncomingTxSequenceID to UserMessage.TransactionHeader.TxSequenceID.

The OrderAck Packet MUST be addressed to the sender order queue by setting the UserMessage.DestinationQueue field to "TCP:*address*\ PRIVATE$\order_queue" or "SPX:*address*\ PRIVATE$\order_queue" where *address* is the UserMessage.UserHeader.SourceQueueManager field value.

If the message is successfully processed then UserMessage.MessagePropertiesHeader.MessageClass MUST be set to MQMSG_CLASS_ORDER_ACK. In the event of a failure, an OrderAck Packet MUST return one of the following negative acknowledgments:

- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q
- MQMSG_CLASS_NACK_Q_DELETED
- MQMSG_CLASS_NACK_Q_EXCEED_QUOTA
- MQMSG_CLASS_NACK_ACCESS_DENIED
- MQMSG_CLASS_NACK_BAD_ENCRYPTION
- MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER
- MQMSG_CLASS_NACK_BAD_SIGNATURE

Details about the preceding negative acknowledgments are as specified in Sending Administration Acknowledgments (section 3.1.5.8.10).

3.1.5.8.7 Recoverable Message Processing

If UserMessage.UserHeader.Flags.DM is set to 0x1, the protocol MUST save the message to disk and then send a SessionAck Packet to acknowledge the message has been durably stored. The protocol MUST wait an amount of time equal to or greater than RecoverableAckSendTimeout before sending the SessionAck Packet. This delay allows acknowledgments from other messages to potentially be batched together.

If this recoverable message is acknowledged by a SessionAck Packet that is generated by another protocol event before the RecoverableAckSendTimeout expires, the SessionAck Packet is not required.

The protocol MUST send a SessionAck Packet as specified in Session Ack Send Timer Event (section 3.1.6.4).

3.1.5.8.8 Inserting a Message into a Local Queue

If the value UserMesage.UserHeader.QueueManagerAddress is equal to QMGuid, or filled with the value 0x00, then the message MUST be inserted into the queue in QueueTable that corresponds to the queue address specified in the UserMessage.UserHeader.DestinationQueue field.

If the value UserMesage.UserHeader.QueueManagerAddress is not equal to QMGuid and is not filled with the value 0x00, the message MUST be inserted into an outgoing queue where it will be

processed by the queue manager and routed to a host closer to the destination. This protocol utilizes the Binary Reliable Message Routing Algorithm specified in [MS-MQBR] to compute the route to the destination queue manager.

3.1.5.8.9 Sending a Trace Message

If the BaseHeader.Flags.TR field is set, the protocol MUST send a report message to the queue specified by the DebugHeader.QueueIdentifier field. Report messages are utilized by application logic to track the delivery of sent messages.

To send a report message, the protocol MUST send a UserMessage Packet with the following field values:

The MessagePropertiesHeader.MessageClass field MUST be set to MQMSG_CLASS_REPORT, the UserMessage.DestinationQueue set to DebugHeader.QueueIdentifier, and UserHeader.Flags.DM set to 0x0.

The MessagePropertiesHeader.MessageData.Label MUST be set to a Unicode string in the format specified by the following ABNF rules:

```
label = qm-id %x3A message-id %x3A hops SP "received by" SP computer
        SP "at"  SP time-date %x0000
qm-id = 4HEXDIG ; MUST be set to the first four hexadecimal digits
                ; of the source queue identifier
message-id = 8HEXDIG ; hexadecimal form of the UserHeader.MessageID
                     ; field
hops = 2HEXDIG ; MUST be set to the UserHeader.Flags.RC field
computer = GUID ; MUST be set to UserHeader.SourceQueueManager field
time-date = hour SP ("AM" / "PM") SP date
hour  =  2DIGIT ":" 2DIGIT [":" 2DIGIT] ; ANSI and Military
date =  day "," month SP 2DIGIT SP year; day, month day year
month  = "Jan" / "Feb" / "Mar" / "Apr" / "May" / "Jun"
         / "Jul" / "Aug" / "Sep" / "Oct" / "Nov" / "Dec"
day = "Mon" / "Tue" / "Wed" / "Thu" / "Fri" / "Sat" / "Sun"
year = 2DIGIT
GUID =  8HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 12HEXDIG
        ; A GUID the form XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
        ; Where each X is a Hex digit
```

The MessagePropertiesHeader.MessageData.MessageBody MUST be set to a Unicode string in the format specified by the following ABNF rules:

```
Report = "<MESSAGE ID>" id "</MESSAGE ID>" CR LF
         "<TARGET QUEUE>" queue "</TARGET QUEUE>" CR LF
id = 8HEXDIG ; MUST be set to UserHeader.MessageID field
queue = queue-format; MUST be set to UserHeader.DestinationQueue field
```

The ABNF rule queue-format is as specified in [MS-MQMQ] section 2.1.

3.1.5.8.10 Sending Administration Acknowledgments

Administration acknowledgment messages are system-generated express UserMessage Packets that are sent to administration queues specified in the packet. These messages can indicate whether a message has reached its destination queue or whether the message has been retrieved. When a message is rejected, an acknowledgment message can indicate the reason for its loss.

Acknowledgments are specified by the **UserMessage.MessagePropertiesHeader.Flags** field.

If UserMesage.UserHeader.QueueManagerAddress is equal to QMGuid, the following logic MUST be applied:<70>

- If UserMessage.MessagePropertiesHeader.Flags.PA is set, the protocol MUST send a positive acknowledgment if the message reaches the destination queue. A positive acknowledgment MUST set the UserMessage.MessagePropertiesHeader.MessageClass field to MQMSG_CLASS_ACK_REACH_QUEUE.
- If the UserMessage.MessagePropertiesHeader.Flags.NA field is set, the protocol MUST send a negative acknowledgment if the message does not reach the destination queue. The negative acknowledgment MUST set UserMessage.MessagePropertiesHeader.MessageClass to one of the negative arrival acknowledgment classes specified in section 2.2.1.5.
- The acknowledgment UserMessage Packet must set the UserMessage.DestinationQueue field to the queue specified in the received message UserMessage.UserHeader.AdminQueue field. The UserMessage.UserHeader.QueueManagerAddress field MUST be set to the value of UserMessage.UserHeader.QueueManagerAddress in the received message. The UserMessage.UserHeader.Flags.DM field MUST be set to 0x0.
- The protocol MUST set UserHeader.MessagePropertiesHeader.CorrelationID to the received message UserMessage.MessagePropertiesHeader.MessageClass value and set UserMessage.UserHeader.ResponseQueue to the UserMessage.UserHeader.DestinationQueue value.

3.1.5.9 Receiving a Ping Packet

A Ping Message packet is valid if the Ping.Signature field has the value 0x5548. An invalid packet MUST be ignored.

A Ping Message packet MUST either be a request from an initiator or a response to a Ping Message packet as specified in Send Ping Packet (section 3.1.5.2.2).

A Ping Message packet is a response if Ping.QMGuid is equal to RemoteQMGuid and Ping.Cookie is equal to PingCookie. If the Ping.StateFlag value is 0x0001, the acceptor has indicated that it can accept the connection and the protocol MUST perform the steps specified in Session Initialization (section 3.1.5.2.3). Otherwise, the acceptor has refused the connection and the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

If the preceding conditions are not true, the Ping Message packet is a request from an initiator. The protocol MUST reply to the request by sending a Ping Message packet to the initiator with the following values:

- Ping.Flags.RC MUST NOT be set if the protocol will accept a session. The decision to accept or refuse a connection is implementation-dependent.<71>
- Ping.Cookie must be set to the value in the received Ping.Cookie field.
- Ping.QMGuid must be set to QMGuid.

3.1.5.10 Closing a Session

A protocol session is closed by executing the steps listed below. The protocol does not send a packet to indicate session closure; instead, the underlying transport connection is simply closed.

- Close the underlying TCP or SPX transport connection.
- Stop all timers.
- SessionState MUST be set to CLOSED.

If the OutgoingMessage table is not empty, the protocol MUST start the Session Retry Connect timer.

3.1.5.11 Accepting an Incoming Connection

When the host accepts an incoming connection from a remote host, it MUST initialize a session as specified in Session Initialization (section 3.1.3.2). The SessionState value MUST be set to WAITING_EC_MSG. The Session Initialization timer MUST be started.

3.1.5.12 Receiving Administration Acknowledgments

Administration acknowledgment messages are express UserMessage Packets that indicate that a sent message has reached its destination queue or that the message has been retrieved from its destination queue.

Administration acknowledgment messages are identified by the **UserMessage.MessagePropertiesHeader.MessageClass** field set to one of the positive or negative arrival acknowledgment classes specified in section **2.2.1.5**.

Administration acknowledgment messages MUST be processed as specified in section 3.1.5.8.

3.1.6 Timer Events

3.1.6.1 Session Retry Connect Timer Event

When the Session Retry Connect Timer fires, the protocol MUST perform the following steps if the OutgoingMessage table is non-empty:

- Initialize the session as specified in Initialization (section 3.1.3).
- Open the session to the remote queue manager as specified in Create a Protocol Session (section 3.1.5.2).

Initializing the session will result in message retransmission as specified in Other Local Events (section 3.1.7).

3.1.6.2 Session Cleanup Timer Event

When the Session Cleanup Timer fires, the protocol SHOULD apply the following logic to close an idle session:

- If SessionActive is FALSE, the protocol SHOULD close the session as specified in Closing a Session (section 3.1.5.10).
- If SessionActive is TRUE, the protocol SHOULD restart the Session Cleanup Timer.

3.1.6.3 Session Ack Wait Timer Event

The Session Ack Wait Timer event indicates a timeout while waiting for a session acknowledgment from the remote host. When the Session Ack Wait Timer fires, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

3.1.6.4 Session Ack Send Timer Event

When the Session Ack Send Timer fires, a SessionAck Packet MUST be sent to the remote host with the following values:

- The BaseHeader.Flags.IN and BaseHeader.Flags.SH fields MUST be set.
- InternalHeader.Flags.PT MUST be set to 0x3.
- SessionHeader.AckSequenceNumber MUST be set to MessageSentCount.
- SessionHeader.RecoverableMsgAckSeqNumber MUST be set to the lowest unacknowledged recoverable message sequence number that has been stored to disk.
- SessionHeader.UserMsgSequenceNumber MUST be set to MessageReceivedCount.
- SessionHeader.RecoverableMsgSeqNumber MUST be set to RecoverableMessageReceivedCount.
- SessionHeader.WindowSize MUST be set to WindowSize.

The Session Ack Send Timer MUST be restarted.

3.1.6.5 Transactional Ack Wait Timer Event

The Transactional Ack Wait Timer event indicates a timeout while waiting for a transactional OrderAck Packet from the remote host. When the Transactional Ack Wait Timer fires, the protocol MUST resend all unacknowledged transactional messages.

The protocol MUST set MessageEntry.AwaitingAck and MessageEntry.ReceivedSessionAck to FALSE for all messages in the OutgoingMessage table where MessageEntry.TxSequenceNumber is not set to 0x00000000.

The preceding step will cause all unacknowledged transactional messages to be resent to the remote queue manager.

3.1.6.6 Session Initialization Timer Event

The Session Initialization Timer event indicates a timeout while contacting the remote host during session initialization. When the Session Initialization Timer fires, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

3.1.7 Other Local Events

3.1.7.1 Outgoing Message Event

The Outgoing Message event indicates that there is a MessageEntry in the OutgoingMessage table ready to be sent to the remote queue manager. The event provides a reference to the corresponding MessageEntry.

This event is triggered when a MessageEntry exists in the OutgoingMessage table with the MessageEntry.AwaitingAck field set to FALSE and UnAckedMessageCount is less than WindowSize.

The following steps MUST be performed to send the message:

- General Processing
- Checking for Message Expiration
- Updating the UserMessage Packet
- Signing the Packet
- Encrypting the Message Body
- Sending the Packet
- Sending Trace Message

Unless specifically noted in a subsequent section, this logic MUST be applied to any UserMessage Packet packet sent.

3.1.7.1.1 General Processing

If the UserMessage.UserHeader.MessageID field is set to 0x00000000, the protocol MUST set UserMessage.UserHeader.MessageID to MessageIdOrdinal. The value of MessageIdOrdinal MUST be incremented by 1.

If the UserMessage Packet contains a TransactionHeader, the UserMessage.BaseHeader.Flags.PR field MUST be set to 0x0, and BaseHeader.Flags.PR MUST be set to 0x0.

3.1.7.1.2 Checking for Message Expiration

The values of UserMessage.UserHeader.TimeToBeReceived and UserMessage.BaseHeader.TimeToReachQueue control the message lifetime. The protocol MUST check the message for expiration before sending.

For the purpose of this section, CURRENT_TIME is defined as the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time).

If CURRENT_TIME minus UserMessage.UserHeader.SentTime is greater than UserMessage.UserHeader.TimeToBeReceived or UserMessage.BaseHeader.TimeToReachQueue, the message has expired. An expired message MUST be deleted from the OutgoingMessage table and MUST NOT be sent to the remote queue manager.

If UserMessage.UserHeader.Flags.JN is set, then an expired message SHOULD be logged locally.<72>

3.1.7.1.3 Updating the UserMessage Packet

If the UserMessage Packet contains a SessionHeader, the protocol MUST remove the SessionHeader from the UserMessage Packet and set UserMessage.BaseHeader.Flags.SH to 0x0.

If the UserMessage Packet contains a TransactionHeader and UserMessage.UserHeader.SourceQueueManager is equal to QMGuid and MessageEntry.Transmitted is FALSE, the following steps MUST be performed:

- UserMessage.TransactionHeader.TxSequenceID MUST be set to TxSequenceID.
- UserMessage.TransactionHeader.PreviousTxSequenceNumber MUST be set to TxSequenceNumber.
- UserMessage.TransactionHeader.TxSequenceNumber MUST be set to TxSequenceNumber + 1.
- MessageTable.TxSequenceNumber MUST be set to TxSequenceNumber.
- The TxSequenceNumber value MUST be incremented by 1.
- The Transactional Ack Wait Timer MUST be started.

If the UserMessage Packet contains a TransactionHeader and MessageEntry.Transmitted is TRUE, TransactionHeader.PreviousTxSequenceNumber MUST be set to TxSequenceNumber of the previous transactional message in the OutgoingMessage table. This is necessary to bridge gaps left by transactional messages that were removed from the table (for example, TimeToReachQueue expired) since the message was first sent.

The value of MessageSentCount MUST be incremented by 1.

MessageEntry.SequenceNumber MUST be set to MessageSentCount.

If UserMessage.UserHeader.Flags.DM is set to 0x1, the value of RecoverableMessageSentCount MUST be incremented by 1 and the MessageEntry.RecoverableSequenceNumber field MUST be set to RecoverableMessageSentCount and UserHeader.Flags.DM set to 0x1.

The value of UnAckedMessageCount MUST be incremented by 1.

If more than 75 percent of the time on the Session Ack Send Timer has elapsed, then BaseHeader.Flags.SH must be set and a SessionHeader MUST be inserted into the UserMessage Packet. The following fields MUST be set in the SessionHeader:

- AckSequenceNumber MUST be set to MessageSentCount.
- RecoverableMsgAckSeqNumber MUST be set to the lowest unacknowledged recoverable message sequence number that has been stored to disk.
- UserMsgSequenceNumber MUST be set to MessageReceivedCount.
- RecoverableMsgSeqNumber MUST be set to RecoverableMessageReceivedCount.
- WindowSize MUST be set to WindowSize.

The MessageEntry.RecoverableSequenceNumber field MUST be set to RecoverableMessageSentCount.

Set MessageEntry.AwaitingAck to TRUE.

The protocol MUST start the Session Ack Wait Timer if it is in the stopped state.

3.1.7.1.4 Signing the Packet

If SecurityHeader.Flags.AU is set, the packet MUST be signed. The following steps MUST be performed to sign the packet:

1. The protocol MUST compute a hash of the fields specified by the SecurityHeader.Flags.AS field by using the hash algorithm specified by the MessagePropertiesHeader.HashAlgorithm field.

2. The SecurityHeader.SecurityData.Signature field MUST be set to the value of the hash encrypted using RSA and the sender private key.

3.1.7.1.5 Encrypting the Message Body

If SecurityHeader.Flags.EB is set, the message body MUST be encrypted. The following steps MUST be performed to encrypt the message body:

The protocol MUST encrypt the MessagePropertiesHeader.MessageData.MessageBody field by using the encryption algorithm specified by MessagePropertiesHeader.EncryptionAlgorithm and a randomly generated encryption key. The MessagePropertiesHeader.PrivacyLevel field specifies the encryption key size that MUST be used. The size of the MessageBody field MUST be adjusted if the encrypted data is a different size than the original field. The MessagePropertiesHeader.AllocationBodySize MUST be set to the size of the encrypted data. The encryption key above MUST be encrypted using RSA and the public key of the destination queue manager in RemoteQMPublicKey. The SecurityData.EncryptionKey field MUST be set to the resulting encrypted data, and the EncryptionKeySize field set to the size of that data.

3.1.7.1.6 Sending the Packet

The UserMessage Packet MUST be sent to the remote queue manager using the TCP/SPX connection associated with the protocol session.

The protocol MUST set MessageEntry.Transmitted to TRUE.

3.1.7.1.7 Sending Trace Message

If the BaseHeader.Flags.TR field is set, the protocol MUST send a report message to the queue specified by the DebugHeader.QueueIdentifier field. Report messages are utilized by application logic to track the delivery of sent messages.

To send a report message, the protocol MUST send a UserMessage Packet with the following field values:

The MessagePropertiesHeader.MessageClass field MUST be set to MQMSG_CLASS_REPORT, the UserMessage.DestinationQueue set to DebugHeader.QueueIdentifier, and UserHeader.Flags.DM set to 0x0.

The MessagePropertiesHeader.MessageData.Label MUST be set to a Unicode string in the format specified by the following ABNF rule:

```
label = qm-id %x3A message-id %x3A hops SP "received by" SP computer
        SP "at" SP time-date %x0000
qm-id = 4HEXDIG ; MUST be set to the first four hexadecimal digits
                ; of the source queue identifier
message-id = 8HEXDIG ; hexadecimal form of the UserHeader.MessageID
                     ; field
hops = 2HEXDIG ; MUST be set to the UserHeader.Flags.RC field
computer = GUID ; MUST be set to UserHeader.SourceQueueManager field
time-date = hour SP ("AM" / "PM") SP date
hour  =  2DIGIT ":" 2DIGIT [":" 2DIGIT] ; ANSI and Military
date =  day "," month SP 2DIGIT SP year; day, month day year
month  = "Jan" / "Feb" / "Mar" / "Apr" / "May" / "Jun"
         / "Jul" / "Aug" / "Sep" / "Oct" / "Nov" / "Dec"
day = "Mon" / "Tue" / "Wed" / "Thu" / "Fri" / "Sat" / "Sun"
```

```
year = 2DIGIT
GUID =  8HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 4HEXDIG "-" 12HEXDIG
        ; A GUID the form XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
        ; Where each X is a Hex digit
```

The MessagePropertiesHeader.MessageData.MessageBody MUST be set to a Unicode string in the format specified by the following ABNF rule:

```
Report = "<MESSAGE ID>" id "</MESSAGE ID>" CR LF
         "<TARGET QUEUE>" queue "</TARGET QUEUE>" CR LF
         "<NEXT HOP>" address "</NEXT HOP>" CR LF
         "<HOP COUNT>" hops " </HOP COUNT>" CR LF
id = 8HEXDIG ; MUST be set to UserHeader.MessageID field
queue = queue-format; MUST be set to UserHeader.DestinationQueue
                    ; field
address = ip-address; MUST be set to IP address of remote host.
hops = 2HEXDIG ;  MUST be set to the UserHeader.Flags.RC field
ip-address=(IPv6address / IPv4address) ; as defined in [RFC3986]
```

The ABNF rule queue-format is as specified in [MS-MQMQ] section 2.1.

3.1.7.2 Message Removed from Destination Queue

Whenever a message is removed from a final destination queue, the protocol MUST send an acknowledgment messages under the conditions described in this section.

Message removal from a destination queue could be the result of the message being read by a higher-layer application, the UserMessage.UserHeader.TimeToBeReceived period having elapsed, or the queue being deleted or purged as specified in Administrator Purges a Queue (section 3.1.4.4). Operations that occur on messages in a destination queue are outside the definition of this protocol; however, the protocol must ensure that messages are tracked and that the acknowledgment logic below is applied.

3.1.7.2.1 Administration Acknowledgment

If administration acknowledgments are requested, a message is sent to the administration queue specified in the message when it is removed from a destination queue. Administration acknowledgment messages are contained inside system-generated express UserMessage Packets.

If UserMessage.MessagePropertiesHeader.Flags.PR is set, the protocol MUST send a positive acknowledgment when the message is retrieved from the destination queue before its UserMessage.UserHeader.TimeToBeReceived expires. A positive acknowledgment MUST set UserMessage.MessagePropertiesHeader.MessageClass to MQMSG_CLASS_ACK_RECEIVE.

If UserMessage.MessagePropertiesHeader.Flags.NR is set, the protocol MUST send a negative acknowledgment if the message is not retrieved from the destination queue before its UserMessage.UserHeader.TimeToBeReceived expires. A negative acknowledgment MUST set UserMessage.MessagePropertiesHeader.MessageClass to one of the negative read acknowledgment classes specified in **Message Class Identifiers (section 2.2.1.5)** to indicate the failure.

3.1.7.2.2 Final Acknowledgment

If UserMessage.TransactionHeader.Flags.FA is set, the protocol MUST send a FinalAck Packet message to the original sender when the message is removed from the destination queue.

The FinalAck.TxSequenceID field MUST be set to UserMessage.TransactionHeader.TxSequenceID, FinalAck.TxSequenceNumber MUST be set to UserMessage.TransactionHeader.TxSequenceNumber, FinalAck.TxPreviousSequenceNumber MUST be set to IncomingTxSequenceNumber, and FinalAck.MessageID MUST be set to UserMessage.UserHeader.MessageID.

The FinalAck Packet MUST be addressed to the sender order queue by setting the UserMessage.DestinationQueue field to "TCP:Host\ PRIVATE$\order_queue" or "SPX:Host\ PRIVATE$\order_queue where Host is the UserMessage.UserHeader.SourceQueueManager field value. The UserMessage.MessagePropertiesHeader.MessageClass MUST be set to an acknowledgment value specified in Session Ack Send Timer Event (section 3.1.6.4).

The UserMessage.MessagePropertiesHeader.MessageClass field MUST be set to a value specified in **Message Class Identifiers (section 2.2.1.5)**.

3.1.7.3 Handling a Network Disconnect

When the underlying transport indicates a disconnect, the protocol MUST close the session as specified in Closing a Session (section 3.1.5.10).

4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the Message Queuing (MSMQ): Message Queuing Binary Protocol.

4.1 Session Initialization and Express Message Example

The following Message Queuing (MSMQ): Message Queuing Binary Protocol packet sequence demonstrates session initialization and transfer of an express message between two queue managers. This example follows the "Session with Express Messages Sent" scenario specified in Session with Express Messages Sent (section 3.1.1.5.1) except that only a single UserMessage Packet is sent.

4.1.1 FRAME 1: Ping Request

```
Client -> Server: Ping packet
 - StateFlag: 2000 (0x7D0)
    Client: (..............1) - Client is independent
    Server: (..............0.) - Server will accept new connection
    Reserved: (00000111110100..) - Reserved
   Signature: 21832 (0x5548)
   Cookie: 4 (0x4)
   QMGUID: {C626EA11-E6B6-9749-9595-9150557358D1}
```

4.1.2 FRAME 2: Ping Response

```
 Server -> Client : Ping packet
 - StateFlag: 45717 (0xB295)
    Client: (..............1) - Client is independent
    Server: (..............0.) - Server will accept new connection
    Reserved: (10110010100101..) - Reserved
   Signature: 21832 (0x5548)
   Cookie: 4 (0x4)
   QMGUID: {FCA09E90-7890-4544-8F11-394C43CD8907}
```

4.1.3 FRAME 3: Establish Connection Request

```
Client -> Server : EstablishConnection Packet
 - MSMQBaseHeader:
    VersionNumber: 16 (0x10)
    Reserved: 192 (0xC0)
  - FlagsBaseHeader: 11 (0xB)
     MessagePriority: (............011) - Message priority = 3
     InternalMessage: (............1...) - Internal message
     SessionHeader: (...........0....) - Session header not included
     DebugSession: (..........0.....) - Debug header not included
     Reserved1: (........00......) - Reserved
     MessageTraceable: (.......0........) - Tracing disabled
     Reserved2: (0000000.........) - Reserved
    Signature: 1380927820 (0x524F494C)
    PacketSize: 572 (0x23C)
```

```
   TimeToReachQueue: 4294967295 (0xFFFFFFFF)
 - MSMQInternalHeader:
    Reserved: 0 (0x0)
  - FlagsInternalHeader: 2 (0x2)
     PacketType: (............0010) -
     Session: (...........0....) - Session valid
     Reserved: (00000000000.....) - Reserved
  + EstablishConnection:
```

4.1.4 FRAME 4: Establish Connection Response

```
Server -> Client : EstablishConnection packet
- MSMQBaseHeader:
   VersionNumber: 16 (0x10)
   Reserved: 90 (0x5A)
 - FlagsBaseHeader: 11 (0xB)
    MessagePriority: (.............011) - Message priority = 3
    InternalMessage: (............1...) - Internal message
    SessionHeader: (...........0....) - Session header not included
    DebugSession: (..........0.....) - Debug header not included
    Reserved1: (........00......) - Reserved
    MessageTraceable: (.......0........) - Tracing disabled
    Reserved2: (0000000.........) - Reserved
   Signature: 1380927820 (0x524F494C)
   PacketSize: 572 (0x23C)
   MessageLife: 4294967295 (0xFFFFFFFF)
 - MSMQInternalHeader:
    Reserved: 0 (0x0)
  - FlagsInternalHeader: 2 (0x2)
     PacketType: (............0010) -
     Session: (...........0....) - Session valid
     Reserved: (00000000000.....) - Reserved
  + EstablishConnection:
```

4.1.5 FRAME 5: Connection Parameters Request

```
Client -> Server : ConnectionParameters packet
- MSMQBaseHeader:
  VersionNumber: 16 (0x10)
  Reserved: 192 (0xC0)
  - FlagsBaseHeader: 11 (0xB)
    MessagePriority: (.............011) - Message priority = 3
    InternalMessage: (............1...) - Internal message
    SessionHeader: (...........0....) - Session header not included
    DebugSession: (..........0.....) - Debug header not included
    Reserved1: (........00......) - Reserved
    MessageTraceable: (.......0........) - Tracing disabled
    Reserved2: (0000000.........) - Reserved
  Signature: 1380927820 (0x524F494C)
  PacketSize: 32 (0x20)
  MessageLife: 4294967295 (0xFFFFFFFF)
- MSMQInternalHeader:
  Reserved: 0 (0x0)
```

```
    - FlagsInternalHeader: 3 (0x3)
      PacketType: (............0011) - ConnectionParameters packet
      Session: (...........0....) - Session valid
      Reserved: (00000000000.....) - Reserved
  - ConnectionParametersHeader:
    RecoverAckTimeout: 1496 (0x5D8)
    AckTimeout: 120000 (0x1D4C0)
    Reserved: 0 (0x0)
    WindowSize: 32 (0x20)
```

4.1.6 FRAME 6: Connection Parameters Response

```
Server -> Client : ConnectionParameters packet
- MSMQBaseHeader:
  VersionNumber: 16 (0x10)
  Reserved: 192 (0xC0)
  - FlagsBaseHeader: 11 (0xB)
    MessagePriority: (............011) - Message priority = 3
    InternalMessage: (............1...) - Internal message
    SessionHeader: (...........0....) - Session header not included
    DebugSession: (..........0.....) - Debug session not included
    Reserved1: (........00......) - Reserved
    MessageTraceable: (.......0........) - Message is not traceable
    Reserved2: (0000000.........) - Reserved
  Signature: 1380927820 (0x524F494C)
  PacketSize: 32 (0x20)
  MessageLife: 4294967295 (0xFFFFFFFF)
- MSMQInternalHeader:
  Reserved: 0 (0x0)
  - FlagsInternalHeader: 3 (0x3)
    PacketType: (............0011) -
    Session: (...........0....) - Session valid
    Reserved: (00000000000.....) - Reserved
- ConnectionParametersHeader:
  RecoverAckTimeout: 1496 (0x5D8)
  AckTimeout: 120000 (0x1D4C0)
  Reserved: 0 (0x0)
  WindowSize: 32 (0x20)
```

4.1.7 FRAME 7: User Message

```
Client -> Server : UserMessage packet
- BaseHeader:
   VersionNumber: 16 (0x10)
   Reserved: 0 (0x0)
 - Flags: 3 (0x3)
    MessagePriority: (............011) - Message priority = 3
    InternalMessage: (............0...) - UserMessge packet
    SessionHeader: (...........0....) - Session header not included
    DebugSession: (..........0.....) - Debug session not included
    Reserved: (........00......) - Reserved
    MessageTraceable: (.......0........) - Tracing disabled
    Reserved2: (0000000.........) - Reserved
   Signature: 1380927820 (0x524F494C)
   PacketSize: 268 (0x10C)
```

```
   MessageLife: 345600 (0x54600)
- UserHeader:
  SourceQueueManager: {C626EA11-E6B6-9749-9595-9150557358D1}
  QueueManagerAddress: {00000000-0000-0000-0000-000000000000}
  TimeToBeReceived : 4294967295 (0xFFFFFFFF)
  SentTime: 1141966310 (0x441105E6)
  MessageID: 2286 (0x8EE)
  - Flags: 2628608 (0x281C00)
    RoutingSvrs: (.........................00000)
      - 0 MSMQ routing servers have handled this packet
    DeliveryMode: (.........................00.....)
      - Express delivery mode
    Reserved: (.........................0.......) - Reserved
    Journaling: (.......................00........) - Disabled
    DestinationQueue: (...................111..........) - Direct queue
    AdminQueue: (................000.............) - None
    ResponseQueue: (.............000.................) - None
    SecurityHdr: (............1...................)
      - SecurityHeader included
    TransactionHdr: (............0.....................)
      - TransactionHeader not included
    MessagePropertyHdr: (..........1.....................)
      - MessagePropertyHeader included
    Connector: (.........0.......................)
      - No ConnectorType field
    MultiQueueFormatHdr: (........0.......................)
     - MultiQueueFormatHeader not included
    Multicast: (.......0.........................)
      - Message not part of multicast operation
    Reserved: (....000.........................) - Reserved
    SoapHdr:(...0.............................) - SoapHeader not included
    Reserved: (000.............................) - Reserved
  - DestinationQueue:
   - DirectQueue:
     Length: 50 (0x32)
     Value: os:conradclh1\private$\q
     Padding: 0 Bytes
- SecurityHeader:
  - Flags: 1 (0x1)
    SenderIDType: (............0001) - SID
    Authenticated: (............0....) - Not authenticated
    Encrypted: (..........0.....) - Not encrypted
    DefProv: (.........1......) - Default provider
    SecInfoEx: (........0.......) - SecurityData not present
    LevelOfAuthentication: (....0000........) - None
    Unused: (0000............)
  SenderIdSize: 28 (0x1C)
  EncryptionKeySize: 0 (0x0)
  SignatureSize: 0 (0x0)
  SenderCertSize: 0 (0x0)
  ProviderInfoSize: 0 (0x0)
  - SecurityData:
    - SenderID: S-1-5-21-2268239182-922028006-1257416481-1003
      Revision: 1 (0x1)
      SubAuthorityCount: 5 (0x5)
    - IdentifierAuthority: {0,0,0,0,0,5} (0x5) [SECURITY_NT_AUTHORITY]
       Value: 0 (0x0)
       Value: 0 (0x0)
       Value: 0 (0x0)
       Value: 0 (0x0)
       Value: 0 (0x0)
       Value: 5 (0x5)
    - SubAuthority: 5 Sub-Authorities
       SubAuthorityValue: 21 (0x15)
       SubAuthorityValue: 2268239182 (0x8732954E)
```

```
        SubAuthorityValue: 922028006 (0x36F507E6)
        SubAuthorityValue: 1257416481 (0x4AF2A721)
        SubAuthorityValue: 1003 (0x3EB)
  - MessagePropertiesHeader:
    - Flags: Acknowledgment sent
      PositiveArriveAck: (.......0) - None
      PositiveReceiveAck: (......0.) - None
      NegativeDeliveryAck: (.....0..) - None
      NegativeReceiveAck: (....0...) - None
      Reserved: (0000....) - Reserved
    LabelLength: 15 (0xF)
    MessageClass: 0 (0x0)
    CorrelationID: Binary Large Object (20 Bytes)
      (20 bytes, all values zero)
    BodyType: VTBSTR [String data in unicode]
    ApplicationTag: 0 (0x0)
    MessageSize: 20 (0x14)
    AllocatedBodySize: 20 (0x14)
    PrivacyLevel: No encryption
    HashAlgorithm: 32772 (0x8004)
    EncryptionAlgorithm: 26114 (0x6602)
    ExtensionSize: 0 (0x0)
    - MessageData:
      Title: Binary Large Object (15 Bytes)
      Extension: Binary Large Object (0 Bytes)
      Body: Binary Large Object (20 Bytes)
      Padding: 1 Bytes
```

4.1.8 FRAME 8: Session Acknowledgment

```
Server -> Client : SessionAck Packet
 - BaseHeader:
   VersionNumber: 16 (0x10)
   Reserved: 205 (0xCD)
   - FlagsBaseHeader: 27 (0x1B)
     MessagePriority: (.............011) - Message priority = 3
     InternalMessage: (............1...) - Internal message
     SessionHeader: (...........1....) - Session header included
     DebugSession: (..........0.....) - Debug session not included
     Reserved1: (........00......) - Reserved
     MessageTraceable: (.......0........) - Tracing disabled
     Reserved2: (0000000.........) - Reserved
   Signature: 1380927820 (0x524F494C)
   PacketSize: 36 (0x24)
   MessageLife: 4294967295 (0xFFFFFFFF)
 - InternalHeader:
   Reserved: 0 (0x0)
   - FlagsInternalHeader: 1 (0x1)
     PacketType: (............0001) - SessionAck packet
     Session: (............0....) - Session valid
     Reserved: (000000000000.....) - Reserved
 - SessionHeader:
   AckSequenceNumber: 1 (0x01)
   RecoverableMsgAckSeqNumber: 0 (0x0)
   RecoverableMsgAckFlags: 0 (0x0)
   UserMsgSequenceNumber: 0 (0x0)
   RecoverableMsgSeqNumber: 0 (0x0)
   WindowSize: 32 (0x20)
```

```
Reserved: 0 (0x0)
```

5 Security

The following sections specify security considerations for implementers of the Message Queuing (MSMQ): Message Queuing Binary Protocol.

5.1 Security Considerations for Implementers

A sender should include a digital certificate and request authentication when sending a UserMessage Packet. A sender may request encryption of the message body to ensure message privacy. Use of the AES encryption algorithm is recommended for the best encryption strength.

Authentication and encryption are not supported when a message is sent to a queue using a direct format name. The information in a UserMessage Packet that is sent using a direct format name is susceptible to tampering.

A receiver should authenticate received UserMessage Packets by verifying an included digital signature and certificate. The receiver SHOULD perform an access check to authorize the message sender to the destination queue.

Windows stores queue manager public keys in Active Directory.<73> An implementer should store public keys in Active Directory or a compatible system as specified in [MS-ADTS].

5.2 Index of Security Parameters

No security parameters are defined for this protocol.

6 Appendix A: Windows Behavior

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

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

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

<1> Section 1.3.4: Windows Server 2008 does not perform source journaling for messages sent to internal queues.

<2> Section 2.1.1: The SPX/IPX protocol is supported only on Windows NT.

<3> Section 2.1.1: The Windows implementation utilizes the Windows Sockets API for TCP/SPX connections. The Windows Socket API is responsible for operations such as selection of the source port used by an initiator and listening/accepting connections by the acceptor.

<4> Section 2.1.1: The Windows implementation utilizes the Windows Sockets API for TCP/SPX connections. The Windows Socket API is responsible for operations such as selection of the source port used by an initiator and listening/accepting connections by the acceptor.

<5> Section 2.2.1.2: Windows sets the timestamp to the time of queue manager install or last system restore. The value is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time). The timestamp can represent dates up to January 19, 2038, after which the representation will overflow. The protocol does not contain support to handle an overflow, and incorrect delivery of transactional messages may occur in that condition. A system restore is the process of recovering after the data in the persistent state store has been lost or is intentionally overwritten by restoring a backup. Setting the timestamp to the current system time after a system restore ensures that transactional messages sent from that time forward are processed correctly.

<6> Section 2.2.1.4: Padding bytes contain uninitialized values.

<7> Section 2.2.2.3: Negative source journaling is not supported by Windows NT. Windows Server 2008 does not perform source journaling for messages sent to internal queues.

<8> Section 2.2.2.3: Windows stores a copy of the message in the local dead-letter queue on failure to deliver. Transactional messages are copied to the local transactional dead-letter queue. The dead-letter queue is a system-generated queue and is implementation-dependent. Windows Server 2008 does not perform source journaling for messages sent to internal queues.

The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<9> Section 2.2.2.3: Windows copies the message to the system journal queue. The journal queue is a system-generated queue and is implementation-dependent. Windows Server 2008 does not perform source journaling for messages sent to internal queues.

The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<10> Section 2.2.2.3: Padding bytes contain uninitialized values.

<11> Section 2.2.2.3: Padding bytes contain uninitialized values.

<12> Section 2.2.2.3: Padding bytes contain uninitialized values.

<13> Section 2.2.2.4: The Windows limit on overall size of the UserMessage Packet is 4 MB because of the storage implementation. This means that the variable-length fields must be handled carefully to prevent the size of the overall UserMessage Packet from exceeding the limit.

<14> Section 2.2.2.4: The Windows limit on overall size of the UserMessage Packet is 4 MB because of the storage implementation. This limitation means that the variable-length fields must be handled carefully to prevent the size of the overall UserMessage Packet from exceeding the limit.

<15> Section 2.2.2.4: 40-bit and 128-bit encryption is not supported by Windows NT. AES encryption is not supported by Windows XP, Windows Server 2003, Windows 2000, or Windows NT.

<16> Section 2.2.2.4: The SHA1 hash algorithm is not supported by Windows 2000 and Windows NT.

<17> Section 2.2.3.1: The Windows default for RecoverableAckTimeout is the roundtrip time, in milliseconds, for the EstablishConnection Packet exchange multiplied by 8. The roundtrip time is measured from the time the client sends the EstablishConnection Packet message until the EstablishConnection Packet response is received from the server. If the resulting value is outside the valid range for the field, the value is set to the closest limit.

<18> Section 2.2.3.1: The default setting is 20,000 milliseconds on a LAN. The Microsoft implementation provides a registry key at HKEY_LOCAL_MACHINE\software\microsoft\msmq\parameters\RoundTripDelay that may be used by the client to specify additional time beyond the default value.

<19> Section 2.2.3.1: The Windows default WindowSize value is 32.

<20> Section 2.2.4.1: The Windows implementation calls the GetVersionEx() Win32 API to determine client or server platform.

<21> Section 2.2.4.1: The **EstablishConnectionHeader.OperatingSystem.QS** field value does not affect the transport settings. The Guaranteed Quality of Service (GQoS) values exchanged are only informational.

<22> Section 2.2.5: The SPX/IPX protocol is supported only on Windows NT.

<23> Section 2.2.6: The SPX/IPX protocol is supported only on Windows NT.

<24> Section 2.2.8.1: The MultiQueueFormatHeader is not supported by Windows 2000 or Windows NT.

<25> Section 2.2.8.4: The Windows default **WindowSize** value is 32.

<26> Section 2.2.8.4: If a destination queue manager encounters a memory allocation error while processing a message, it sets the window size to 1 in the outgoing SessionHeader header, which

causes the destination queue manager to reduce its window to the same size. The source queue manager doubles the window size every 30 seconds until the window size returns to the default value of 32.

<27> Section 2.2.8.5: The **TransactionHeader.Flags.FM** and **TransactionHeader.Flags.LM** fields are not supported by Windows NT.

<28> Section 2.2.8.6: Windows stores the queue manager public key in the MSMQ-Encrypt-Key attribute of the queue manager's MSMQ-Configuration object in Active Directory (AD). Directory Service Schema is as specified in [MS-MQMQ]. During the queue manager startup, this value is queried once using the **LDAP** protocol.

<29> Section 2.2.8.6: The Windows limit on overall size of the UserMessage Packet is 4 MB because of the storage implementation. This limitation means that the variable-length fields must be handled carefully to prevent the size of the overall UserMessage Packet from exceeding the limit.

<30> Section 2.2.8.7: The SoapHeader is not intended to be used with the Message Queuing (MSMQ): Message Queuing Binary Protocol. It is only present when an MSMQ user application sets the PROPID_M_SOAP_BODY, PROPID_M_SOAP_ENVELOPE, or PROPID_M_SOAP_HEADER MSMQ message properties to application-defined values. These values are simply passed through if present and are not evaluated.

<31> Section 2.2.8.8: Windows does not support message tracing when the message contains a SoapHeader or MultiQueueFormatHeader. To prevent a DebugHeader from being added to the packet by code that subsequently processes the packet, the sending protocol adds a DebugHeader with the Flags field set to 0. This ensures that tracing is not enabled for the UserMessage Packet later.

<32> Section 2.2.9: The unused fields are uninitialized data in the Windows implementation.

<33> Section 2.2.9: The Windows default value is 0x00000000.

<34> Section 3.1.1.1.2: Each Windows client and server generates a unique GUID upon setup and stores it durably as a binary value under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ\Parameters\MachineCache\QMId registry key.

<35> Section 3.1.1.1.2: The Windows implementation persistently stores up to the last 10,000 MessageIdentifier values. Any values greater than 30 minutes old are discarded.

<36> Section 3.1.1.1.2: Windows initializes the MessageIdOrdinal value to 0x000000 when it does not exist in persistent storage.

<37> Section 3.1.1.1.2: Windows performs no special action when the MessageIdOrdinal value rolls over. Values are reused after the rollover; however, the rollover condition does not affect message delivery guarantees because the MessageIDHistoryTable length is sufficiently short. The Windows implementation persistently stores up to the last 10,000 MessageIdentifier values. Any values greater than 30 minutes old are discarded.

<38> Section 3.1.1.1.3: Each Windows client and server generates a unique **GUID**, as specified in [MS-DTYP], upon setup and stores it durably as a binary value under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ\Parameters\MachineCache\QMId registry key.

<39> Section 3.1.1.1.3: The Windows default WindowSize value is 32.

<40> Section 3.1.1.3: The Windows implementation creates a new sequence when there are no unacknowledged transactional messages.

<41> Section 3.1.1.4.1: Windows discards express, recoverable, and transactional messages that have been acknowledged.

<42> Section 3.1.1.4.1: Windows discards express, recoverable, and transactional messages that have been acknowledged.

<43> Section 3.1.1.4.2: The Windows implementation waits a minimum of 500 milliseconds before sending an OrderAck Packet. If a new transactional message is received during this period, the OrderAck Packet is delayed another 500 milliseconds. This process continues and can delay the OrderAck Packet up to 10 seconds.

<44> Section 3.1.1.4.2: Windows discards express, recoverable, and transactional messages that have been acknowledged.

<45> Section 3.1.1.5.1: Windows Server 2008 and Windows Vista do not utilize the Ping Message mechanism when connecting to remote queue managers.

<46> Section 3.1.2.1: The Microsoft implementation sets the Session Initialization Timer to a value in milliseconds equal to 60000 + (2 * RoundTripDelay). The RoundTripDelay value is contained in the registry key at HKEY_LOCAL_MACHINE\software\microsoft\msmq\parameters\RoundTripDelay. If the registry key does not exist then the RoundTripDelay value is 0.

<47> Section 3.1.2.2: The Windows default value for SessionCleanup timer is 300,000 milliseconds for the client and 120,000 milliseconds for the server.

<48> Section 3.1.2.3: The Windows default reconnection timeout is 5 minutes.

<49> Section 3.1.2.6: The Windows default timeout is 30 seconds. The timeout grows as the number of sequential timeouts occur. The first, second, and third timeouts have periods of 30 seconds. The fourth, fifth, and sixth timeouts are 5 minutes. The seventh, eighth, and ninth timeouts are 30 minutes, and thereafter, the timeout period is 6 hours.

<50> Section 3.1.3.2: Windows sets the timestamp to the time of queue manager install or last system restore. The value is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal Time).

<51> Section 3.1.3.2: The Windows default WindowSize value is 32.

<52> Section 3.1.3.2: The default setting is 20,000 milliseconds on a LAN. The Microsoft implementation provides a registry key at HKEY_LOCAL_MACHINE\software\microsoft\msmq\parameters\RoundTripDelay that may be used by the client to specify additional time beyond the default value.

<53> Section 3.1.3.2: The Windows default for RecoverableAckTimeout is the roundtrip time, in milliseconds, for the EstablishConnection Packet exchange multiplied by 8. The roundtrip time is measured from the time the client sends the EstablishConnection Packet message until the EstablishConnection Packet response is received from the server. If the resulting value is outside the valid range for the field, the value is set to the closest limit.

<54> Section 3.1.3.2: Windows stores the MSMQ internal certificate used by the Crypto API in a Certificate Store under HKCU\Software\Microsoft\MSMQ\CertStore. The associated keys are located in the registry at HKCU\Software\Microsoft\Cryptography\UserKeys\MSMQ. MSMQ always uses the Crypto API to access the encryption keys.

<55> Section 3.1.3.2: Windows stores the remote queue manager public key in the MSMQ-Encrypt-Key attribute of the remote queue manager MSMQ-Configuration object in Active Directory (AD). The Directory Service Schema is as specified in [MS-MQMQ] section 2.6.

<56> Section 3.1.3.2: Each Windows client and server generates a unique **GUID**, as specified in [MS-DTYP], upon setup and stores it durably as a binary value under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ\Parameters\MachineCache\QMId registry key.

<57> Section 3.1.3.2: The Windows default value is 0x00000000.

<58> Section 3.1.5.2.1: Windows determines the host by querying the directory service using either LDAP or the Directory Service Protocol. The Directory Service is as specified in [MS-MQMA]; the Directory Service Schema is as specified in [MS-MQMQ]; and the Directory Service Protocol is as specified in [MS-MQDS].

A public queue format name contains a queue GUID value that can be used to query a directory service by using the LDAP protocol to identify the host NetBIOS name.

The following is an example of a public format name containing a queue GUID:

PUBLIC=BD6956E3-1A28-49E0-8F95-0CE757B537D3

The NetBIOS name of the computer that hosts the above queue can be retrieved using the following LDAP query:

```
Base DN: DC=ACME,DC=com
Filter: (&(objectclass=MSMQQueue)
(objectGuid=\E3Vi\BD\28\1A\E01\8F\95\0C\E7W\B57\D3))
```

Query Result:

```
distinguishedName:
CN=testqueue,CN=msmq,CN=MSMQSERVER1,CN=Computers,
DC=ACME,DC=com;
```

The resulting "distinguishedName" attribute contains the NetBIOS name of the host in the node above the "CN=msmq" node. In the above example, the host NetBIOS name is "MSMQSERVER1".

The Windows implementation performs the above query each time that a protocol session is established to the destination host computer. A protocol session may be used to send multiple messages to the destination host; however, the host NetBIOS name is queried only once when the protocol session is established.

<59> Section 3.1.5.2.2: Windows Server 2008 and Windows Vista do not utilize the Ping Message mechanism when connecting to remote queue managers. The Ping Message mechanism is always utilized by Windows Server 2003, Windows XP, Windows 2000, and Windows NT. The mechanism simply provides the initiator with information on whether a connection will likely be accepted. The decision to accept or not to accept the connection in the response Ping Message is not a guarantee, because the acceptor state could change before it receives an Establish Connection packet from the initiator. An implementation can choose not to send a Ping Message packet with no adverse side effects.

<60> Section 3.1.5.4.1: Windows sets the ConnectionParametersHeader.WindowSize field to the value of ConnectionParametersHeader.WindowSize in the received packet. An implementation is free to specify a different value.

<61> Section 3.1.5.5.4: Windows copies the message to the system journal queue. The journal queue is a system-generated queue and is implementation-dependent. The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<62> Section 3.1.5.6: Windows performs positive source journaling for transactional messages after the corresponding FinalAck Packet message is received.

<63> Section 3.1.5.6: Windows stores a copy of the message in the local dead-letter queue on failure to deliver. Transactional messages are copied to the local transactional dead-letter queue. The dead-letter queue is a system-generated queue and is implementation-dependent. The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<64> Section 3.1.5.6: The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<65> Section 3.1.5.6: The Windows implementation creates a new sequence when there are no unacknowledged transactional messages.

<66> Section 3.1.5.7: The Windows implementation creates a new sequence when there are no unacknowledged transactional messages.

<67> Section 3.1.5.8.3: MSMQ performs an AccessCheck against the destination queue Access Control List (ACL).

<68> Section 3.1.5.8.5: Windows stores a copy of the message in the local dead-letter queue on failure to deliver. Transactional messages are copied to the local transactional dead-letter queue. The dead-letter queue is a system-generated queue and is implementation-dependent. The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<69> Section 3.1.5.8.6: The Windows implementation waits a minimum of 500 milliseconds before sending an OrderAck Packet. If a new transactional message is received during this period, the OrderAck Packet is delayed another 500 milliseconds. This process continues and can delay the OrderAck Packet up to 10 seconds.

<70> Section 3.1.5.8.10: Windows Server 2008 does not send administration acknowledgments for messages sent to internal queues.

<71> Section 3.1.5.9: The server uses the NT Client Access License (CAL) number to verify licenses of independent clients. Multiple connections from the same independent client are counted as one.

<72> Section 3.1.7.1.2: Windows stores a copy of the message in the local dead-letter queue on failure to deliver. Transactional messages are copied to the local transactional dead-letter queue. The dead-letter queue is a system-generated queue and is implementation-dependent. The Windows dead-letter queue, transactional dead-letter queue, and journal queue are named "Deadletter$", "XactDeadletter", and "Journal$", respectively.

<73> Section 5.1: Windows stores the queue manager public key in the MSMQ-Encrypt-Key attribute of the queue manager's MSMQ-Configuration object in Active Directory (AD). The Directory

Service Schema is as specified in [MS-MQMQ] section 2.4. This value is queried using the LDAP protocol once during the queue manager startup.

7 Index